Driver Libraries Help

MPLAB Harmony Integrated Software Framework

Volume V: MPLAB Harmony Framework Reference

This volume provides API reference information for the framework libraries included in your installation of MPLAB Harmony.

Description

This volume is a programmer reference that details the interfaces to the libraries that comprise MPLAB Harmony and explains how to use the libraries individually to accomplish the tasks for which they were designed.

Driver Libraries Help

This section provides descriptions of the Driver libraries that are available in MPLAB Harmony.

Driver Library Overview

This topic provides help for the MPLAB Harmony driver libraries. It includes a general driver usage overview, as well as sections providing a programmer's reference for each driver that describes its interface and explains how to use it.

Introduction

Introduces MPLAB Harmony device drivers and explains common usage concepts.

Description

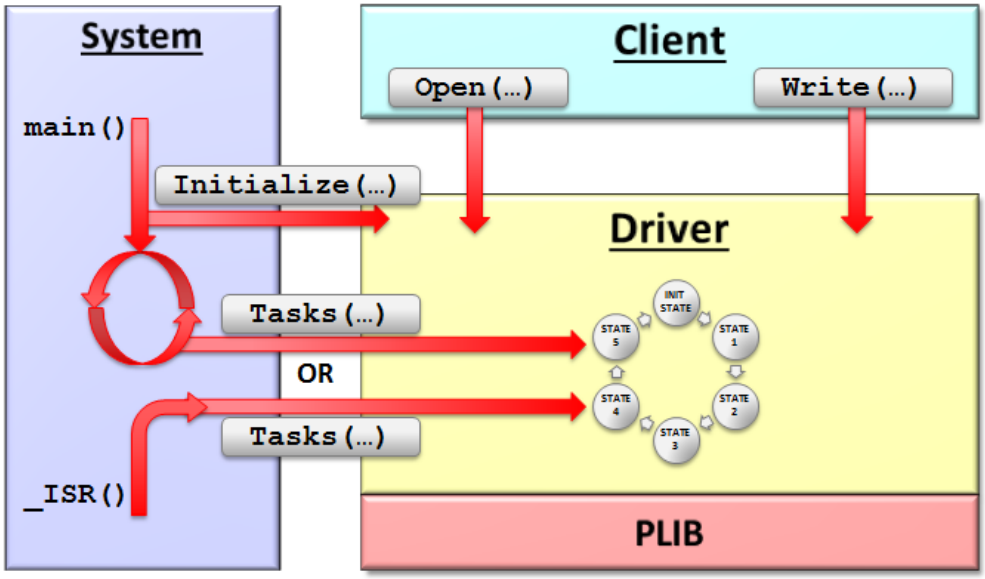
MPLAB Harmony device drivers (usually referred to as "drivers") provide simple, highly abstracted C-language interfaces to peripherals and other resources. A driver's interface allows applications and other client modules to easily interact with the peripheral it controls using consistent usage models. Some functions are similar on all drivers, while other functions are unique to a particular type of driver or peripheral. However, driver interface functions are generally independent of the details of how a given peripheral is implemented on any specific hardware or of how many instances of that peripheral exist in a given system.

Drivers normally utilize MPLAB Harmony Peripheral Libraries (PLIBs) to access and control peripheral hardware that is built into the processor (and is directly addressable by it). However, drivers can also support external peripheral hardware by calling another driver that directly controls a built-in peripheral to which the external peripheral is connected. For example, an SD Card driver may use a SPI driver to access its external SD Card Flash device. A driver may even be completely abstracted away from any hardware (utilizing no peripheral hardware at all), simply controlling some software resource (such as a buffer queue) or providing some service (such as data formatting or encryption). Using this method, driver and other modules may be "stacked" into layers of software, with each responsible for the details of managing its own resources while hiding those details from client modules that use them.

Regardless of the type of peripheral or resource that a MPLAB Harmony driver manages, a driver has the following fundamental responsibilities:

- Provide a common system-level interface to the resource
- Provide a highly abstracted file system style client interface to the resource
- Manage the state of the peripheral or resource
- Manage access to the resource

A driver's system interface can be thought of as being a horizontal interface and its client interface can be thought of as being a vertical interface, as shown in the following block diagram.



The horizontal or "system" interface provides functions to initialize the driver and keep it running. To keep a driver running, a system loop or ISR function (but never both in the same system) calls its state machine "tasks" function repeatedly, as necessary. Therefore, a driver's system interface is normally only called by code that is generated by the MPLAB Harmony Configurator (MHC) when you select and configure the driver. Its purpose is to ensure that the driver works independently (conceptually in the background), providing the capabilities it implements. By contrast, the application (or any other "client" of the driver) normally only interacts with the driver's vertical "client" interface (often thought of as the driver's API). The client interface provides functions to open the driver for use and interact with it, reading or writing data or performing device-type specific operations. The client interface is what allows the application to access the peripheral in a safe and easy way without worrying about the details of the driver or what other clients it may be serving.

The following sections describe in general terms how to use these two interfaces and give specific examples to help illustrate the concepts. The subsequent help sections for each individual driver describe their specific interfaces in detail; listing all supported functions, parameters, and return values as well as their data types and expected behavior. You may also refer to the MPLAB Harmony Driver Development guide for additional information on MPLAB Harmony drivers and for information on how to develop your own drivers, if needed.

Using a Driver's System Interface

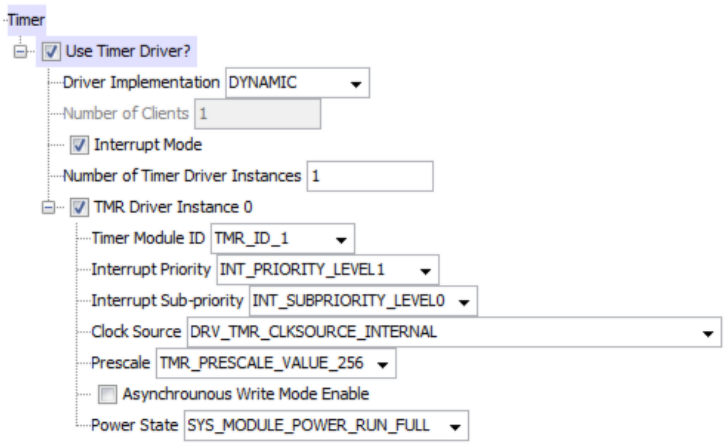
Introduces the System Interface of a MPLAB Harmony device driver and explains its usage.

Description

An MPLAB Harmony driver's system interface provides functions to initialize, deinitialize, and reinitialize an instance of a driver, as well as functions to maintain its state machine (and/or implement its Interrupt Service Routine) and check its current "running" status. Normally, as an MPLAB Harmony application developer or a developer of a "client" module that uses the driver, you will not call the system interface functions directly. The MHC generates calls to the system interface functions of any driver that is used in a project when it generates the system configuration files. Exactly which functions are called and exactly how they're called depends on the configuration options selected in the project's active configuration.

For example, when the box next to "Use Timer Driver?" is selected in the MHC Options tree (within *MPLAB Harmony & Application Configuration* > *Harmony Framework Configuration* > *Drivers* > *Timer*), as shown in the following figure, the MHC will generate all necessary definitions and function calls for the Timer Driver's system interface.

Example Timer Driver MHC Options



These configuration selections, which are set by default once "Use Timer Driver" is selected, will cause the MHC to generate the following definitions in the `system_config.h` header file for the main project's current configuration when **Generate Code** is clicked.

Example Driver Options in system_config.h

```
/*** Timer Driver Configuration ***/
#define DRV_TMR_INTERRUPT_MODE            true
#define DRV_TMR_INSTANCES_NUMBER          1
#define DRV_TMR_CLIENTS_NUMBER            1

/*** Timer Driver 0 Configuration ***/
#define DRV_TMR_PERIPHERAL_ID_IDX0         TMR_ID_1
#define DRV_TMR_INTERRUPT_SOURCE_IDX0      INT_SOURCE_TIMER_1
#define DRV_TMR_CLOCK_SOURCE_IDX0          DRV_TMR_CLKSOURCE_INTERNAL
#define DRV_TMR_PRESCALE_IDX0              TMR_PRESCALE_VALUE_256
#define DRV_TMR_OPERATION_MODE_IDX0        DRV_TMR_OPERATION_MODE_16_BIT
#define DRV_TMR_ASYNC_WRITE_ENABLE_IDX0    false
#define DRV_TMR_POWER_STATE_IDX0           SYS_MODULE_POWER_RUN_FULL
```

It is important to notice that the Driver Implementation selection in the MHC graphical interface does not correlate to a `#define` statement in the `system_config.h` file. Instead, it determines which implementation of the driver this configuration will use. Drivers may have more than one implementation. For example, most drivers have both static and dynamic implementations. A static implementation is usually the smaller of the two, but it is only capable of controlling one instance of a peripheral. An equivalent dynamic implementation will be larger, but it is capable of managing multiple instances of the same type of peripheral using a single instance of the source code (and thus, one instance of the object code). Some drivers may have additional implementations, each one optimized for a different usage. The Driver Implementation pull-down control in the MHC graphical interface allows you to select which implementation the current configuration will use. Normally, you can use only a single implementation of a driver in a given configuration. If you change driver implementations, it changes which implementation is used for all all instances of a peripheral.

The number of instances option, for example, Number of Timer Driver Instances, which correlates to the DRV_TMR_INSTANCES_NUMBER definition, determines how many instances of a static driver implementation will be generated or how many instances of a peripheral a dynamic driver implementation will manage. Drivers may also be designed to allow multiple different clients (applications or other modules) to share the same instance of a peripheral or resource. Therefore, a driver will have an option to determine a maximum number of simultaneous clients that it

can support. For example, Number of Clients (DRV_TMR_CLIENTS_NUMBER) in the Timer Driver, which is fixed at one (1) and cannot be changed, which indicates that the Timer Driver is a single-client driver). The last implementation-specific configuration option in this example is the "Interrupt Mode" (DRV_TMR_INTERRUPT_MODE) setting. This option determines if the implementation is configured to run polled or interrupt driven (discussed further, in a following section). MPLAB Harmony drivers are generally designed to run most effectively in an interrupt-driven configuration, but they can also be run in a polled configuration to simplify debugging or to support task prioritization in an RTOS configuration.

The remaining configuration options are all instance-specific initialization options. For a dynamic implementation of a driver, these options are passed into the driver's Initialize function through an "init" data structure, as shown in the following example.

Example Driver Init Structure in system_init.c

```
const DRV_TMR_INIT drvTmr0InitData =
{
    .moduleInit.sys.powerState = DRV_TMR_POWER_STATE_IDX0,
    .tmrId = DRV_TMR_PERIPHERAL_ID_IDX0,
    .clockSource = DRV_TMR_CLOCK_SOURCE_IDX0,
    .prescale = DRV_TMR_PRESCALE_IDX0,
    .mode = DRV_TMR_OPERATION_MODE_16_BIT,
    .interruptSource = DRV_TMR_INTERRUPT_SOURCE_IDX0,
    .asyncWriteEnable = false,
};
```

The exact meaning and usage of these options are described in the **Configuring the Library** section in the Help documentation for each library. The live MHC Help windowpane displays the associated help section whenever you select one of these options in the options tree.

There is one instance-specific initialization option of which you should take special notice: the peripheral ID option (.tmrId, in the Timer Driver example shown). This initialization option associates the driver instance (a zero-based index number) with the peripheral-hardware instance number, as defined by the data sheet for the processor in use. For a dynamic driver, this association is actually made when the driver's initialize function is called and passes a pointer to the init data structure, as shown in the following code example.

Example Driver Initialize Call in system_init.c

```
/* Initialize Drivers */
sysObj.drvTmr0 = DRV_TMR_Initialize(DRV_TMR_INDEX_0, (SYS_MODULE_INIT *)&drvTmr0InitData);
```

In this example, the driver index (DRV_TMR_INDEX_0) is defined as a numeric constant with a value of zero (0). This line of code associates driver instance 0 with hardware timer instance 1 by calling the DRV_TMR_Initialize function from the system initialization code and passing a pointer to the `drvTmr0InitData` structure. As shown earlier, the Timer Driver's init structure contains the value TMR_ID_1 (defined by the timer peripheral library), in its `.tmrId` data member.

In a static implementation, the driver peripheral ID macro (DRV_TMR_PERIPHERAL_ID_IDX0) defined in `system_config.h` is hard-coded into the driver's instance-specific initialization function when it is generated by the MHC, instead of defining an "init" structure, as shown in the following example; however, the effect is the same.

Example Static Driver Initialize Function

```
void DRV_TMR0_Initialize(void)
{
    PLIB_TMR_Stop(DRV_TMR_PERIPHERAL_ID_IDX0);
    PLIB_TMR_ClockSourceSelect(DRV_TMR_PERIPHERAL_ID_IDX0, DRV_TMR_CLOCK_SOURCE_IDX0);
    PLIB_TMR_PrescaleSelect(DRV_TMR_PERIPHERAL_ID_IDX0, DRV_TMR_PRESCALE_IDX0);
    PLIB_TMR_Mode16BitEnable(DRV_TMR_PERIPHERAL_ID_IDX0);
    PLIB_TMR_Counter16BitClear(DRV_TMR_PERIPHERAL_ID_IDX0);
    PLIB_TMR_Period16BitSet(DRV_TMR_PERIPHERAL_ID_IDX0, 0);
}
```

The DRV_TMR0_Initialize function (with an instance number '0' in the name) in the previous example, is a static version of the DRV_TMR_Initialize system interface function. The call to this function is created by the MHC when it generates the system code. Therefore, that call is always generated with the correct name and with the correct instance number in the name. However, when calling client interface functions (open, close, read, write, etc.) from your own applications, you *should not* use an instance number in the function name. Dynamic drivers implement the client interface functions without any index numbers in their names. Instead, they use an index or handle parameter to identify the instance of the driver with which to interact. Also, when using static implementations of the drivers, the dynamic API functions are mapped (using the index or handle parameter) to the appropriate static function with the index number in its name. Therefore, calling the dynamic API function makes your application always portable, using whichever driver instance is configured to the index value with which you open the driver.

Note: Calling the static versions of the interface function (with the index numbers in their names) is not prohibited. However, it will limit the portability of your application.

Understanding this mechanism is critical to understanding how to access the desired peripheral hardware instance. Therefore, it is worth looking at a few demonstration applications to see how it is used. Also, refer to *Volume IV: MPLAB Harmony Development > Key Concepts > Key One-to-Many Relationships* for additional information on the concepts of having multiple implementations, instances, and clients.

Something else worth noting about the previous example call to the Timer Driver's initialize functions is that when using a dynamic implementation, it returns a value called an "object handle". In the previous example, that object handle was stored in a system configuration object data member (`sysObj.drvTmr0`). Object handles returned by module initialization functions are stored in a system configuration structure normally named `sysObj`. The definition of this structure is generated in the `system_definitions.h` header file the MHC, as shown in the following example.

Example System Object Data Structure Definition in system_definitions.h

```
typedef struct
{
```

```
    SYS_MODULE_OBJ  sysDevcon;
    SYS_MODULE_OBJ  drvTmr0;

} SYSTEM_OBJECTS;

extern SYSTEM_OBJECTS sysObj;
```

As shown in the previous example, this structure is “extern’d” for use by the other system files. It should not be used by application or library files, only by the system files for a single configuration. The `sysObj` structure is defined (and allocated in memory) by the `system_init.c` file, as shown in the following example.

Example System sysObj Definition in system_init.c

```
/* Structure to hold the object handles for the modules in the system. */
SYSTEM_OBJECTS sysObj;
```

For this discussion, you can ignore the `sysDevcon` member of the SYSTEM_OBJECTS structure as it will contain the handle for a different library. The important thing to note is that the `drvTmr0` member must be passed into the Timer Driver’s other system interface functions so that the driver has access to the data it needs manage that specific instance of itself (and the associated peripheral hardware), as shown by the following timer ISR example.

Example Timer ISR in system_interrupt.c

```
void __ISR(_TIMER_1_VECTOR, ipl1AUTO) IntHandlerDrvTmrInstance0(void)
{
    DRV_TMR_Tasks(sysObj.drvTmr0);
}
```

In this ISR example, there are three important things to notice.

First, the ISR function itself is associated with a specific vector through the `__ISR` macro. Different interrupt vectors are associated with different peripheral instances and interrupts on different processors. That is why MPLAB Harmony ISR vector functions are generated in the configuration-specific `system_interrupt.c` file instead of being part of the driver library itself.

Second, the DRV_TMR_Tasks function implements the actual ISR logic of the TMR driver. Most MPLAB Harmony drivers are designed to run interrupt driven and their tasks functions implement the software state machine logic necessary to keep the driver’s interrupt sequence moving from one interrupt to the next until the driver’s task is complete.

Third, the `sysObj.drvTmr0` object handle’s value is passed into the driver’s tasks function so that it has access to the data it requires to control instance zero (0) of the Timer Driver and its associated hardware instance, which must match the ISR vector instance from which it is called.

By default, the Timer Driver is configured to run interrupt-driven, as shown previously. This is not necessarily true for all drivers. However, most drivers (including the Timer Driver) can run in a Polled mode by simply changing the configuration settings. For example, by clearing the "Interrupt Mode" option in the MHC configuration tree and regenerating the configuration code, the previous example ISR will be removed from `system_interrupt.c` and a call to the Timer Driver’s tasks function will be added to the polled system tasks function, as shown by the following `system_tasks.c` example code.

Example Call to Timer Tasks from system_tasks.c

```
void SYS_Tasks ( void )
{
    /* Maintain system services */
    SYS_DEVCON_Tasks(sysObj.sysDevcon);

    /* Maintain Device Drivers */
    DRV_TMR_Tasks(sysObj.drvTmr0);

    /* Maintain the application's state machine. */
    APP_Tasks();
}
```

In this example, the Timer Driver’s tasks function is called from the polled loop in main by the SYS_Tasks function. The driver’s tasks must still receive the `sysObj.drvTmr0` object handle value and its logic operates in exactly the same way, with one exception. Because the driver is now polled, the DRV_TMR_INTERRUPT_MODE option is now defined as false. This causes the driver to be built so that it does not enable its own interrupt, allowing it to run in the polled loop and to not require an ISR.

For additional information on the device driver system interface, refer to *Volume IV: MPLAB Harmony Development > MPLAB Harmony Driver Development Guide > System Interface* and to the documentation for the individual system interface functions for the driver in question.

Using a Driver's Client Interface

Introduces the Client Interface (or API) of a MPLAB Harmony device driver and explains common usage models.

Description

Applications (or any other “client” of a MPLAB Harmony device driver) normally only interact with the driver’s client interface (often called its API). The client interface provides functions to “open” the driver (creating a link between the client and the driver) and interact with it, to transfer data or perform operations that are specific to a given type of device, and to “close” the driver (releasing the link). Once a driver has been configured and the configuration code has been generated, the application can assume that the driver will be initialized by the system-wide initialization function (SYS_Initialize) and that its tasks functions will be called as required from either the system-wide tasks function (SYS_Tasks) or from the

appropriate ISR, depending upon how the driver was designed and configured.

To interact with the driver, a client must first call the driver's open function. This is necessary because all other client interface functions require a "handle" to the device driver that is returned by the open function, as shown in the following example.

Example Call to a Driver's Open Function

```
appData.handleTmr = DRV_TMR_Open(APP_TMR_DRV_INDEX, DRV_IO_INTENT_EXCLUSIVE);
if( DRV_HANDLE_INVALID != appData.handleTmr )
{
    // Advance to next application state.
}
```

In this example, the first parameter to the DRV_TMR_Open function is the APP_TMR_DRV_INDEX macro, which is a constant defined to the value of the desired driver instance index number in the `system_config.h` header file. This value must be the same as the index number used when the desired driver was initialized (as shown in the previous section). This is how the client becomes associated with a specific instance of a driver.

The second parameter identifies how the client intends to use the driver. Here, the client wants to have exclusive access to the driver. This means that no other client can currently have an active handle to this driver or this call will fail and return a value of DRV_HANDLE_INVALID. Drivers can also be opened as shared, as blocking or non-blocking and for reading, writing, or both. Refer to the help for the DRV_IO_INTENT data type for additional information about the IO intent parameter of driver open functions. This parameter is merely an advisory parameter. How it is used by the driver is implementation dependent and will be described in the driver's help documentation.

Finally, if the open function was successful, the returned value will be a valid handle to the driver instance. This value is opaque and meaningless to the caller, but it must be passed back to the driver as the first parameter to every other client interface function provided by the driver. A valid handle identifies both the instance of the driver with which the caller interacts and it identifies the client performing the call. This means that, two different client applications or modules opening the same driver in the same system at the same time will receive different values for their "opened" handle. If, for any reason, the driver cannot support the "open" request (it is not finished initializing itself, it has already been opened for exclusive access, or cannot accept new open requests for any reason), it will return a value of DRV_HANDLE_INVALID, indicating the client cannot use it at this time. The DRV_HANDLE_INVALID value is the only non-opaque value that a client should consider meaningful. All other values are only meaningful to the driver that provided them.

Note: The `appData.handleTmr` variable in the previous example is a member of the application's `appData` structure. This structure is generated by the MHC as part of the initial application template and should be used to hold an applications state variables.

When the client is finished using a driver, it may close it, as shown in the following example.

Example Call to a Driver's Close Function

```
DRV_TMR_Close(appData.handleTmr);
```

This action releases the link to the driver, invalidating the handle and releasing any resources allocated by the driver to track requests from the client. Notice that the close function demonstrates the use of the driver handle, requiring it as a parameter. However, after the close function returns, the handle value cannot be used again. Therefore, the client should not call the driver's close function until it is done using the driver or it will have to call open again and obtain a new handle to use the driver again. In fact, since many embedded applications are always running, they often do not bother to close drivers they use. But, applications that can go idle or that can be stopped and restarted or that need to share a driver with other clients, but want to conserve resources, or that want use the driver exclusively, can close a driver when they are finished with it for a time and reopen it later when needed. In fact, this is a good way to share a single-client driver, or a driver that supports exclusive access, allowing each client to open it and use it only when a valid handle is obtained.

Using a Driver in an Application

Describes how to write a state-machine based application that uses a MPLAB Harmony driver.

Description

MPLAB Harmony generally treats all software modules, including applications, as state machines that have an "initialize" function and a "tasks" function. In fact, when not using a RTOS, it essentially treats the entire system as one large state machine that runs in a common super loop in the "main" function, as shown in the following code example.

Example Main Function

```
int main ( void )
{
    SYS_Initialize(NULL);

    while(true)
    {
        SYS_Tasks();
    }

    return (EXIT_FAILURE);
}
```

For the purpose of this discussion, it is important to understand that the application's APP_Initialize function is called from the SYS_Initialize function, along with the initialization of functions of all drivers and other libraries before execution enters the endless `while(true)` super loop that continuously calls the system-wide SYS_Tasks function. The application's APP_Tasks function is then called from the SYS_Tasks function

inside of the super loop, along with all other polled modules in the system. If you are not already familiar with the organization of an MPLAB Harmony project, please refer to *Volume I: Getting Started With MPLAB Harmony > What is MPLAB Harmony?* for more information.

An application that uses a driver must define a DRV_HANDLE variable, as shown in the following example application header file.

Example Driver Application Header (app.h)

```
#include "driver/usart/drv_usart.h"

typedef enum
{
    APP_STATE_SETUP=0,
    APP_STATE_MESSAGE_SEND,
    APP_STATE_MESSAGE_WAIT,
    APP_STATE_DONE

} APP_STATES;


typedef struct
{
    APP_STATES  state;
    DRV_HANDLE  usart;
    char *      message;

} APP_DATA;
```

In this previous example, the driver handle variable is named `usart`. To keep the application well organized, it is common to keep all of the application's state variables (including one called "state" that holds the current state of the application's state machine) in a common structure (APP_DATA). This structure must be allocated in the application's source file (usually named `app.c`) and initialized by the application's initialization function, as shown in the following example.

Example Driver Application Initialization

```
APP_DATA appData;

void APP_Initialize ( void )
{
    /* Place the App in its initial state. */
    appData.state   = APP_STATE_SETUP;
    appData.usart   = DRV_HANDLE_INVALID;
    appData.message = "Hello World\n";
}
```

The APP_Initialze function must initialize the state variable (appData.state) to put the application's state machine in its initial state (the APP_STATE_SETUP value from the APP_STATES enumeration). It must also initialize the driver-handle variable (`appData.usart`), so that the state machine knows it is not yet valid, and any other application variables (like the string pointer, `appData.message`).

Once the application's data structure has been initialized, it is safe for the system (the main and SYS_Tasks functions) to call the application's APP_Tasks function from the super loop to keep it running. The APP_Tasks function then executes state transition code as it switches between states, as demonstrated by the following example.

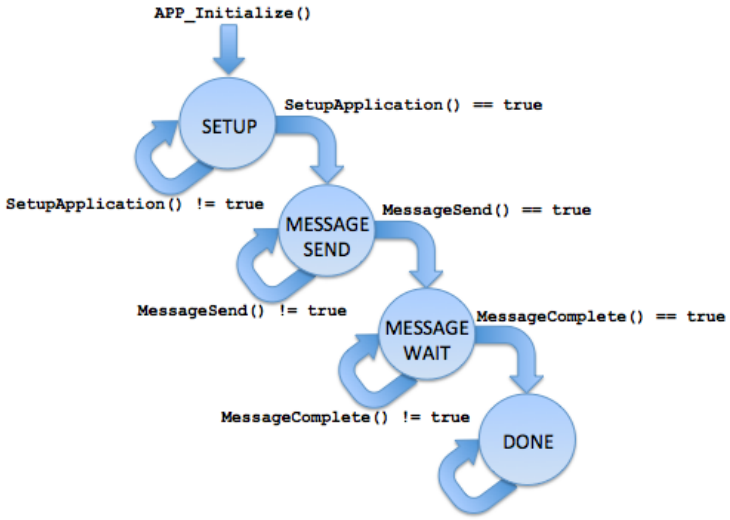
Example Application State Machine Using a Driver

```
void APP_Tasks ( void )
{
    switch ( appData.state )
    {
        case APP_STATE_SETUP:
        {
            if (SetupApplication() == true)
            {
                appData.state = APP_STATE_MESSAGE_SEND;
            }
            break;
        }

        case APP_STATE_MESSAGE_SEND:
        {
            if (MessageSend() == true)
            {
                appData.state = APP_STATE_MESSAGE_WAIT;
            }
            break;
        }

        case APP_STATE_MESSAGE_WAIT:
```

```
        {
            if (MessageComplete() == true)
            {
                appData.state = APP_STATE_DONE;
            }
            break;
        }

        case APP_STATE_DONE:
        default:
        {
            break;
        }
    }
}
```

There are numerous ways to implement a state machine. However, in this example, the application changes state when the APP_Tasks function assigns a new value from the APP_STATES enumeration to the `appData.states` variable. This happens when one of the state transition function returns true. The end result is an overall application state machine execution that retries each state transition until it succeeds before moving on to the next state, as shown in the following diagram.

Application State Machine



Note: The APP_STATE_ prefix and all inter-word underscores were removed from the state names to simplify the diagram.

After APP_Initialize places the state machine in its initial APP_STATE_SETUP state, the APP_Tasks function will call the SetupApplication function when it is called. When SetupApplication returns true indicating it has completed its task, the state machine advances to the next state. Otherwise, it stays in the same state and retries the tasks in the SetupApplication function. This pattern repeats for the APP_STATE_MESSAGE_SEND state and the `MessageSend` function as well as the APP_STATE_MESSAGE_WAIT state and the `MessageComplete` function. When all functions have returned true, the state machine to transitions to the APP_STATE_DONE state where it unconditionally stays having completed its tasks.

The sum total of the tasks performed by each transition function completes the overall task of the application. For an application that uses a driver like this example, this includes opening the driver, sending the message, and closing the driver when the message has been sent. How each individual transition function in this example application accomplishes its portion of the overall task, is described in the examples in the following sections to demonstrate how drivers are commonly used.

Opening a Driver

Describes how to open a driver in a state-machine based application.

Description

To use a MPLAB Harmony driver, an application (or other client) must call the driver's "open" function and obtain a valid handle to it, as shown by the following code example.

Example Opening a Driver

```
static bool SetupApplication ( void )
{
    if (appData.usart == DRV_HANDLE_INVALID)
```

```
    {
        appData.usart = DRV_USART_Open(APP_USART_DRIVER_INDEX,
                                       (DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_NONBLOCKING));
    }

    if (appData.usart == DRV_HANDLE_INVALID)
    {
        return false;
    }

    return true;
}
```

This example demonstrates the implementation of a state-transition function in a state machine-based application (as shown in the previous Using a Driver in an Application section). The `SetupApplication` function assumes that the `appData.usart` variable has been initialized to a value of DRV_HANDLE_INVALID when the application's state machine was initialized. Therefore, it checks this variable every time it is called to see if it has already completed its task. If `appData.usart` contains a value of DRV_HANDLE_INVALID, this indicates that the driver has not yet been successfully opened, causing the function to attempt to open the driver by calling DRV_USART_Open.

If the USART driver is ready and able to support a new client it will return a valid handle. If it is not ready or able to accept a new client, the driver will return DRV_HANDLE_INVALID and the `SetupApplication` function will return false and the application will stay in the same state and try to open the driver again the next time its state machine tasks function is called. When DRV_USART_Open returns a valid handle (a handle that is not equal to DRV_HANDLE_INVALID), the `SetupApplication` function returns true, allowing the application's state machine to advance.

This technique allows the application to try repeatedly to open the driver until it succeeds and guarantees that the application's state machine will not advance until it has done so. A more sophisticated application might use a time-out mechanism or some other error handling logic to take alternative action if it cannot open the driver in an acceptable time period. However, this simple implementation demonstrates the basic concept of how an MPLAB Harmony application (or any other client module) can safely open a driver before attempting to use it.

Using Driver Interface Functions

Describes how to use a device driver's synchronous client interface functions, such as those that read and write data.

Description

To use a MPLAB Harmony driver's client interface, the application must first obtain a valid handle from the driver's "open" function. The examples in this section assume that that has already occurred and that the value of the USART driver handle in the appData.usart variable is valid. The following example code demonstrates the implementation of a state transition function in a state machine-based application (as shown in the previous Using a Driver in an Application section) that writes data to a USART driver for transmission on the associated USART peripheral.

Example Writing Data To a Driver

```
static bool MessageSend ( void )
{
    size_t  count;
    size_t  length = strlen(appData.message);

    count = DRV_USART_Write(appData.usart, appData.message, length);

    appData.message += count;

    if (count == length)
    {
        return true;
    }

    return false;
}
```

In this example, the appData.message variable is a char pointer pointing to a null-terminated C-language string that was defined and initialized, as shown in the Using a Driver in an Application section. When `MessageSend` function is first called by the application's state machine, it points to the first character in the string to be transmitted. The function calculates the current length of the message string (using the standard C-language strlen function) and calls the driver's DRV_USART_Write function, passing it the valid driver handle (`appData.usart`) along with the pointer to the message string and its length, to transmit the message string on the associated USART.

If the driver is configured for blocking, the DRV_USART_Write function will not return until it has processed all of the data in the message string. However, that usually requires the use of a RTOS. Normally, in a bare-metal system (one that does not use a RTOS), MPLAB Harmony drivers are used in a non-blocking mode. In that case, a driver will perform as much of a task as it can when one of its interface functions is called without blocking. This means that the function will then return immediately, not waiting for the task to complete, and provide information on how much of the task was completed so the client can react appropriately. In this example, the DRV_USART_Write function will return a count of the number of bytes that were processed by the USART driver by this call to the function.

The `MessageSend` function captures the number of bytes processed by the DRV_USART_Write function in a local count variable. It then effectively removes those bytes from the message string by incrementing the pointer by count bytes (`appData.message` is a char pointer that increments by the size of one byte for every '1' added to it). Then, the `MessageSend` function checks to see if it was able to write the entire string

by comparing the value of count to the value of length that it calculated before calling the driver's write function. If the two are equal, the task is complete and the `MessageSend` function returns true and the application's state machine can continue to the next state. If the two values are not equal, this indicates there are remaining bytes in the message string. The `MessageSend` function returns false and the application must stay in the same state so that the function can attempt to send the remaining bytes next time it is called. A driver only accepts data when it can process it; therefore, the client can call its data transfer function as many times as necessary, even when the function returns bytes processed if it cannot accept more data at that time.

When a client has called a driver interface function there are really only two possibilities. Either the operation has completed when the function returns, or the operation continues after the function has returned. If the operation completes immediately, the client can continue on without taking further action. However, in this example, while the USART driver may have accepted some of the bytes in the message string (perhaps copying them to an internal hardware or software FIFO buffer), it still takes some time to transmit the data over the USART peripheral. In many cases the client may need to know when the operation has actually completed. For this reason, most drivers provide one or more status functions that client applications may call to determine the current status of an operation, as demonstrated in the following example.

Example Using a Driver Status Function

```
static bool MessageComplete ( void )
{
    if (DRV_USART_ClientStatus(appData.usart) == DRV_USART_CLIENT_STATUS_BUSY)
    {
        return false;
    }
    return true;
}
```

This example extends the previous one and assumes that the `MessageSend` function has returned true and the application has moved to a new state where it calls this function to determine when the driver is idle, which indicates that the message has been completely transmitted. To do that, the `MessageComplete` function calls the DRV_USART_ClientStatus function. If its return value is DRV_USART_CLIENT_STATUS_BUSY, the USART driver is still working on a previous request by the client. If any other status value is returned, this indicates that the driver is no longer busy with a current request and the `MessageComplete` function returns true so that the client application's state machine can move on. A more sophisticated example would check for other possible status values that might indicate some error has occurred and take appropriate action. However, this example is sufficient to demonstrate the concept of checking a driver status function to determine when it is safe to move to another state.

Since the client application stays in the same state calling the status function each time its tasks function is called until the desired status is returned, it is effectively polling the status as if it were in a `while` loop. In fact, it is in the system-wide while loop. However, by not trapping the CPU within its own internal `while` loop, the application allows other modules (including, potentially, the driver it is using) to continue running and servicing requests. Failing to allow the rest of the system to run can result in a deadlock where the polling application is waiting for a status; however, the driver it is polling will never be able to provide the expected status, as the driver's own tasks function is not allowed to run. This is why it is important to use the technique described here to "poll" status from modules outside of the current module.

Using Asynchronous and Callback Functions

Describes how to use an asynchronous interface function to start a driver operation and receive a callback when the operation is complete.

Description

When a client calls a function that is part of an asynchronous interface, the function starts the request and returns immediately, without finishing the request. The client can then either poll a status function to determine when the request has finished (as demonstrated in the Using Driver Interface Functions section) or it can utilize a callback function to receive a notification from the driver when the request has finished. So, the difference between an asynchronous interface and a synchronous interface is that a synchronous interface may finish all or part of the request before returning, whereas an asynchronous interface will always return immediately having only started the request. Determination of when the request has completed is handled separately.

The examples in this section reimplement some of the code from the example application described in the previous sections to demonstrate how to use asynchronous queuing and callback interfaces instead of the synchronous status-polling interface demonstrated in the Using Driver Interface Functions section. To use an asynchronous interface, we will first add a couple of new variables to our example application's data structure, as shown by the following structure definition.

Example Driver Application Header (app.h)

```
typedef struct
{
    APP_STATES              state;
    DRV_HANDLE              usart;
    char *                  message;
    DRV_USART_BUFFER_HANDLE messageHandle;
    bool                    messageDone;

} APP_DATA;
```

The `state`, `usart`, and `message` members of the APP_DATA structure are used in exactly the same way as they were in the previous examples. The `messageHandle` variable will be explained later and the `messageDone` variable is a Boolean flag used by the callback function to indicate to the application's state machine that the message has been completely processed by the driver. Using these new mechanisms results in very minor changes to the application's state machine, as shown in the following example `APP_Initialize` and `APP_Tasks` implementations.

Example Driver Application State Machine (app.c)

```
void APP_Initialize ( void )
{
    appData.state           = APP_STATE_SETUP;
    appData.usart           = DRV_HANDLE_INVALID;
    appData.message         = APP_MESSAGE;
    appData.messageHandle   = DRV_USART_BUFFER_HANDLE_INVALID;
}

void APP_Tasks ( void )
{
    switch ( appData.state )
    {
        case APP_STATE_SETUP:
        {
            if (SetupApplication() == true)
            {
                appData.state = APP_STATE_MESSAGE_SEND;
            }
            break;
        }

        case APP_STATE_MESSAGE_SEND:
        {
            if (MessageSend() == true)
            {
                appData.state = APP_STATE_MESSAGE_WAIT;
            }
            break;
        }

        case APP_STATE_MESSAGE_WAIT:
        {
            if (appData.messageDone)
            {
                DRV_USART_Close(appData.usart);
                appData.state = APP_STATE_DONE;
            }
            break;
        }

        case APP_STATE_DONE:
        default:
        {
            break;
        }
    }
}
```

As described previously, the `SetupApplication` state transition function opens the USART driver and the `MessageSend` function sends the message to it. However, there is no need for a `MessageComplete` state transition function. Instead, the application must implement a callback function that will set the `appData.messageDone` Boolean flag when the driver calls the application "back" to indicate that the message has been sent.

Note: The `AppInitialize` function initializes the `state`, `usart`, and `message` members of the `appData` structure as previously described. And, it also initializes the `messageHandle` member with an invalid value to indicate that the message has not yet been sent. However, it does not initialize the `messageDone` flag because it is more appropriate to clear the flag elsewhere, immediately before calling the driver to send the message.

To use a callback mechanism requires the client to implement and register a callback function. A client must register this function after opening the driver, but prior to calling the driver to initiate the operation. This is often done in the same state transition that opens the driver, as shown in the following `SetupApplication` example.

Example Registering a Driver Callback Function

```
static void BufferDone ( DRV_USART_BUFFER_EVENT  event,
                         DRV_USART_BUFFER_HANDLE bufferHandle,
                         uintptr_t context )
{
    APP_DATA *pAppData = (APP_DATA *)context;
```

```
    if (event == DRV_USART_BUFFER_EVENT_COMPLETE)
    {
        if (bufferHandle == pAppData->messageHandle)
        {
            pAppData->messageDone = true;
            return;
        }
    }

    /* Error */
    return;
}


static bool SetupApplication ( void )
{
    if (appData.usart == DRV_HANDLE_INVALID)
    {
        appData.usart = DRV_USART_Open(APP_USART_DRIVER_INDEX,
                                       (DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_NONBLOCKING));
    }

    if (appData.usart == DRV_HANDLE_INVALID)
    {
        return false;
    }

    DRV_USART_BufferEventHandlerSet(appData.usart, BufferDone, (uintptr_t)&appData);
    return true;
}
```

This code block implements both the `BufferDone` callback function and the application's `SetupApplication` state transition function. After successfully opening the driver, the `SetupApplication` function calls the DRV_USART_BufferEventHandlerSet function and passes it the driver handle (`appData.usart`) once it is valid, along with the address of the `BufferDone` callback function and a context value.

The context value can be anything that will fit in an integer large enough to hold a pointer (it is a `uintptr_t` variable). However, this parameter is most commonly used to pass a pointer to the caller's own data structure as demonstrated here (even though it is not strictly necessary). This is done primarily to support multi-instance clients. (Refer to *Volume IV: MPLAB Harmony Development > Key Concepts* for information on multiple instances.) A multi-instance client is designed to manage multiple instances of itself by allocating multiple instances of its own data structure, but only one instance of its object code. Passing a pointer to the data structure in the context variable identifies the specific instance that was used when calling the driver.

Once the callback function has been registered with the driver, the application can transition to a state where it attempts to initiate an asynchronous operation. The following example demonstrates the use of a buffer-queuing write function to transmit a message over the USART.

Example Queuing a Buffer to a Driver

```
static bool MessageSend ( void )
{
    appData.messageDone = false;
    DRV_USART_BufferAddWrite(appData.usart, &appData.messageHandle,
                             appData.message, strlen(appData.message));

    if (appData.messageHandle == DRV_USART_BUFFER_HANDLE_INVALID)
    {
        return false;
    }

    return true;
}
```

Before attempting to send the message, this implementation of the `MessageSend` state transition function clears the `appData.messageDone` flag so it can detect when the message has completed. Then, it calls the DRV_USART_BufferAddWrite function to queue up the buffer containing the message to be transmitted by the USART driver. To that function, it passes the USART driver handle (`appData.usart`), the address of the `appData.messageHandle` variable, the pointer to the message buffer (`appData.message`), and the size of the buffer in bytes as calculated by the `strlen` function. The USART driver then adds this buffer to its internal queue of buffers to transmit and provides a handle to the caller that identifies that buffer's place in the queue by storing it to the `appData.messageHandle` variable.

If, for some reason, the driver is unable to successfully queue up the buffer (perhaps the queue is full), it will assign a value of DRV_USART_BUFFER_HANDLE_INVALID to the `appData.messageHandle` variable. If that happens, the `MessageSend` function returns false and the application will stay in the same state and retry the operation again next time its tasks function is called. But, if the operation succeeds, the application advances to the next state.

Once the driver completes the operation, it will call the client's callback function. As shown in the `BufferDone` code example, the driver passes it an enumeration value that identifies which event has just occurred (the `DRV_USART_BUFFER_EVENT_COMPLETE` value) in the event parameter. It

also passes it the handle of the buffer that has just completed (`bufferHandle`). The client can use the `bufferHandle` value to verify that it matches the value stored in the `appData.bufferHandle` variable to uniquely identify an individual buffer. This is very useful when a client queues up multiple buffers at the same, which is being shown in this example as a demonstration.

The context parameter to the `BufferDone` function contains a pointer to the application's global (`appData`) data structure. (This is the same value that was passed in the context parameter to the DRV_USART_BufferEventHandlerSet function.) While not strictly necessary in this example, it is very useful for multi-instance clients such as dynamic device drivers and middleware to identify which instance of the client requested the operation. The callback function simply casts the context value back into a pointer to the client's own data structure's data type (`APP_DATA` in this example) and uses it to access the structure members. (Again, please refer to *Volume IV: MPLAB Harmony Development > Key Concepts* for information on multiple instances.)

The callback function uses the `event` parameter to identify why the callback occurred. If it was called to indicate that the buffer has been processed, the `event` parameter will contain the value DRV_USART_BUFFER_EVENT_COMPLETE. If it contains any other value an error has occurred. The `BufferDone` callback also checks to verify that the buffer that completed was the same buffer that it queued up by comparing the `bufferHandle` value it was passed with the value assigned to the `appData.messageHandle` variable when the application called DRV_USART_BufferAddWrite. It accesses the message handle value it saved using the `pAppData` pointer given to it through the context parameter just. Once it has verified that the buffer it queued has completed, it sets the `pAppData->messageDone` flag to notify the application's state machine and execution returns to the driver.

Note: It is important to understand that the `MessageDone` callback function executes in the context of the driver, not the application. Depending on how the system is configured, this means that it may be called from within the driver's ISR context or from another thread context if using a RTOS.

In this example, the `APP_Tasks` application state machine function is essentially the same as the state machine for the synchronous example. The only difference is that when the application is in the `APP_STATE_MESSAGE_WAIT` state, it checks the `appData.messageDone` flag to determine when to close the driver and transition to the `APP_STATE_DONE` state instead of calling a transition function. (It could still do this in a state transition function, but it was done differently in this example to emphasize the concept.)

The advantage of using an asynchronous interface over a synchronous one is that it allows the client's state machine to continue on, potentially doing something else while the requested operation completes. Whereas a synchronous interface has the possibility of blocking the client's state machine until the operation finishes (when used in a RTOS configuration). An asynchronous interface will always return immediately without blocking (whether a RTOS is used or not). Because of this, most asynchronous interfaces will also allow queuing of more than one operation at a time. This allows client applications to keep a driver continuously busy by keeping the driver's queue full, maximizing data throughput or operation speed. By contrast, a synchronous interface requires one operation to complete before the synchronous function can be called again to cause the next one to begin.

The cost of this capability is that an asynchronous interface has the added complexity of a callback function (if the client cares when the operation finishes) and the fact that a callback function may be called from within the driver's ISR context, depending on how the driver was designed and configured. This fact generally restricts what can be done within the callback function. For example, it is usually a bad idea to perform lengthy processing within a callback function as it will block all lower priority ISRs (as well as the main loop or other threads) while that processing occurs. Also, it is usually best to not call back into the driver's own interface functions unless those functions are documented as being safe to call from within the driver's callback context. Many interface functions (particularly data transfer and data queuing functions) must use semaphores or mutexes to protect their internal data structures in RTOS environments and those constructs cannot be used from within an ISR.

It is also important to not make non-atomic (read-modify-write) accesses to the client's own state data from within the callback function, as the client cannot protect itself against an interrupt that is owned by the driver. That is why a separate Boolean flag variable is commonly used to indicate to the client that the callback has occurred. Most other processing should occur in the client's state machine. It is usually best to simply capture the event and return as quickly as possible from the callback function and let the application's state machine tasks function perform any lengthy processing or calling back into the driver.

Please refer to *Volume IV: MPLAB Harmony Development* for additional information.

Library Interface

Constants

	Name	Description
	DRV_CONFIG_NOT_SUPPORTED	Not supported configuration.
	DRV_HANDLE_INVALID	Invalid device handle.
	DRV_IO_ISBLOCKING	Returns if the I/O intent provided is blocking
	DRV_IO_ISEXCLUSIVE	Returns if the I/O intent provided is non-blocking.
	DRV_IO_ISNONBLOCKING	Returns if the I/O intent provided is non-blocking.
	_DRV_COMMON_H	This is macro _DRV_COMMON_H.
	_PLIB_UNSUPPORTED	Abstracts the use of the unsupported attribute defined by the compiler.

Data Types

	Name	Description
	DRV_CLIENT_STATUS	Identifies the current status/state of a client's connection to a driver.
	DRV_HANDLE	Handle to an opened device driver.

	DRV_IO_BUFFER_TYPES	Identifies to which buffer a device operation will apply.
	DRV_IO_INTENT	Identifies the intended usage of the device when it is opened.

Description

Data Types

DRV_CLIENT_STATUS Enumeration

Identifies the current status/state of a client's connection to a driver.

File

driver_common.h

C

```
typedef enum {
  DRV_CLIENT_STATUS_ERROR_EXTENDED = -10,
  DRV_CLIENT_STATUS_ERROR = -1,
  DRV_CLIENT_STATUS_CLOSED = 0,
  DRV_CLIENT_STATUS_BUSY = 1,
  DRV_CLIENT_STATUS_READY = 2,
  DRV_CLIENT_STATUS_READY_EXTENDED = 10
} DRV_CLIENT_STATUS;
```

Members

Members	Description
DRV_CLIENT_STATUS_ERROR_EXTENDED = -10	Indicates that a driver-specific error has occurred.
DRV_CLIENT_STATUS_ERROR = -1	An unspecified error has occurred.
DRV_CLIENT_STATUS_CLOSED = 0	The driver is closed, no operations for this client are ongoing, and/or the given handle is invalid.
DRV_CLIENT_STATUS_BUSY = 1	The driver is currently busy and cannot start additional operations.
DRV_CLIENT_STATUS_READY = 2	The module is running and ready for additional operations
DRV_CLIENT_STATUS_READY_EXTENDED = 10	Indicates that the module is in a driver-specific ready/run state.

Description

Driver Client Status

This enumeration identifies the current status/state of a client's link to a driver.

Remarks

The enumeration used as the return type for the client-level status routines defined by each device driver or system module (for example, DRV_USART_ClientStatus) must be based on the values in this enumeration.

DRV_HANDLE Type

Handle to an opened device driver.

File

driver_common.h

C

```
typedef uintptr_t DRV_HANDLE;
```

Description

Device Handle

This handle identifies the open instance of a device driver. It must be passed to all other driver routines (except the initialization, deinitialization, or power routines) to identify the caller.

Remarks

Every application or module that wants to use a driver must first call the driver's open routine. This is the only routine that is absolutely required for every driver.

If a driver is unable to allow an additional module to use it, it must then return the special value DRV_HANDLE_INVALID. Callers should check the handle returned for this value to ensure this value was not returned before attempting to call any other driver routines using the handle.

DRV_IO_BUFFER_TYPES Enumeration

Identifies to which buffer a device operation will apply.

File

driver_common.h

C

```
typedef enum {
  DRV_IO_BUFFER_TYPE_NONE = 0x00,
  DRV_IO_BUFFER_TYPE_READ = 0x01,
  DRV_IO_BUFFER_TYPE_WRITE = 0x02,
  DRV_IO_BUFFER_TYPE_RW = DRV_IO_BUFFER_TYPE_READ|DRV_IO_BUFFER_TYPE_WRITE
} DRV_IO_BUFFER_TYPES;
```

Members

Members	Description
DRV_IO_BUFFER_TYPE_NONE = 0x00	Operation does not apply to any buffer
DRV_IO_BUFFER_TYPE_READ = 0x01	Operation applies to read buffer
DRV_IO_BUFFER_TYPE_WRITE = 0x02	Operation applies to write buffer
DRV_IO_BUFFER_TYPE_RW = DRV_IO_BUFFER_TYPE_READ|DRV_IO_BUFFER_TYPE_WRITE	Operation applies to both read and write buffers

Description

Device Driver IO Buffer Identifier

This enumeration identifies to which buffer (read, write, both, or neither) a device operation will apply. This is used for "flush" (or similar) operations.

DRV_IO_INTENT Enumeration

Identifies the intended usage of the device when it is opened.

File

driver_common.h

C

```
typedef enum {
  DRV_IO_INTENT_READ,
  DRV_IO_INTENT_WRITE,
  DRV_IO_INTENT_READWRITE,
  DRV_IO_INTENT_BLOCKING,
  DRV_IO_INTENT_NONBLOCKING,
  DRV_IO_INTENT_EXCLUSIVE,
  DRV_IO_INTENT_SHARED
} DRV_IO_INTENT;
```

Members

Members	Description
DRV_IO_INTENT_READ	Read
DRV_IO_INTENT_WRITE	Write
DRV_IO_INTENT_READWRITE	Read and Write
DRV_IO_INTENT_BLOCKING	The driver will block and will return when the operation is complete
DRV_IO_INTENT_NONBLOCKING	The driver will return immediately
DRV_IO_INTENT_EXCLUSIVE	The driver will support only one client at a time
DRV_IO_INTENT_SHARED	The driver will support multiple clients at a time

Description

Device Driver I/O Intent

This enumeration identifies the intended usage of the device when the caller opens the device. It identifies the desired behavior of the device driver for the following:

- Blocking or non-blocking I/O behavior (do I/O calls such as read and write block until the operation is finished or do they return immediately and require the caller to call another routine to check the status of the operation)
- Support reading and/or writing of data from/to the device
- Identify the buffering behavior (sometimes called "double buffering" of the driver. Indicates if the driver should maintain its own read/write buffers and copy data to/from these buffers to/from the caller's buffers.
- Identify the DMA behavior of the peripheral

Remarks

The buffer allocation method is not identified by this enumeration. Buffers can be allocated statically at build time, dynamically at run-time, or even allocated by the caller and passed to the driver for its own usage if a driver-specific routine is provided for such. This choice is left to the design of the individual driver and is considered part of its interface.

These values can be considered "flags". One selection from each of the groups below can be ORed together to create the complete value passed to the driver's open routine.

Constants

DRV_CONFIG_NOT_SUPPORTED Macro

Not supported configuration.

File

driver_common.h

C

```
#define DRV_CONFIG_NOT_SUPPORTED (((unsigned short) -1))
```

Description

Not supported configuration

If the configuration option is not supported on an instance of the peripheral, use this macro to equate to that configuration. This option should be listed as a possible value in the description of that configuration option.

DRV_HANDLE_INVALID Macro

Invalid device handle.

File

driver_common.h

C

```
#define DRV_HANDLE_INVALID (((DRV_HANDLE) -1))
```

Description

Invalid Device Handle

If a driver is unable to allow an additional module to use it, it must then return the special value DRV_HANDLE_INVALID. Callers should check the handle returned for this value to ensure this value was not returned before attempting to call any other driver routines using the handle.

Remarks

None.

DRV_IO_ISBLOCKING Macro

Returns if the I/O intent provided is blocking

File

driver_common.h

C

```
#define DRV_IO_ISBLOCKING(intent) (intent & DRV_IO_INTENT_BLOCKING)
```

Description

Device Driver Blocking Status Macro

This macro returns if the I/O intent provided is blocking.

Remarks

None.

DRV_IO_ISEXCLUSIVE Macro

Returns if the I/O intent provided is non-blocking.

File

driver_common.h

C

```
#define DRV_IO_ISEXCLUSIVE(intent) (intent & DRV_IO_INTENT_EXCLUSIVE)
```

Description

Device Driver Exclusive Status Macro

This macro returns if the I/O intent provided is non-blocking.

Remarks

None.

DRV_IO_ISNONBLOCKING Macro

Returns if the I/O intent provided is non-blocking.

File

driver_common.h

C

```
#define DRV_IO_ISNONBLOCKING(intent) (intent & DRV_IO_INTENT_NONBLOCKING )
```

Description

Device Driver Non Blocking Status Macro

This macro returns if the I/ intent provided is non-blocking.

Remarks

None.

_DRV_COMMON_H Macro

File

driver_common.h

C

```
#define _DRV_COMMON_H
```

Description

This is macro _DRV_COMMON_H.

_PLIB_UNSUPPORTED Macro

Abstracts the use of the unsupported attribute defined by the compiler.

File

driver_common.h

C

```
#define _PLIB_UNSUPPORTED
```

Description

Unsupported Attribute Abstraction

This macro nulls the definition of the _PLIB_UNSUPPORTED macro, to support compilation of the drivers for all different variants.

Remarks

None.

Example

void _PLIB_UNSUPPORTED PLIB_USART_Enable(USART_MODULE_ID index);

This function will not generate a compiler error if the interface is not defined for the selected device.

Files

Files

Name	Description
driver.h	This file aggregates all of the driver library interface headers.
driver_common.h	This file defines the common macros and definitions used by the driver definition and implementation headers.

Description

driver.h

This file aggregates all of the driver library interface headers.

Description

Driver Library Interface Header Definitions

Driver Library Interface Header This file aggregates all of the driver library interface headers so client code only needs to include this one single header to obtain prototypes and definitions for the interfaces to all driver libraries. A device driver provides a simple well-defined interface to a hardware peripheral that can be used without operating system support or that can be easily ported to a variety of operating systems. A driver has the fundamental responsibilities:

- Providing a highly abstracted interface to a peripheral
- Controlling access to a peripheral
- Managing the state of a peripheral

Remarks

The directory in which this file resides should be added to the compiler's search path for header files.

File Name

drv.h

Company

Microchip Technology Inc.

driver_common.h

This file defines the common macros and definitions used by the driver definition and implementation headers.

Enumerations

	Name	Description
	DRV_CLIENT_STATUS	Identifies the current status/state of a client's connection to a driver.
	DRV_IO_BUFFER_TYPES	Identifies to which buffer a device operation will apply.
	DRV_IO_INTENT	Identifies the intended usage of the device when it is opened.

Macros

	Name	Description
	_DRV_COMMON_H	This is macro _DRV_COMMON_H.
	_PLIB_UNSUPPORTED	Abstracts the use of the unsupported attribute defined by the compiler.
	DRV_CONFIG_NOT_SUPPORTED	Not supported configuration.
	DRV_HANDLE_INVALID	Invalid device handle.
	DRV_IO_ISBLOCKING	Returns if the I/O intent provided is blocking
	DRV_IO_ISEXCLUSIVE	Returns if the I/O intent provided is non-blocking.
	DRV_IO_ISNONBLOCKING	Returns if the I/O intent provided is non-blocking.

Types

	Name	Description
	DRV_HANDLE	Handle to an opened device driver.

Description

Driver Common Header Definitions

This file defines the common macros and definitions used by the driver definition and the implementation header.

Remarks

The directory in which this file resides should be added to the compiler's search path for header files.

File Name

drv_common.h

Company

Microchip Technology Inc.

ADC Driver Library

This section describes the Analog-to-Digital Converter (ADC) Driver Library.

Introduction

This Analog-to-Digital Converter (ADC) driver provides an interface to manage the ADC module on the Microchip family of microcontrollers.

Description

An ADC is a vital part of any system that interfaces to real-world signals. While there are many techniques for analog-to-digital conversion, the Microchip family of microcontrollers uses Successive Approximation as one of its primary techniques.

Through MHC, this driver provides APIs to interact with the ADC module.

Note: Only Static implementation is supported for the ADC Driver Library.

Library Interface

Functions

	Name	Description
	DRV_ADC_Deinitialize	Deinitializes the DRV_ADC_Initialize driver module **Implementation:** Static
	DRV_ADC_Initialize	Initializes the ADC driver. **Implementation:** Static
	DRV_ADC_SamplesAvailable	Identifies if specified ADC Driver input has any samples available to read. **Implementation:** Static
	DRV_ADC_SamplesRead	Reads the converted sample data from ADC input Data buffer. **Implementation:** Static
	DRV_ADC_Start	Starts the software trigger for the ADC driver sampling and converting analog to digital values. **Implementation:** Static
	DRV_ADC_Stop	Stops the Global Software Level Trigger from continuing triggering for converting ADC data. **Implementation:** Static
	DRV_ADCx_Close	Closes the ADC instance for the specified driver index. **Implementation:** Static
	DRV_ADCx_Open	Opens the ADC instance for the specified driver index. **Implementation:** Static

Description

This section lists the interface routines, data types, constants and macros for the library.

Functions

DRV_ADC_Deinitialize Function

Deinitializes the DRV_ADC_Initialize driver module

Implementation: Static

File

help_drv_adc.h

C

```
void DRV_ADC_Deinitialize();
```

Returns

None.

Description

This function deinitializes the ADC Driver module for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Remarks

None.

Preconditions

None.

Function

void DRV_ADC_Deinitialize(void)

DRV_ADC_Initialize Function

Initializes the ADC driver.

Implementation: Static

File

help_drv_adc.h

C

```
void DRV_ADC_Initialize();
```

Returns

None.

Description

This function initializes the ADC Driver module for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Remarks

This function must be called before any other ADC function is called. This function should only be called once during system initialization.

Preconditions

None.

Function

void DRV_ADC_Initialize(void)

DRV_ADC_SamplesAvailable Function

Identifies if specified ADC Driver input has any samples available to read.

Implementation: Static

File

help_drv_adc.h

C

```
bool DRV_ADC_SamplesAvailable(uint8_t bufIndex);
```

Returns

- true - When ADC data buffer is available to be read
- false - When ADC data buffer is not available

Description

This function identifies whether the specified ADC Driver input has any samples available to read.

Remarks

None.

Preconditions

The following functions have been called:

- DRV_ADC_Initialize
- DRV_ADCx_Open
- DRV_ADC_Start or other triggered by source setup in MHC

Parameters

Parameters	Description
uint8_t bufIndex	ADC input number (ANx)

Function

bool DRV_ADC_SamplesAvailable(uint8_t bufIndex);

DRV_ADC_SamplesRead Function

Reads the converted sample data from ADC input Data buffer.

Implementation: Static

File

help_drv_adc.h

C

```
uint32_t DRV_ADC_SamplesRead(uint8_t bufIndex);
```

Returns

uint32_t - ADC converted sample data.

Description

This function returns the converted sample data from ADC input Data buffer.

Remarks

None.

Preconditions

The following functions have been called:

- DRV_ADC_Initialize
- DRV_ADCx_Open
- DRV_ADC_Start or other triggered by source setup in MHC

Parameters

Parameters	Description
uint8_t bufIndex	Analog input number (ANx)

Function

uint32_t DRV_ADC_SamplesRead(uint8_t bufIndex);

DRV_ADC_Start Function

Starts the software trigger for the ADC driver sampling and converting analog to digital values.

Implementation: Static

File

help_drv_adc.h

C

```
void DRV_ADC_Start();
```

Returns

None.

Description

This function provides a global edge and level trigger for the ADC driver to start the conversion.

Remarks

None.

Preconditions

The following functions have been called:

- DRV_ADC_Initialize
- DRV_ADCx_Open

Function

void DRV_ADC_Start(void);

DRV_ADC_Stop Function

Stops the Global Software Level Trigger from continuing triggering for converting ADC data.

Implementation: Static

File

help_drv_adc.h

C

```
void DRV_ADC_Stop();
```

Returns

None.

Description

This function stops the Global Software Level Trigger from continuing triggering for converting ADC data.

Remarks

None.

Preconditions

The following functions have been called:

- DRV_ADC_Initialize
- DRV_ADCx_Open

Function

void DRV_ADC_Stop(void);

DRV_ADCx_Close Function

Closes the ADC instance for the specified driver index.

Implementation: Static

File

help_drv_adc.h

C

```
void DRV_ADCx_Close();
```

Returns

None.

Description

This function closes the specified driver instance (where 'x' is the instance number) making it ready for clients to use it.

Remarks

'x' indicates the instance number.

Preconditions

DRV_ADC_Initialize has been called.

Function

void DRV_ADCx_Close(void)

DRV_ADCx_Open Function

Opens the ADC instance for the specified driver index.

Implementation: Static

File

help_drv_adc.h

C

```
void DRV_ADCx_Open();
```

Returns

None.

Description

This function opens the specified driver instance (where 'x' is the instance number) making it ready for clients to use it.

Remarks

'x' indicates the instance number.

Preconditions

DRV_ADC_Initialize has been called.

Function

void DRV_ADCx_Open(void)

Bluetooth Driver Libraries

This section describes the Bluetooth Driver Libraries that are included in your installation of MPLAB Harmony.

BM64 Bluetooth Driver Library

This section describes the BM64 Bluetooth Driver Library.

Introduction

This library provides an Applications Programming Interface (API) to manage a BM64 Module that is connected to a Microchip PIC32 microcontroller using UART and I²S for providing Bluetooth solutions for audio and Bluetooth Low Energy (BLE) applications.

Description

The BM64 is a Bluetooth 4.2 Stereo Module that supports classic A2DP, AVRCP, HFP, HSP, and SPP protocols, as well as Bluetooth Low Energy (BLE).

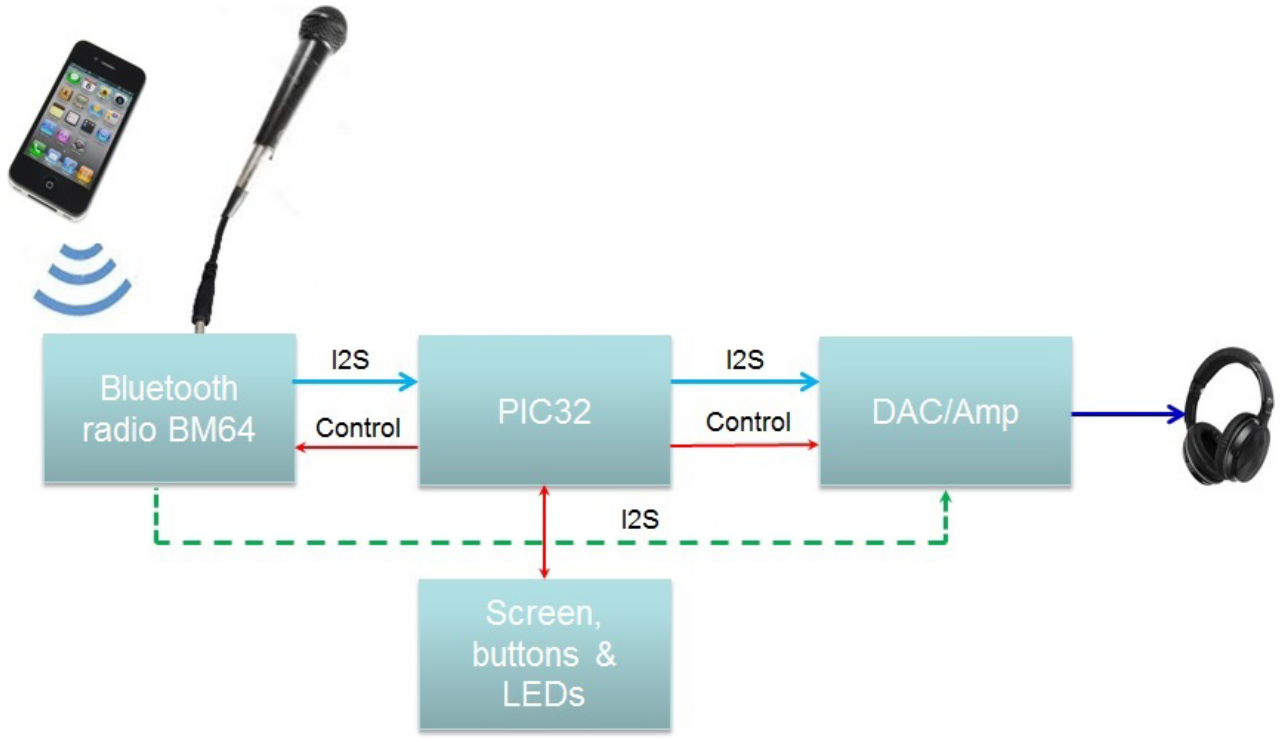
The BM64 streams I2S audio at up to 24-bit and 96 kHz, and uses a UART to receive commands from the host microcontroller (PIC32) and send events back over the same interface.

Protocols supported by the BM64 include A2DP, AVRCP, HFP, HSP, SPP, and BLE. However, this version of the driver only supports A2DP, AVRCP, HFP, and BLE.

The BM64 can be connected to a microphone (for HFP) and also has line-input; however, the latter is not supported by this driver. The multi-speaker modes of the BM64 are also not handled by this driver.

A typical interface of BM64 to a Microchip PIC32 device is provided in the following diagram:

BM64 to PIC32 Device Interface

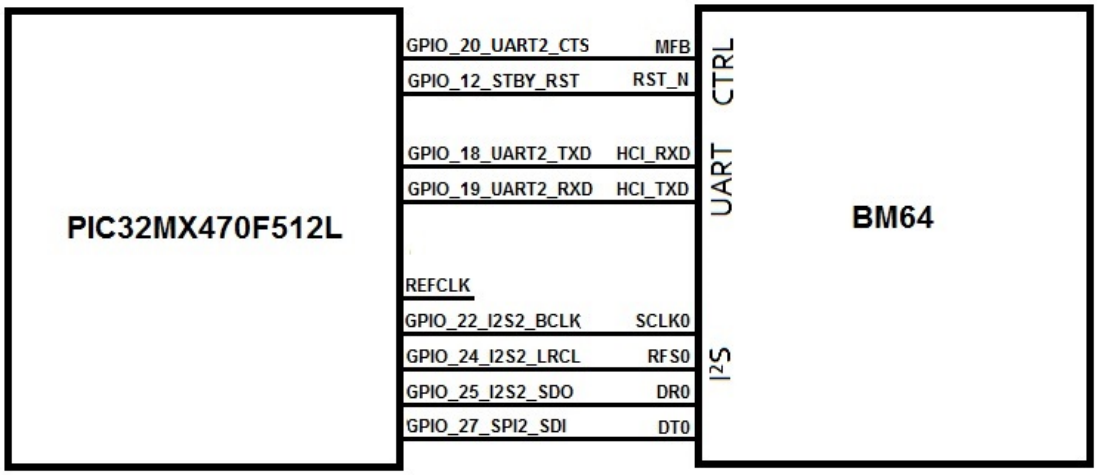


An example demonstration application using this library to interface with the BM64 for audio is BM64_a2dp_hfp, which runs on the PIC32 Bluetooth Audio Development Kit and is used to stream A2DP audio from a Bluetooth host such as a smartphone to a pair of headphones connected to the Audio DAC Daughter Board which comes with the PIC32 Bluetooth Audio Development Kit. The smartphone is controlled using the AVRCP functions of the library. That demonstration can also automatically answer a voice call coming in via Hands-Free Protocol (HFP), interrupting (and pausing) any A2DP streaming in progress.

An example demonstration application using this library to interface with the BM64 for BLE functionality is BM64_ble_comm, which is used to send a string of characters to a smartphone when one of the push buttons is pressed on the PIC32 Bluetooth Audio Development Kit, and to receive a string of characters from the smartphone and display them on the LCD of the PIC32 Bluetooth Audio Development Kit, both using the "Transparent Service" feature of the BM64.

The following diagram shows the specific connections used in the PIC32 Bluetooth Audio Development Kit, which uses a PIC32MX470F512L microcontroller:

PIC32 Device and Module Connections



Using the Library

This topic describes the basic architecture of the BM64 Bluetooth Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_bm64.h`

The interface to the BM64 Bluetooth Driver library is defined in the `drv_bm64.h` header file. Any C language source (`.c`) file that uses the BM64 Bluetooth Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Library Source Files:

The BM64 Bluetooth Driver Library source files are provided in the `<install-dir>\framework\driver\bluetooth\bm64\src` directory. This folder may contain optional files and alternate implementations. Please refer to Configuring the Library for instructions on how to select optional features. and to Building the Library for instructions on how to build the library.

When the library is being used to stream A2DP audio from the BM64 to the PIC32, the BM64 must be configured as a I2S slave device. See the application BM64_bootloader demonstration application for instructions on how to do this.

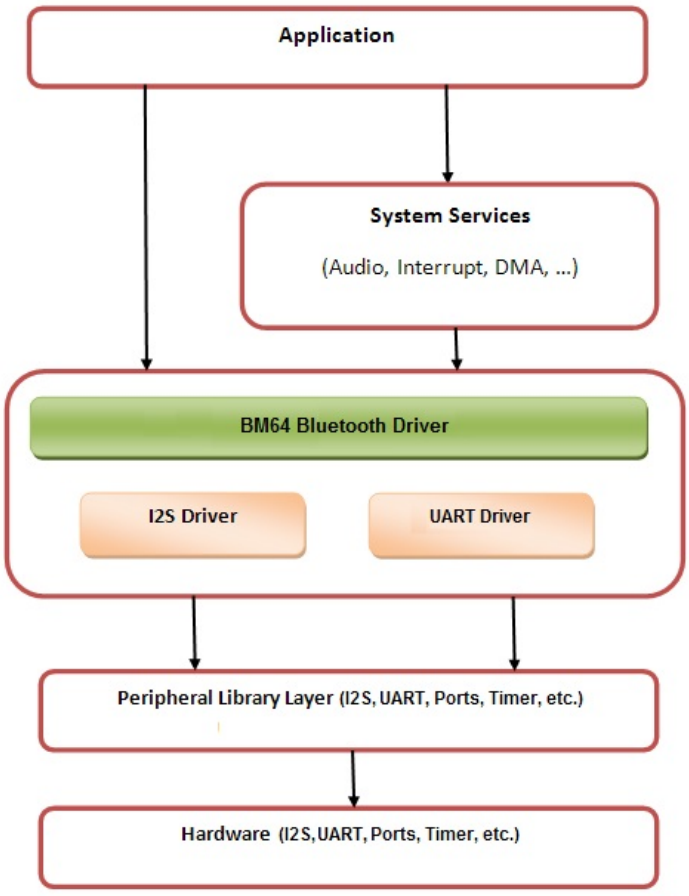
Abstraction Model

This library provides a low-level abstraction of the BM64 Bluetooth Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The abstraction model shown in the following diagram depicts how the BM64 Bluetooth Driver is positioned in the MPLAB Harmony framework. The BM64 Bluetooth Driver uses the USART Driver Library to control the BM64 and receive event notifications, the I2S Driver Library is used to receive audio from the BM64, and the Timer Driver for periodic timing.

BM64 Bluetooth Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The BM64 Bluetooth Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced BM64 Bluetooth module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the BM64 Bluetooth Driver Library.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, tasks and status functions.
Client Setup Functions	Provides open and close functions.
Data Transfer Functions	Provides data transfer functions.
Settings Functions	Provides driver specific functions for settings, such as volume control and sampling rate.
Bluetooth-specific Functions	Provides functions that are Bluetooth-specific.
AVRCP Functions	Provides functions that are used for AVRCP control.
Device Name and Address Functions	Provides functions for getting and setting the Bluetooth name and address.
BLE Functions	Provides BLE-specific functions.

How the Library Works

Provides information on how the library works.

Description

The library provides interfaces to support:

- System
- Client Setup
- Data Transfer
- Settings
- Bluetooth
- AVRCP
- Device Name and Address
- BLE

The library can be used by programs providing functionality for audio (A2DP, AVRCP and BLE), or BLE, or both.

For audio (A2DP/AVRCP/HFP), typically, there will be one simple state machine for the application and a second state machine just for the audio. After the application initializes, the audio state machine will open the BM64 Bluetooth Driver using a call to its Open function. Then, it will set up callbacks for each of two event handlers, and then open the codec driver using a call to its Open function and set up a callback from it. Then, the driver will wait until the BM64 initialization is complete, at which time the application state machine instructs the audio state machine to perform an initial buffer read from the BM64 using an AddRead call.

```
case AUDIO_STATE_OPEN:
{
    if (SYS_STATUS_READY == DRV_BT_Status())
    {
        // open BT module, including RX audio stream
        audioData.bt.handle = DRV_BT_Open(DRV_IO_INTENT_READ, DRV_BT_PROTOCOL_ALL);
        if(audioData.bt.handle != DRV_HANDLE_INVALID)
        {
            audioData.state = AUDIO_STATE_SET_BT_BUFFER_HANDLER;
        }
    }
}
break;
case AUDIO_STATE_SET_BT_BUFFER_HANDLER:
{
    DRV_BT_BufferEventHandlerSet(audioData.bt.handle, audioData.bt.bufferHandler,
                                 audioData.bt.context);
    DRV_BT_EventHandlerSet(audioData.bt.handle, audioData.bt.eventHandler, (uintptr_t)0);
                           audioData.state = AUDIO_STATE_CODEC_OPEN;
}
break;
case AUDIO_STATE_CODEC_OPEN:
```

```
{
    audioData.codec.handle =  DRV_CODEC_Open(DRV_CODEC_INDEX_0,  DRV_IO_INTENT_WRITE |
                                             DRV_IO_INTENT_EXCLUSIVE);
    if(audioData.codec.handle != DRV_HANDLE_INVALID)
    {
        audioData.state = AUDIO_STATE_CODEC_SET_BUFFER_HANDLER;
    }
}
break;
case AUDIO_STATE_CODEC_SET_BUFFER_HANDLER:
{
    _setCodecSamplingRate(DRV_BT_AUDIO_SAMPLING_RATE);
    DRV_CODEC_BufferEventHandlerSet(audioData.codec.handle, audioData.codec.bufferHandler,
                                    audioData.codec.context);
    audioData.state = AUDIO_STATE_INIT_DONE;
}
break;
case AUDIO_STATE_INIT_DONE:
{
    // waits in this state until BT initialization done and app state machine
    // calls audioStart() to set state to AUDIO_STATE_BT_SUBMIT_INITIAL_READS
    break;
}
```

After the initial buffer read has been completed, the buffer event handler for the BM64 will get a DRV_BT_BUFFER_EVENT_COMPLETE event. Once the queue has filled up, this will advance the audio state machine's state so that it adds the buffer to the codec's queue using its AddWrite function call. It then also makes a new call to the AddRead function to keep the queue filled.

When the buffer event handler for the codec gets a DRV_CODEC_BUFFER_EVENT_COMPLETE event, it will mark the buffer free for use again. See the BM64 demonstration application, BM64_a2dp_hfp, for more information and more example code.

BLE-only applications are much simpler since they do not have to process any audio. Again typically there will be one simple state machine for the application and a second state machine just for the BLE functionality. After the application initializes, the BLE state machine will open the BM64 Bluetooth driver using a call to its Open function, then it will set up a callback for an event handler.

The application will call one of the BLE Send functions to send data to the host (smartphone). The event handler will be called whenever data has been received from the BM64, or when the connection status changes. See the BM64 demonstration application, BM64_ble_comm, for more information and example code.

System Functions

This section describes the BM64 Bluetooth driver functions for initialization, maintaining task state and returning status.

Description

Initialization

The function DRV_BM64_Initialize is called by the function SYS_Initialize, in the file system_init.c, to initialize the BM64 Bluetooth driver using constants from the generated system_config.h file.

Tasks

The function DRV_BM64_Tasks is called from the System Task Service via the function SYS_Tasks in the file system_tasks.c to maintain the driver's internal control and data interface state machine.

One can use the function DRV_BM64_TasksReq to make a power on/power off task request (DRV_BM64_REQ_SYSTEM_OFF or DRV_BM64_REQ_SYSTEM_ON).

Status

The function DRV_BM64_Status returns the BM64 Bluetooth driver status, such as SYS_STATUS_READY, SYS_STATUS_BUSY, or SYS_STATUS_ERROR. The driver should not be opened until it has been marked ready.

Example:

```
// note generic version of call (DRV_BT instead of DRV_BM64) is used
if (SYS_STATUS_READY == DRV_BT_Status())
{
// This means the driver can be opened using the
// DRV_BT_Open() function.
}
```

The BM64-specific function DRV_BM64_GetPowerStatus returns the current power status, e.g. DRV_BM64_STATUS_OFF, DRV_BM64_STATUS_OFF, and DRV_BM64_STATUS_READY. Once it returns a ready status, this means the BM64 driver has completed its internal state machine initialization and can begin processing audio.

Example:
```
case APP_STATE_WAIT_INIT:
{
// note generic version of call (DRV_BT instead of DRV_BM64) is used
if (DRV_BT_STATUS_READY == DRV_BT_GetPowerStatus())
{
appData.state=APP_STATE_IDLE;
// can start processing audio
}
}
```

Client Functions

This section describes the BM64 Bluetooth driver functions for client setup (open, close, and setting up event handlers).

Description

Open and Close

For the application to start using an instance of the module, it must call the DRV_BM64_Open function which provides a driver handle to the BM64 Bluetooth driver instance.

Note: It is necessary to check the status of driver initialization before opening a driver instance. The status of the BM64 Bluetooth Driver can be known by calling DRV_BM64_Status.

Example:
```
case AUDIO_STATE_OPEN:
{
    if (SYS_STATUS_READY == DRV_BT_Status())
    {
        // open BT module, including RX audio stream
        audioData.bt.handle = DRV_BT_Open(DRV_IO_INTENT_READ, DRV_BT_PROTOCOL_ALL);
        if(audioData.bt.handle != DRV_HANDLE_INVALID)
        {
            audioData.state = AUDIO_STATE_SET_BT_BUFFER_HANDLER;
        }
    }
}
```

Event Handlers

Event handlers are functions in the user's code that are used as callbacks from the driver when something occurs that the client needs to know about.

The function DRV_BM64_BufferEventHandlerSet is called by a client to identify a buffer-related event handling function for the driver to call back. The prototype for the callback is defined by DRV_BM64_BUFFER_EVENT_HANDLER. The callback will be called with the event DRV_BT_BUFFER_EVENT_COMPLETE.

The function DRV_BM64_EventHandlerSet is called by a client to identify a general event handling function for the driver to call back. The prototype for the callback is defined by DRV_BM64_EVENT_HANDLER.

For audio applications, the callback will be called with events such as DRV_BT_EVENT_VOLUME_CHANGED, DRV_BT_EVENT_SAMPLERATE_CHANGED, and DRV_BT_EVENT_PLAYBACK_STATUS_CHANGED. For BLE applications, the callback will be called for events such as DRV_BT_EVENT_BLESPP_MSG_RECEIVED and DRV_BT_EVENT_BLE_STATUS_CHANGED.

Example:
```
case APP_STATE_SET_BT_BUFFER_HANDLER:
{
    // note generic version of calls (DRV_BT instead of DRV_BM64) are used
    DRV_BT_BufferEventHandlerSet(appData.bt.handle, appData.bt.bufferHandler,
                                 appData.bt.context);
    DRV_BT_EventHandlerSet(appData.bt.handle, appData.bt.eventHandler, (uintptr_t)0);
    appData.state = APP_STATE_CODEC_OPEN;
}
```

Data Transfer Function

This section describes the BM64 Bluetooth Driver data transfer function.

Description

The function DRV_BM64_BufferAddRead schedules a non-blocking read operation. It returns with a valid buffer handle in the bufferHandle

argument if the read request was scheduled successfully.

If the requesting client registered an event callback with the driver, the driver will issue a DRV_BM64_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully or DRV_BM64_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Example:

```
case APP_STATE_BT_BUFFER_COMPLETE:
{
    if (!_bufferUsed[appData.readIndex])
    {
       //Next BT Read Queued
       // note generic version of call (DRV_BT instead of DRV_BM64) is used
       DRV_BT_BufferAddRead(appData.bt.handle, &appData.bt.readBufHandle,
               audioBuffer[appData.readIndex],   appData.bt.bufferSize);
       if(appData.bt.readBufHandle != DRV_BT_BUFFER_HANDLE_INVALID)
       {
           appData.bt.readBufHandle = DRV_BT_BUFFER_HANDLE_INVALID;
           _bufferUsed[appData.readIndex] = true;
           appData.readIndex++;
           if(appData.readIndex >= AUDIO_QUEUE_SIZE)
           {
               appData.readIndex = 0;
           }
           appData.state = APP_STATE_BT_WAIT_FOR_BUFFER_COMPLETE;
       }
    }
}
```

Settings Functions

This section describes the BM64 Bluetooth Driver functions for getting and changing settings such as volume and sample rate.

Description

The function DRV_BM64_VolumeGet returns the volume for the current mode (A2DP or HFP) in percent (0-100), and the corresponding function DRV_BM64_VolumeSet sets the volume in percent.

The functions DRV_BM64_VolumeUp and DRV_BM64_VolumeDown turn the volume up and down on the host device (e.g. smartphone) by one increment (about 3% of full-scale). Either of these will result in a callback with the event DRV_BM64_EVENT_VOLUME_CHANGED specifying the new volume setting.

Example:

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    // bump the volume up one notch based on a button press
    if (BSP_SwitchStateGet(BSP_SWITCH_2)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_volumeUp(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}

. . .

// later, a call will come back to the event handler callback function
// (previously set up via a call to DRV_BM64_EventHandlerSet)
static void _BLEEventHandler(DRV_BT_EVENT event, uint32_t param, uintptr_t context)
{
    switch(event)
    {
        case DRV_BM64_EVENT_VOLUME_CHANGED:
        {
            uint16_t volume7bits = (127*param)/100;     // convert to 7 bits
            DRV_AK4384_VolumeSet(audioData.codec.handle,         // update codec's volume
                    DRV_AK4384_CHANNEL_LEFT_RIGHT,volume7bits);
            laString tempStr;
            char buf[5];
            sprintf(buf,"%3d%%",param);
            laWidget_SetVisible((laWidget*)GFX_VOLUME_VALUE, LA_TRUE);
            tempStr = laString_CreateFromCharBuffer(buf, &LiberationSans12);
            laLabelWidget_SetText(GFX_VOLUME_VALUE, tempStr);    // update screen
```

```
            laString_Destroy(&tempStr);
        }
    }
}
```

Sample Rate

This section describes the functions for getting and setting the sampling rate (e.g., 8000, 44100, or 48000 Hz) as a 32-bit integer.

Description

The function DRV_BM64_EnterBTPairingMode is used to enter into pairing mode. Once the BM64 is paired with a device, it will automatically attempt to connect with it again on the next power cycle.

Calling DRV_BM64_DisconnectAllLinks will disconnect the BM64 from the host (smartphone) but will not erase the pairing. So another call to the function DRV_BM64_LinkLastDevice will reconnect. However calling the function DRV_BM64_ForgetAllLinks will erase all pairing information, and another call to DRV_BM64_EnterBTPairingMode will be required to re-establish a connection.

Example:

```
    case BUTTON_STATE_PRESSED:      // (debouncing not shown)
    {
        // initiate pairing with a button press
        if (BSP_SwitchStateGet(BSP_SWITCH_1)==BSP_SWITCH_STATE_PRESSED))
        {
            // note generic version of call (DRV_BT instead of DRV_BM64) is used
            DRV_BT_EnterBTPairingMode(appData.bt.handle);
            appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
        }
    }
```

The function DRV_BM64_GetLinkStatus returns the current link status, returning an 8-bit value containing the current link status defined by DRV_BM64_LINKSTATUS enum. This can be used to restrict calls to AVRCP functions only when an AVRCP link is established.

Example:

```
   // note generic version of call (DRV_BT instead of DRV_BM64) is used
   if (DRV_BT_GetLinkStatus(appData.bt.handle) & DRV_BT_AVRCP_LINK_STATUS)
   {
       DRV_BT_CancelForwardOrRewind(appData.bt.handle);
   }
```

AVRCP Functions

This section describes the functions for getting and setting the Bluetooth device's name and address.

Description

The function DRV_BM64_SetBDName is called to set a temporary Bluetooth device name from an ASCII string buffer. The function DRV_BM64_GetBDName is called to get the current Bluetooth device name, and DRV_BM64_GetBDAddress is called to get the Bluetooth device address.

Example:

```
    laString tempStr;
    char buf [DRV_BT_MAXBDNAMESIZE+1];
      // note generic version of calls (DRV_BT instead of DRV_BM64) are used
    DRV_BT_GetBDName(appData.bt.handle, buf, DRV_BT_MAXBDNAMESIZE+1);
    tempStr = laString_CreateFromCharBuffer(buf, &LiberationSans12);
    laLabelWidget_SetText(GFX_BTNAME_VALUE, tempStr);   // display BT name
    laString_Destroy(&tempStr);
    DRV_BT_GetBDAddress(appData.bt.handle, buf);
    tempStr = laString_CreateFromCharBuffer(buf, &LiberationSans12);
    laLabelWidget_SetText(GFX_BTADDRESS_VALUE, tempStr);    // display BT address
    laString_Destroy(&tempStr);
```

BLE Functions

This section describes the functions specific to Bluetooth Low Energy (BLE) operations, such as sending and receiving data, and BLE connection-related operations.

Description

The function DRV_BM64_ReadByteFromBLE is used to receive data one byte at a time; the function DRV_BM64_ReadDataFromBLE is used to receive multiple bytes. Each of them return a Boolean, which is true if data is returned or false if there is no data to return. You can use the function DRV_BM64_ClearBLEData to clear out the receive buffer before starting.

Example:

```
uint8_t byte;
// note generic versions of calls (DRV_BT instead of DRV_BM64) are used
DRV_BT_ClearBLEData(appData.bt.handle);
// wait for byte to arrive
while (!DRV_BT_ReadByteFromBLE(appData.bt.handle, &byte))
{
    // should have some sort of way to break out of here if byte never arrives
}
```

Sending Data

The function DRV_BM64_SendByteOverBLE Is used to send one byte of data at a time; the function DRV_BM64_SendDataOverBLE is used to send multiple bytes of data.

Example:

```
#define BUFSIZE    100
uint8_t buf [BUFSIZE];
 // (code goes here to fill in buffer with data)

// note generic version of call (DRV_BT instead of DRV_BM64) is used
DRV_BT_SendDataOverBLE(appData.bt.handle, buf, BUFSIZE);
```

Connection Status

The function DRV_BM64_BLE_EnableAdvertising is called to enable or disable BLE advertising.

The function DRV_BM64_BLE_QueryStatus queries the BM64 to respond with a DRV_BM64_EVENT_BLE_STATUS_CHANGED event, which will indicate if the BM64 BLE status is standby, advertising, scanning or connected.

Example:

```
// note generic version of call (DRV_BT instead of DRV_BM64) is used
DRV_BT_BLE_QueryStatus(appData.bt.handle);
. . .
// later, a call will come back to the event handler callback function
// (previously set up via a call to DRV_BM64_EventHandlerSet)
static void _BLEEventHandler(DRV_BT_EVENT event, uint32_t param, uintptr_t context)
{
    switch(event)
    {
        case DRV_BT_EVENT_BLE_STATUS_CHANGED:
        {
            // do case switch based on param variable
        switch(param)
        {
            case DRV_BM64_BLE_STATUS_STANDBY:
            case DRV_BM64_BLE_STATUS_SCANNING:
                laWidget_SetVisible((laWidget*)GFX_CONNECTED, LA_FALSE);
                laWidget_SetVisible((laWidget*)GFX_PAIRED, LA_FALSE);
                laWidget_SetVisible((laWidget*)GFX_NOPAIR_NOCONNECTION, LA_TRUE);
                break;
            case DRV_BM64_BLE_STATUS_ADVERTISING:
                laWidget_SetVisible((laWidget*)GFX_CONNECTED, LA_FALSE);
                laWidget_SetVisible((laWidget*)GFX_PAIRED, LA_TRUE);    // actually, advertising
                laWidget_SetVisible((laWidget*)GFX_NOPAIR_NOCONNECTION, LA_FALSE);
                break;
            case DRV_BM64_BLE_STATUS_CONNECTED:
                laWidget_SetVisible((laWidget*)GFX_CONNECTED, LA_TRUE);
                laWidget_SetVisible((laWidget*)GFX_PAIRED, LA_FALSE);
                laWidget_SetVisible((laWidget*)GFX_NOPAIR_NOCONNECTION, LA_FALSE);
                break;
        }
   }
}
```

Configuring the Library

Macros

	Name	Description
	INCLUDE_BM64_BLE	Identifies whether the driver should include BLE

	INCLUDE_BM64_I2S	Identifies whether the driver should include HFP,A2DP,AVRCP functionality.
	INCLUDE_DEPRECATED_MMI_COMMANDS	Identifies whether the driver should use deprecated MMI commands.

Description

The configuration of the BM64 Bluetooth Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the BM64 Bluetooth Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the BM64 Bluetooth Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

INCLUDE_BM64_BLE Macro

Identifies whether the driver should include BLE

File

drv_bm64_config_template.h

C

```
#define INCLUDE_BM64_BLE
```

Description

Include BLE features?

Identifies whether the driver should include BLE (Bluetooth Low Energy) functions.

This option currently does not have any effect on the code size.

true (checked, default) - include BLE functionality. false (unchecked) - do not include BLE functionality.

Remarks

None

INCLUDE_BM64_I2S Macro

Identifies whether the driver should include HFP,A2DP,AVRCP functionality.

File

drv_bm64_config_template.h

C

```
#define INCLUDE_BM64_I2S
```

Description

Include HFP,A2DP,AVRCP protocols?

Identifies whether the driver should include the interface to support HFP, A2DP and AVRCP protocols, which by default also brings in the I2S driver and the default codec based on the BSP selected.

If you are building a BLE-only application, uncheck this option.

true (checked, default) - include HFP,A2DP,AVRCP functionality. false (unchecked) - do not include HFP,A2DP,AVRCP functionality.

Remarks

None

INCLUDE_DEPRECATED_MMI_COMMANDS Macro

Identifies whether the driver should use deprecated MMI commands.

File

drv_bm64_config_template.h

C

```
#define INCLUDE_DEPRECATED_MMI_COMMANDS
```

Description

Use Deprecated MMI Commands?

There are currently two versions of the BM64 Audio UART Command Set, which is used by the PIC32 to send commands to the BM64 module and receive responses (events) back from the BM64. The original is version 1.00 and the updated one is version 2.0x. Version 2.0x deprecates some MMI commands, and adds some new commands to replace them.

If the DRV_BM64_PlayPreviousSong and DRV_BM64_PlayNextSong functions are not working but other AVRCP functions are working properly, try unchcing this option.

true (checked, default) - use deprecated MMI commands. false (unchecked) - do not deprecated MMI commands.

Remarks

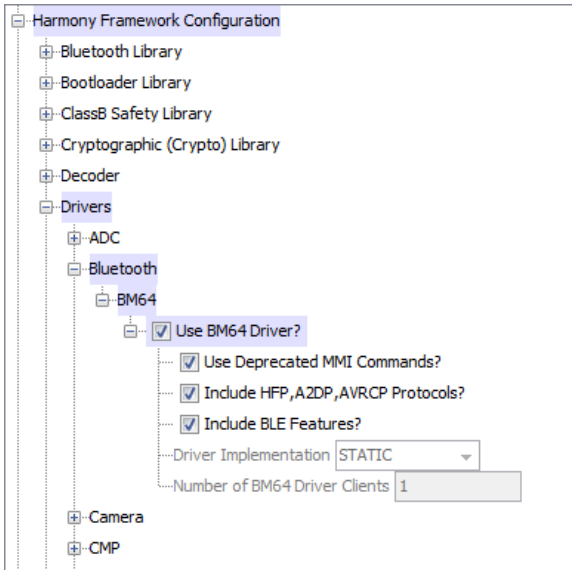
None

Configuring the MHC

Provides examples on how to configure the MPLAB Harmony Configurator (MHC) for a specific driver.

Description

The following figure shows an example MHC configuration for the BM64 Bluetooth Driver.



The option **Include HFP,A2DP,AVRCP protocols?** identifies whether the driver should include the interface to support HFP, A2DP and AVRCP protocols, which by default also brings in the I2S driver and the default codec based on the BSP selected. If you are building a BLE-only application, uncheck this option.

The option **Include BLE features?** identifies whether the driver should include BLE functions. If you are not using any BLE functionality, uncheck this option.

When **Use BM64 Driver?** is selected, and you have already selected the PIC32 Bluetooth Audio Development Kit (AK4384), the proper configuration for the AK4384, I2S, and Timer will have already been made for you, including:

- Under Drivers/CODEC,
  - Use_Codec_AK4384 selected
  - I2S Driver (used for data interface interface) instance set to DRV_I2S_INDEX_1
- Under I2S,
  - Use I2S Driver Selected
  - DMA Mode Selected
  - Transmit DMA Support Selected
  - Receive DMA Support Selected
  - Enable DMA Channel Interrupts selected
  - Sampling Rate set to 8000
- Number of I2S Instances set to 2
- I2S Driver Instance 0 selected
  - I2S Module ID set to SPI_ID_2 (BM64 Module as wired on BTADK)
  - Audio Protocol Mode set to DRV_I2S_AUDIO_I2S
- I2S Driver Instance 1 selected

- I2S Module ID set to SPI_ID_1 (AK4384 DAC Module as wired on BTADK)
- Audio Protocol Mode set to DRV_I2S_AUDIO_LFET_JUSTIFIED

Building the Library

This section lists the files that are available in the BM64 Bluetooth Driver Library.

Description

This section lists the files that are available in the `/src` folder of the BM64 Bluetooth Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/bluetooth/bm64`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_bm64.h	Header file that exports the driver API.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
drv_bm64_ble.h	Header file for the internal functions of the driver related to BLE.
drv_bm64_command_decode.h	Header file for the internal functions of the driver for decoding events from the BM64.
drv_bm64_command_send.h	Header file for the internal functions of the driver for sending commands to the BM64
drv_bm64_gpio.h	Header file for the internal functions of the driver related to the BM64's control pins.
drv_bm64_line_in.h	Header file for the internal functions of the driver related to the BM64's line in input.
./src/framework/driver/bluetooth/bm64/drv_bm64_local.h	Header file for the functions local to the BM64 driver (generated from template).
drv_bm64_sha1.h	Header file for the internal functions of the driver for performing SHA hashes.
drv_bm64_uart.h	Header file for the internal functions of the driver related to the BM64's UART interface.
./src/framework/driver/bluetooth/bm64/src/drv_bm64.c	Main source implementation file for the driver (generated from template).
src/drv_bm64_ble.c	Source file for the internal functions of the driver related to BLE.
src/drv_bm64_command_decode.c	Source file for the internal functions of the driver for decoding events from the BM64.
src/drv_bm64_command_send.c	Source file for the internal functions of the driver for sending commands to the BM64
src/drv_bm64_gpio.c	Source file for the internal functions of the driver related to the BM64's control pins.
src/drv_bm64_line_in.c	Source file for the internal functions of the driver related to the BM64's line in input.
src/drv_bm64_sha1.c	Source file for the internal functions of the driver for performing SHA hashs.
src/drv_bm64_uart.c	Source file for the internal functions of the driver related to the BM64's UART interface.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
There are no optional files for this driver.	N/A

Module Dependencies

The BM64 Bluetooth Driver Library depends on the following modules:

- I2S Driver Library
- Timer Driver Library
- USART Driver Library

Library Interface

a) System Functions

	Name	Description
	DRV_BM64_GetPowerStatus	Gets the current status of the BM64 Bluetooth driver module (BM64-specific).
	DRV_BM64_Initialize	Initializes hardware and data for the instance of the BM64 Bluetooth module
	DRV_BM64_Status	Gets the current system status of the BM64 Bluetooth driver module.
	DRV_BM64_TaskReq	Make a power on/power off task request.
	DRV_BM64_Tasks	Maintains the driver's control and data interface state machine.

b) Client Setup Functions

	Name	Description
	DRV_BM64_BufferEventHandlerSet	This function allows a client to identify a event handling function for the driver to call back.
	DRV_BM64_Close	Close an opened-instance of the BM64 Bluetooth driver.
	DRV_BM64_EventHandlerSet	This function allows a client to identify an event handling function for the driver to call back.
	DRV_BM64_Open	Open the specified BM64 driver instance and returns a handle to it

c) Data Transfer Functions

	Name	Description
	DRV_BM64_BufferAddRead	Schedule a non-blocking driver read operation.

d) Settings Functions

	Name	Description
	DRV_BM64_SamplingRateGet	Return the current sampling rate.
	DRV_BM64_SamplingRateSet	Set the current sampling rate.
	DRV_BM64_volumeDown	Turn the volume down on the host device.
	DRV_BM64_VolumeGet	returns 7-bit value 0-127
	DRV_BM64_VolumeSet	Set current volume.
	DRV_BM64_volumeUp	Turn the volume up on the host device.

e) Bluetooth-specific Functions

	Name	Description
	DRV_BM64_DisconnectAllLinks	Disconnect all links.
	DRV_BM64_EnterBTPairingMode	Enter Bluetooth pairing mode.
	DRV_BM64_ForgetAllLinks	Forget all pairings.
	DRV_BM64_GetLinkStatus	Return link status.
	DRV_BM64_LinkLastDevice	Link last device.

f) AVRCP Functions

	Name	Description
	DRV_BM64_CancelForwardOrRewind	Cancel previous fast forward or rewind request.
	DRV_BM64_FastForward	Fast forward the media.
	DRV_BM64_GetPlayingStatus	Return the current playing status of the device.
	DRV_BM64_Pause	Pause playback.
	DRV_BM64_Play	Start playback.
	DRV_BM64_PlayNextSong	Play the next song.
	DRV_BM64_PlayPause	Toggle play/pause mode.
	DRV_BM64_PlayPreviousSong	Play the previous song.
	DRV_BM64_Rewind	Rewind the media.
	DRV_BM64_Stop	Stop playback.

g) Device Name and Address Functions

	Name	Description
	DRV_BM64_GetBDAddress	Return the Bluetooth address.

	DRV_BM64_GetBDName	Return Bluetooth device name.
	DRV_BM64_SetBDName	Set the Bluetooth device name.

h) BLE Functions

	Name	Description
	DRV_BM64_ClearBLEData	Clear the BLE receive buffer.
	DRV_BM64_ReadByteFromBLE	Read a byte over BLE.
	DRV_BM64_ReadDataFromBLE	Read data over BLE.
	DRV_BM64_SendByteOverBLE	Send a byte over BLE.
	DRV_BM64_SendDataOverBLE	Send data over BLE.
	DRV_BM64_BLE_QueryStatus	Query BM64 LE status.
	DRV_BM64_BLE_EnableAdvertising	Enable or disable advertising.

i) Data Types and Constants

	Name	Description
	DRV_BM64_BUFFER_EVENT	This is macro DRV_BM64_BUFFER_EVENT.
	DRV_BM64_BUFFER_EVENT_COMPLETE	This is macro DRV_BM64_BUFFER_EVENT_COMPLETE.
	DRV_BM64_BUFFER_HANDLE	This is macro DRV_BM64_BUFFER_HANDLE.
	DRV_BM64_BUFFER_HANDLE_INVALID	This is macro DRV_BM64_BUFFER_HANDLE_INVALID.
	DRV_BM64_DATA32	BM64 defines based on I2S interface
	DRV_BM64_MAXBDNAMESIZE	
	DRV_BM64_BUFFER_EVENT_HANDLER	prototype for callback for DRV_BM64_BufferEventHandlerSet
	DRV_BM64_DRVR_STATUS	BM64 driver status
	DRV_BM64_EVENT	events that can be returned to a client via callback
	DRV_BM64_EVENT_HANDLER	prototype for callback for DRV_BM64_EventHandlerSet
	DRV_BM64_LINKSTATUS	BM64 link status
	DRV_BM64_PLAYINGSTATUS	This is type DRV_BM64_PLAYINGSTATUS.
	DRV_BM64_PROTOCOL	BM64 protocols
	DRV_BM64_REQUEST	BM64 power on/off request
	DRV_BM64_SAMPLE_FREQUENCY	BM64 sample frequency
	DRV_BM64_BLE_STATUS	This is type DRV_BM64_BLE_STATUS.

Description

This section describes the API functions of the BM64 Bluetooth Driver library.

Refer to each section for a detailed description.

a) System Functions

DRV_BM64_GetPowerStatus Function

Gets the current status of the BM64 Bluetooth driver module (BM64-specific).

File

drv_bm64.h

C

```
DRV_BM64_DRVR_STATUS DRV_BM64_GetPowerStatus();
```

Returns

Driver status, encoded as type DRV_BM64_DRVR_STATUS enum.

Description

Function DRV_BM64_GetPowerStatus:

DRV_BM64_DRVR_STATUS DRV_BM64_GetPowerStatus(void);

This routine provides the current status (power on/off/ready) of the BM64 Bluetooth driver module passed back as type DRV_BM64_DRVR_STATUS enum.

Remarks

A status of DRV_BT_STATUS_READY means the drivers state machine has finished initialization and is ready to stream audio.

Preconditions

DRV_BM64_Initialize must have been called to initialize the driver instance.

Example

```
case APP_STATE_WAIT_INIT:
{
   // note generic version of call (DRV_BT instead of DRV_BM64) is used
   if (DRV_BT_STATUS_READY == DRV_BT_GetPowerStatus())
   {
       appData.state=APP_STATE_IDLE;
       // start can processing audio
   }
}
break;
```

DRV_BM64_Initialize Function

Initializes hardware and data for the instance of the BM64 Bluetooth module

File

drv_bm64.h

C

```
void DRV_BM64_Initialize();
```

Returns

None.

Description

Function DRV_BM64_Initialize:

void DRV_BM64_Initialize(void);

This routine initializes the BM64 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

Remarks

This routine must be called before any other BM64 driver routine is called. This routine should only be called once during system initialization. This routine will never block for hardware access.

Preconditions

None.

Example

```
// (in SYS_Initialize, system_init.c)*

DRV_BM64_Initialize();
```

DRV_BM64_Status Function

Gets the current system status of the BM64 Bluetooth driver module.

File

drv_bm64.h

C

```
SYS_STATUS DRV_BM64_Status();
```

Returns

Driver status, encoded as type SYS_STATUS enum:

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed SYS_STATUS_ERROR - Indicates that the specified module is in an error state *

Description

Function DRV_BM64_Status:

SYS_STATUS DRV_BM64_Status(void);

This routine provides the current status of the BM64 Bluetooth driver module, passed back as type SYS_STATUS.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

None.

Example

```
* // note generic version of call (DRV_BT instead of DRV_BM64) is used
if (SYS_STATUS_READY == DRV_BT_Status())
{
    // This means the driver can be opened using the
    // DRV_BT_Open() function.
}
```

DRV_BM64_TaskReq Function

Make a power on/power off task request.

File

drv_bm64.h

C

```
void DRV_BM64_TaskReq(DRV_BM64_REQUEST request);
```

Returns

None.

Description

Function DRV_BM64_TaskReq:

void DRV_BM64_TaskReq(DRV_BM64_REQUEST request);

Make a power on/power off task request using the DRV_BM64_REQUEST enum.

Remarks

None.

Preconditions

DRV_BM64_Initialize must have been called to initialize the driver instance.

Example

```
// note generic version of call (DRV_BT instead of DRV_BM64) is used
DRV_BT_TaskReq(DRV_BM64_REQ_SYSTEM_ON);
```

Parameters

Parameters	Description
request	power on/off request of type DRV_BM64_REQUEST

DRV_BM64_Tasks Function

Maintains the driver's control and data interface state machine.

File

drv_bm64.h

C

```
void DRV_BM64_Tasks();
```

Returns

None.

Description

Function DRV_BM64_Tasks:

void DRV_BM64_Tasks(void);

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations.

This function should be called from the SYS_Tasks() function.

Remarks

This routine is not normally called directly by an application. Instead it is called by the system's Tasks routine (SYS_Tasks).

Preconditions

None.

Example

```
// (in SYS_Tasks, system_tasks.c)

// Maintain Device Drivers
DRV_BM64_Tasks();
```

b) Client Setup Functions

DRV_BM64_BufferEventHandlerSet Function

This function allows a client to identify a event handling function for the driver to call back.

File

drv_bm64.h

C

```
void DRV_BM64_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_BM64_BUFFER_EVENT_HANDLER eventHandler,
const uintptr_t contextHandle);
```

Returns

None.

Description

Function DRV_BM64_BufferEventHandlerSet:

void DRV_BM64_EventHandlerSet(DRV_HANDLE handle, const DRV_BM64_BUFFER_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

When a client calls DRV_BM64_BufferAddRead function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The context parameter contains a handle to the client context, provided at the time the event handling function is registered using the DRV_BM64_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client.

The event handler should be set before the client performs any "BM64 Bluetooth Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case APP_STATE_SET_BT_BUFFER_HANDLER:
{
    // note generic version of call (DRV_BT instead of DRV_BM64) is used
    DRV_BT_BufferEventHandlerSet(appData.bt.handle,
                                 appData.bt.bufferHandler,
                                 appData.bt.context);

    DRV_BT_EventHandlerSet(appData.bt.handle,
                                 appData.bt.eventHandler,
                                 (uintptr_t)0);

    appData.state = APP_STATE_CODEC_OPEN;
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client
eventHandler	pointer to a function to be called back (prototype defined by DRV_BM64_BUFFER_EVENT_HANDLER)
contextHandle	handle to the client context

DRV_BM64_Close Function

Close an opened-instance of the BM64 Bluetooth driver.

File

drv_bm64.h

C

```
void DRV_BM64_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_Close:

void DRV_BM64_Close(DRV_HANDLE handle);

This routine closes an opened-instance of the BM64 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_BM64_Open before the caller may use the driver again

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
// note generic version of call (DRV_BT instead of DRV_BM64) is used
DRV_BT_Close(appData.bt.handle);
```

*

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_EventHandlerSet Function

This function allows a client to identify an event handling function for the driver to call back.

File

drv_bm64.h

C

```
void DRV_BM64_EventHandlerSet(DRV_HANDLE handle, const DRV_BM64_EVENT_HANDLER eventHandler, const uintptr_t
contextHandle);
```

Returns

None.

Description

Function DRV_BM64_EventHandlerSet:

void DRV_BM64_EventHandlerSet(DRV_HANDLE handle, const DRV_BM64_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);

This function allows a client to identify a command event handling function for the driver to call back when an event has been received from the BM64.

The context parameter contains a handle to the client context, provided at the time the event handling function is registered using the DRV_BM64_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client.*

The event handler should be set before the client performs any "BM64 Bluetooth Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when an event has occurred, it does not need to register a callback.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case APP_STATE_SET_BT_BUFFER_HANDLER:
{
    DRV_BT_BufferEventHandlerSet(appData.bt.handle,
                                 appData.bt.bufferHandler,
                                 appData.bt.context);

    // note generic version of call (DRV_BT instead of DRV_BM64) is used
    DRV_BT_EventHandlerSet(appData.bt.handle,
                                 appData.bt.eventHandler,
                                 (uintptr_t)0);

    appData.state = APP_STATE_CODEC_OPEN;
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client
eventHandler	pointer to a function to be called back (prototype defined by DRV_BM64_EVENT_HANDLER)
contextHandle	handle to the client context

DRV_BM64_Open Function

Open the specified BM64 driver instance and returns a handle to it

File

drv_bm64.h

C

```
DRV_HANDLE DRV_BM64_Open(const DRV_IO_INTENT ioIntent, const DRV_BM64_PROTOCOL protocol);
```

Returns

valid handle to an opened BM64 device driver unique to client

Description

Function DRV_BM64_Open:

DRV_HANDLE DRV_BM64_Open(const DRV_IO_INTENT ioIntent, const DRV_BM64_PROTOCOL protocol);

This routine opens the specified BM64 Bluetooth driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

Only DRV_IO_INTENT_READ is a valid ioIntent option as the BM64 Bluetooth driver audio stream is read-only.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_BM64_Close routine is called. This routine will never block waiting for hardware. If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Currently only one client is allowed at a time.

Preconditions

DRV_BM64_Initialize must have been called to initialize the driver instance.

Example

```
case APP_STATE_OPEN:
{
    if (SYS_STATUS_READY == DRV_BT_Status())
    {
        // open BT module, including RX audio stream
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        appData.bt.handle = DRV_BT_Open(DRV_IO_INTENT_READ, DRV_BT_PROTOCOL_ALL);

        if(appData.bt.handle != DRV_HANDLE_INVALID)
        {
            appData.state = APP_STATE_SET_BT_BUFFER_HANDLER;
        }
        else
        {
            // Got an Invalid Handle.  Wait for BT module to Initialize
        }
    }
}
break;
```

Parameters

Parameters	Description
ioIntent	valid handle to an opened BM64 device driver unique to client
protocol	specifies which protocol(s) the client intends to use with this driver. One of the various DRV_BM64_PROTOCOL enum values, including DRV_BM64_PROTOCOL_ALL.

c) Data Transfer Functions

DRV_BM64_BufferAddRead Function

Schedule a non-blocking driver read operation.

File

drv_bm64.h

C

```
void DRV_BM64_BufferAddRead(const DRV_HANDLE handle, DRV_BM64_BUFFER_HANDLE * bufferHandle, void * buffer,
size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_BM64_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_BM64_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_BM64_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_BM64_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the BM64 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another BM64 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case APP_STATE_BT_BUFFER_COMPLETE:
{
   //BT RX
   if (!_bufferUsed[appData.readIndex])
   {
       //Next BT Read Queued
       // note generic version of call (DRV_BT instead of DRV_BM64) is used
       DRV_BT_BufferAddRead(appData.bt.handle,
                            &appData.bt.readBufHandle,
                            audioBuffer[appData.readIndex],
                            appData.bt.bufferSize);

       if(appData.bt.readBufHandle != DRV_BT_BUFFER_HANDLE_INVALID)
       {
           appData.bt.readBufHandle = DRV_BT_BUFFER_HANDLE_INVALID;
           _bufferUsed[appData.readIndex] = true;

           //QUEUE HEAD Index (for next BT read)
           appData.readIndex++;
           if(appData.readIndex >= AUDIO_QUEUE_SIZE)
           {
               appData.readIndex = 0;
           }
           appData.state = APP_STATE_BT_WAIT_FOR_BUFFER_COMPLETE;
       }
       else
       {
           SYS_DEBUG(0, "BT Buffer Read FAILED!!!");
       }
   }
   else
   {
       //Overrun -- Wait for Read buffer to become available.
       SYS_DEBUG(0, "Buffer Overrunrn");
   }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client
bufferHandle	pointer to an argument that contains the return buffer handle
buffer	pointer to buffer that will contain received data
size	buffer size in bytes.

Function

void DRV_BM64_BufferAddRead(const DRV_HANDLE handle,
DRV_BM6_BUFFER_HANDLE *bufferHandle, void *buffer, size_t size)

d) Settings Functions

DRV_BM64_SamplingRateGet Function

Return the current sampling rate.

File

drv_bm64.h

C

```
uint32_t DRV_BM64_SamplingRateGet(DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_SamplingRateGet:
uint32_t DRV_BM64_SamplingRateGet(DRV_HANDLE handle);
Return the current sampling rate as a 32-bit integer.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
uint32_t sampleRate;

// note generic version of call (DRV_BT instead of DRV_BM64) is used
sampleRate = DRV_BT_SamplingRateGet(appData.bt.handle);
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_SamplingRateSet Function

Set the current sampling rate.

File

drv_bm64.h

C

```
void DRV_BM64_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

Returns

None.

Description

Function DRV_BM64_SamplingRateSet:

void DRV_BM64_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);

Set the current sampling rate (passed as a 32-bit integer).

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
// set sample rate to 44.1 kHz
// note generic version of call (DRV_BT instead of DRV_BM64) is used
DRV_BT_SamplingRateSet(appData.bt.handle, 44100);
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client
samplingRate	sampling rate in Hz (8000, 16000, 44100 or 48000)

DRV_BM64_volumeDown Function

Turn the volume down on the host device.

File

drv_bm64.h

C

```
void DRV_BM64_volumeDown(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_VolumeDown:

void DRV_BM64_VolumeDown(const DRV_HANDLE handle);

Turn the volume down on the host device by one increment (about 3% of full-scale).

Remarks

This will result in a callback with the event DRV_BM64_EVENT_VOLUME_CHANGED specifying the new volume setting for the codec.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_2)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_volumeUp(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_VolumeGet Function

File

drv_bm64.h

C

```
uint8_t DRV_BM64_VolumeGet(const DRV_HANDLE handle);
```

Description

returns 7-bit value 0-127

DRV_BM64_VolumeSet Function

Set current volume.

File

drv_bm64.h

C

```
void DRV_BM64_VolumeSet(const DRV_HANDLE handle, uint8_t volume);
```

Returns

None.

Description

Function DRV_BM64_VolumeSet:

void DRV_BM64_VolumeSet(const DRV_HANDLE handle, uint8_t volume);

Set volume for current mode (A2DP, HFP etc.) in percent (0-100).

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
// note generic version of call (DRV_BT instead of DRV_BM64) is used *
volume = DRV_BT_VolumeGet(appData.bt.handle,50);       // set volume to 50%
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client
volume	volume level in percent, 0-100

DRV_BM64_volumeUp Function

Turn the volume up on the host device.

File

drv_bm64.h

C

```
void DRV_BM64_volumeUp(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_VolumeUp:

void DRV_BM64_VolumeUp(const DRV_HANDLE handle);

Turn the volume up on the host device by one increment (about 3% of full-scale).

Remarks

This will result in a callback with the event DRV_BM64_EVENT_VOLUME_CHANGED specifying the new volume setting for the codec.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_1)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_volumeUp(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

e) Bluetooth-specific Functions

DRV_BM64_DisconnectAllLinks Function

Disconnect all links.

File

drv_bm64.h

C

```
void DRV_BM64_DisconnectAllLinks(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_DisconnectAllLinks:

void DRV_BM64_DisconnectAllLinks(const DRV_HANDLE handle);

Disconnect all current links to a Bluetooth host.

Remarks

Does not unpair the device, just disconnects. Use DRV_BM64_LinkLastDevice to reconnect. Use DRV_BM64_ForgetAllLinks to forget all pairings.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_2)==BSP_SWITCH_STATE_PRESSED))
```

```
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_DisconnectAllLinks(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_EnterBTPairingMode Function

Enter Bluetooth pairing mode.

File

drv_bm64.h

C

```
void DRV_BM64_EnterBTPairingMode(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_EnterBTPairingMode:

void DRV_BM64_EnterBTPairingMode(const DRV_HANDLE handle);

Starting the pairing process, making this BM64 available for pairing with a Bluetooth host.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_1)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_EnterBTPairingMode(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_ForgetAllLinks Function

Forget all pairings.

File

drv_bm64.h

C

```
void DRV_BM64_ForgetAllLinks(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_ForgetAllLinks:

void DRV_BM64_ForgetAllLinks(const DRV_HANDLE handle);

Forget (erase) all links and pairings stored in EEPROM.

Remarks

After this is called, one must call DRV_BM64_EnterBTPairingMode to establish a connection to a Bluetooth host again.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_2)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_ForgetAllLinks(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_GetLinkStatus Function

Return link status.

File

drv_bm64.h

C

```
DRV_BM64_LINKSTATUS DRV_BM64_GetLinkStatus(const DRV_HANDLE handle);
```

Returns

8-bit value defined by DRV_BM64_LINKSTATUS enum.

Description

Function DRV_BM64_GetLinkStatus:

DRV_BM64_LINKSTATUS DRV_BM64_GetLinkStatus(const DRV_HANDLE handle);

Returns a 8-bit value containing current link status as bit flags for SCO (bit 0), ACL, HFP, A2DP, AVRCP, SPP, IAP, MAP (bit 7)

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_3)==BSP_SWITCH_STATE_PRESSED))
    {
        DRV_BT_PLAYINGSTATUS playingStatus = DRV_BT_GetPlayingStatus(appData.bt.handle);
        if ((playingStatus==DRV_BT_PLAYING_FF)||(playingStatus==DRV_BT_PLAYING_FR))
        {
```

```
            // note generic version of call (DRV_BT instead of DRV_BM64) is used
            if (DRV_BT_GetLinkStatus(appData.bt.handle) & DRV_BT_AVRCP_LINK_STATUS)
            {
                DRV_BT_CancelForwardOrRewind(appData.bt.handle);
            }
        }
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_LinkLastDevice Function

Link last device.

File

drv_bm64.h

C

```
void DRV_BM64_LinkLastDevice(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_LinkLastDevice:

void DRV_BM64_LinkLastDevice(const DRV_HANDLE handle);

Link (connect) to last device that was previously linked.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_2)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_LinkLastDevice(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

f) AVRCP Functions

DRV_BM64_CancelForwardOrRewind Function

Cancel previous fast forward or rewind request.

File

drv_bm64.h

C

```
void DRV_BM64_CancelForwardOrRewind(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_CancelForwardOrRewind:

void DRV_BM64_CancelForwardOrRewind(const DRV_HANDLE handle);

Send an AVRCP command to the host device to cancel a previous fast forward or rewind request.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_3)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_CancelForwardOrRewind(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_FastForward Function

Fast forward the media.

File

drv_bm64.h

C

```
void DRV_BM64_FastForward(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_FastForward:

void DRV_BM64_FastForward(const DRV_HANDLE handle);

Send an AVRCP command to the host device to Fast forward the media.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_5)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_FastForward(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_GetPlayingStatus Function

Return the current playing status of the device.

File

drv_bm64.h

C

```
DRV_BM64_PLAYINGSTATUS DRV_BM64_GetPlayingStatus(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_GetPlayingStatus:

void DRV_BM64_GetPlayingStatus(const DRV_HANDLE handle);

Return the current AVRCP playing status of the device, e.g. stopped, playing, paused, fast forward or rewind, encoded as as the enum DRV_BM64_PLAYINGSTATUS.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_3)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_PLAYINGSTATUS playingStatus = DRV_BT_GetPlayingStatus(appData.bt.handle);
        if ((playingStatus==DRV_BT_PLAYING_FF)||(playingStatus==DRV_BT_PLAYING_FR))
        {
           if (DRV_BT_GetLinkStatus(appData.bt.handle) & DRV_BT_AVRCP_LINK_STATUS)
           {
               DRV_BT_CancelForwardOrRewind(appData.bt.handle);
           }
        }
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_Pause Function

Pause playback.

File

drv_bm64.h

C

```
void DRV_BM64_Pause(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_Pause:

void DRV_BM64_Pause(const DRV_HANDLE handle);

Send an AVRCP command to the host device to pause.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_3)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_Pause(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_Play Function

Start playback.

File

drv_bm64.h

C

```
void DRV_BM64_Play(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_Play:

DRV_BM64_Play(const DRV_HANDLE handle);

Send an AVRCP command to the host device to initiate or resume playback.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_3)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_Play(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_PlayNextSong Function

Play the next song.

File

drv_bm64.h

C

```
void DRV_BM64_PlayNextSong(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_PlayNextSong:

void DRV_BM64_PlayNextSong(const DRV_HANDLE handle);

Send an AVRCP command to the host device to play the next song in a playlist.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_3)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_PlayNextSong(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_PlayPause Function

Toggle play/pause mode.

File

drv_bm64.h

C

```
void DRV_BM64_PlayPause(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_PlayPause:

void DRV_BM64_PlayPause(const DRV_HANDLE handle);

Send an AVRCP command to the host device to toggle the play/pause mode.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_3)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_PlayPause(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_PlayPreviousSong Function

Play the previous song.

File

drv_bm64.h

C

```
void DRV_BM64_PlayPreviousSong(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_PlayPreviousSong:

void DRV_BM64_PlayPreviousSong(const DRV_HANDLE handle);

Send an AVRCP command to the host device to play the previous song in a playlist.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_5)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_PlayPreviousSong(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_Rewind Function

Rewind the media.

File

drv_bm64.h

C

```
void DRV_BM64_Rewind(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_Rewind:

void DRV_BM64_Rewind(const DRV_HANDLE handle);

Send an AVRCP command to the host device to rewind the media.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_5)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_Rewind(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_Stop Function

Stop playback.

File

drv_bm64.h

C

```
void DRV_BM64_Stop(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_Stop:

void DRV_BM64_Stop(const DRV_HANDLE handle);

Send an AVRCP command to the host device to stop playback.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
case BUTTON_STATE_PRESSED:      // (debouncing not shown)
{
    if (BSP_SwitchStateGet(BSP_SWITCH_3)==BSP_SWITCH_STATE_PRESSED))
    {
        // note generic version of call (DRV_BT instead of DRV_BM64) is used
        DRV_BT_Stop(appData.bt.handle);
        appData.buttonState=BUTTON_STATE_WAIT_FOR_RELEASE;
    }
}
break;
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

g) Device Name and Address Functions

DRV_BM64_GetBDAddress Function

Return the Bluetooth address.

File

drv_bm64.h

C

```
void DRV_BM64_GetBDAddress(const DRV_HANDLE handle, char* buffer);
```

Returns

None.

Description

Function DRV_BM64_GetBDAddress:

void DRV_BM64_GetBDAddress(const DRV_HANDLE handle, char* buffer);

Return the Bluetooth address of the device as an ASCII string.

Remarks

Buffer must be at least 18 bytes in length (6 octets separated by ?:?, e.g. able to hold "12:34:56:78:90:120").

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
laString tempStr;
char buf [18];

// note generic version of call (DRV_BT instead of DRV_BM64) is used
DRV_BT_GetBDAddress(appData.bt.handle, buf);
tempStr = laString_CreateFromCharBuffer(buf, &LiberationSans12);
laLabelWidget_SetText(GFX_BTADDRESS_VALUE, tempStr);    // display BT address
laString_Destroy(&tempStr);
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client
buffer	pointer to a char buffer at least 18 bytes long

DRV_BM64_GetBDName Function

Return Bluetooth device name.

File

drv_bm64.h

C

```
void DRV_BM64_GetBDName(const DRV_HANDLE handle, char* buffer, const uint8_t buflen);
```

Returns

None.

Description

Function DRV_BM64_GetBDName:

void DRV_BM64_GetBDName(const DRV_HANDLE handle, char* buffer, const uint8_t buflen);

Return the Bluetooth device name as an ASCII string.

Remarks

If name is longer than buflen-1 bytes long, it will be truncated to fit inside the buffer.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
laString tempStr;
char buf [DRV_BT_MAXBDNAMESIZE+1];

// note generic version of call (DRV_BT instead of DRV_BM64) is used
DRV_BT_GetBDName(appData.bt.handle, buf, DRV_BT_MAXBDNAMESIZE+1);
tempStr = laString_CreateFromCharBuffer(buf, &LiberationSans12);
laLabelWidget_SetText(GFX_BTNAME_VALUE, tempStr);   // display BT name
laString_Destroy(&tempStr);
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client
buffer	pointer to a char buffer at least buflen bytes long
buflen	length of buffer (including terminating 0 byte)

DRV_BM64_SetBDName Function

Set the Bluetooth device name.

File

drv_bm64.h

C

```
void DRV_BM64_SetBDName(const DRV_HANDLE handle, const char* buffer);
```

Returns

None.

Description

Function DRV_BM64_SetBDName:

void DRV_BM64_SetBTName(const DRV_HANDLE handle, const char* buffer);

Set a temporary Bluetooth device name from an ASCII string buffer.

Remarks

The name is set for this session only; if the BM64 is reset (e.g. power is lost) the name will revert to the Bluetooth name stored in EEPROM.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
// note generic version of call (DRV_BT instead of DRV_BM64) is used
DRV_BT_SetBDName(appData.bt.handle, "Temporary BM64 Name");
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client
buffer	pointer to a char buffer containing the new name

h) BLE Functions

DRV_BM64_ClearBLEData Function

Clear the BLE receive buffer.

File

drv_bm64.h

C

```
void DRV_BM64_ClearBLEData(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_ClearBLEData:

void DRV_BM64_ClearBLEData(const DRV_HANDLE handle);

Clears the buffer used when receiving characters via the DRV_BM64_ReadByteFromBLE and DRV_BM64_ReadDataFromBLE calls.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
uint8_t byte;

// note generic versions of calls (DRV_BT instead of DRV_BM64) is used
DRV_BT_ClearBLEData(appData.bt.handle);

// wait for byte to arrive
while (!DRV_BT_ReadByteFromBLE(appData.bt.handle, &byte))
{
    // should have some sort of way to break out of here if byte never arrives
}
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_ReadByteFromBLE Function

Read a byte over BLE.

File

drv_bm64.h

C

```
bool DRV_BM64_ReadByteFromBLE(const DRV_HANDLE handle, uint8_t* byte);
```

Returns

bool - true if a byte was returned, false if receive buffer empty

Description

Function DRV_BM64_ReadByteFromBLE:

bool DRV_BM64_ReadByteFromBLE(const DRV_HANDLE handle, uint8_t* byte);

Read one byte over BLE using the BM64's "Transparent Service" feature.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
uint8_t byte;

// note generic version of call (DRV_BT instead of DRV_BM64) is used
if (DRV_BT_ReadByteFromBLE(appData.bt.handle, &byte))  // if byte received
{
    // do something
}
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client
byte	pointer to a uint8_t to receive the data

DRV_BM64_ReadDataFromBLE Function

Read data over BLE.

File

drv_bm64.h

C

```
bool DRV_BM64_ReadDataFromBLE(const DRV_HANDLE handle, uint8_t* byte, uint16_t size);
```

Returns

bool - true if data was returned, false if receive buffer empty

Description

Function DRV_BM64_ReadDataFromBLE:

bool DRV_BM64_ReadDataFromBLE(const DRV_HANDLE handle, uint8_t* bytes, uint16_t size);

Read data over BLE using the BM64's "Transparent Service" feature.

Remarks

No more than size bytes will be returned, even if more are available.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
#define BUFSIZE  100
uint8_t buf [BUFSIZE];

// note generic version of call (DRV_BT instead of DRV_BM64) is used
if (DRV_BT_ReadDataFromBLE(appData.bt.handle, buf, BUFSIZE))  // if data received
{
    // do something
}
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client
bytes	pointer to a uint8_t buffer at least size bytes long
size	length of buffer (including

DRV_BM64_SendByteOverBLE Function

Send a byte over BLE.

File

drv_bm64.h

C

```
void DRV_BM64_SendByteOverBLE(const DRV_HANDLE handle, uint8_t byte);
```

Returns

None.

Description

Function DRV_BM64_SendByteOverBLE:

void DRV_BM64_SendByteOverBLE(const DRV_HANDLE handle, uint8_t byte);

Send one byte over BLE using the BM64's "Transparent Service" feature.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
uint8_t byte;

byte = 10;     // set to some value
```

```
// note generic version of call (DRV_BT instead of DRV_BM64) is used
DRV_BT_SendByteOverBLE(appData.bt.handle, byte);
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client
byte	uint8_t of data to be sent

DRV_BM64_SendDataOverBLE Function

Send data over BLE.

File

drv_bm64.h

C

```
void DRV_BM64_SendDataOverBLE(const DRV_HANDLE handle, uint8_t* bytes, uint16_t size);
```

Returns

None.

Description

Function DRV_BM64_SendDataOverBLE:

void DRV_BM64_SendDataOverBLE(const DRV_HANDLE handle, uint8_t* bytes, uint16_t size);

Send data over BLE using the BM64's "Transparent Service" feature.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
#define BUFSIZE    100
uint8_t buf [BUFSIZE];

// (code to fill in buffer with data)

// note generic version of call (DRV_BT instead of DRV_BM64) is used
DRV_BT_SendDataOverBLE(appData.bt.handle, buf, BUFSIZE);
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client
bytes	pointer to a uint8_t buffer at least size bytes long
size	length of buffer (including

DRV_BM64_BLE_QueryStatus Function

Query BM64 LE status.

File

drv_bm64.h

C

```
void DRV_BM64_BLE_QueryStatus(const DRV_HANDLE handle);
```

Returns

None.

Description

Function DRV_BM64_BLE_QueryStatus:

void DRV_BM64_BLE_QueryStatus(const DRV_HANDLE handle);

Queries the BM64 to respond with a DRV_BM64_EVENT_BLE_STATUS_CHANGED event, which will indicate if the BM64 BLE status is standby, advertising, scanning or connected.

Remarks

RV_BM64_BLE_QueryStatus is non-blocking; it returns right away and sometime later (perhaps tens or hundreds of ms) the event handler callback will be called.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
// note generic version of call (DRV_BT instead of DRV_BM64) is used
DRV_BT_BLE_QueryStatus(appData.bt.handle);

. . .

// later, a call will come back to the event handler callback function
// (previously set up via a call to DRV_BM64_EventHandlerSet)
static void _BLEEventHandler(DRV_BT_EVENT event, uint32_t param, uintptr_t context)
{
    switch(event)
    {
        case DRV_BT_EVENT_BLE_STATUS_CHANGED:
        {
            // do case switch based on param variable
        }
   }
}
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client

DRV_BM64_BLE_EnableAdvertising Function

Enable or disable advertising.

File

drv_bm64.h

C

```
void DRV_BM64_BLE_EnableAdvertising(const DRV_HANDLE handle, bool enable);
```

Returns

None.

Description

Function DRV_BM64_BLE_EnableAdvertising:

void DRV_BM64_BLE_EnableAdvertising(const DRV_HANDLE handle, bool enable);

Enable or disable BLE advertising.

Remarks

None.

Preconditions

DRV_BM64_Open must have been called to obtain a valid opened device handle.

Example

```
// note generic version of call (DRV_BT instead of DRV_BM64) is used
```

```
DRV_BM64_BLE_EnableAdvertising(appData.bt.handle, true);
```

Parameters

Parameters	Description
handle	valid handle to an opened BM64 device driver unique to client
enable	true to enable advertising, false to disable advertising

i) Data Types and Constants

DRV_BM64_BUFFER_EVENT Macro

File

drv_bm64.h

C

```
#define DRV_BM64_BUFFER_EVENT DRV_I2S_BUFFER_EVENT
```

Description

This is macro DRV_BM64_BUFFER_EVENT.

DRV_BM64_BUFFER_EVENT_COMPLETE Macro

File

drv_bm64.h

C

```
#define DRV_BM64_BUFFER_EVENT_COMPLETE DRV_I2S_BUFFER_EVENT_COMPLETE
```

Description

This is macro DRV_BM64_BUFFER_EVENT_COMPLETE.

DRV_BM64_BUFFER_HANDLE Macro

File

drv_bm64.h

C

```
#define DRV_BM64_BUFFER_HANDLE DRV_I2S_BUFFER_HANDLE
```

Description

This is macro DRV_BM64_BUFFER_HANDLE.

DRV_BM64_BUFFER_HANDLE_INVALID Macro

File

drv_bm64.h

C

```
#define DRV_BM64_BUFFER_HANDLE_INVALID DRV_I2S_BUFFER_HANDLE_INVALID
```

Description

This is macro DRV_BM64_BUFFER_HANDLE_INVALID.

DRV_BM64_DATA32 Macro

File

drv_bm64.h

C

```
#define DRV_BM64_DATA32 DRV_I2S_DATA32
```

Description

BM64 defines based on I2S interface

DRV_BM64_MAXBDNAMESIZE Macro

File

drv_bm64.h

C

```
#define DRV_BM64_MAXBDNAMESIZE 32
```

Section

Constants

DRV_BM64_BUFFER_EVENT_HANDLER Type

File

drv_bm64.h

C

```
typedef void (* DRV_BM64_BUFFER_EVENT_HANDLER)(DRV_BM64_BUFFER_EVENT event, uintptr_t contextHandle);
```

Description

prototype for callback for DRV_BM64_BufferEventHandlerSet

DRV_BM64_DRVR_STATUS Enumeration

File

drv_bm64.h

C

```
typedef enum {
  DRV_BM64_STATUS_NONE,
  DRV_BM64_STATUS_OFF,
  DRV_BM64_STATUS_ON,
  DRV_BM64_STATUS_READY
} DRV_BM64_DRVR_STATUS;
```

Description

BM64 driver status

DRV_BM64_EVENT Enumeration

File

drv_bm64.h

C

```
typedef enum {
  DRV_BM64_EVENT_NONE = 0,
  DRV_BM64_EVENT_NSPK_STATUS,
  DRV_BM64_EVENT_LINE_IN_STATUS,
  DRV_BM64_EVENT_A2DP_STATUS,
  DRV_BM64_EVENT_CALL_STATUS_CHANGED,
  DRV_BM64_EVENT_CODEC_TYPE,
  DRV_BM64_EVENT_HFP_CONNECTED,
  DRV_BM64_EVENT_HFP_DISCONNECTED,
  DRV_BM64_EVENT_A2DP_CONNECTED,
  DRV_BM64_EVENT_A2DP_DISCONNECTED,
  DRV_BM64_EVENT_AVRCP_CONNECTED,
```

```
    DRV_BM64_EVENT_AVRCP_DISCONNECTED,
    DRV_BM64_EVENT_SPP_CONNECTED,
    DRV_BM64_EVENT_IAP_CONNETED,
    DRV_BM64_EVENT_SPP_IAP_DISCONNECTED,
    DRV_BM64_EVENT_ACL_CONNECTED,
    DRV_BM64_EVENT_ACL_DISCONNECTED,
    DRV_BM64_EVENT_SCO_CONNECTED,
    DRV_BM64_EVENT_SCO_DISCONNECTED,
    DRV_BM64_EVENT_MAP_CONNECTED,
    DRV_BM64_EVENT_MAP_DISCONNECTED,
    DRV_BM64_EVENT_SYS_POWER_ON,
    DRV_BM64_EVENT_SYS_POWER_OFF,
    DRV_BM64_EVENT_SYS_STANDBY,
    DRV_BM64_EVENT_SYS_PAIRING_START,
    DRV_BM64_EVENT_SYS_PAIRING_OK,
    DRV_BM64_EVENT_SYS_PAIRING_FAILED,
    DRV_BM64_EVENT_LINKBACK_SUCCESS,
    DRV_BM64_EVENT_LINKBACK_FAILED,
    DRV_BM64_EVENT_BD_ADDR_RECEIVED,
    DRV_BM64_EVENT_PAIR_RECORD_RECEIVED,
    DRV_BM64_EVENT_LINK_MODE_RECEIVED,
    DRV_BM64_EVENT_PLAYBACK_STATUS_CHANGED,
    DRV_BM64_EVENT_AVRCP_VOLUME_CTRL,
    DRV_BM64_EVENT_AVRCP_ABS_VOLUME_CHANGED,
    DRV_BM64_EVENT_HFP_VOLUME_CHANGED,
    DRV_BM64_EVENT_VOLUME_CHANGED,
    DRV_BM64_EVENT_SAMPLERATE_CHANGED,
    DRV_BM64_EVENT_NSPK_SYNC_POWER_OFF,
    DRV_BM64_EVENT_NSPK_SYNC_VOL_CTRL,
    DRV_BM64_EVENT_NSPK_SYNC_INTERNAL_GAIN,
    DRV_BM64_EVENT_NSPK_SYNC_ABS_VOL,
    DRV_BM64_EVENT_NSPK_CHANNEL_SETTING,
    DRV_BM64_EVENT_NSPK_ADD_SPEAKER3,
    DRV_BM64_EVENT_LE_STATUS_CHANGED,
    DRV_BM64_EVENT_LE_ADV_CONTROL_REPORT,
    DRV_BM64_EVENT_LE_CONNECTION_PARA_REPORT,
    DRV_BM64_EVENT_LE_CONNECTION_PARA_UPDATE_RSP,
    DRV_BM64_EVENT_GATT_ATTRIBUTE_DATA,
    DRV_BM64_EVENT_PORT0_INPUT_CHANGED,
    DRV_BM64_EVENT_PORT1_INPUT_CHANGED,
    DRV_BM64_EVENT_PORT2_INPUT_CHANGED,
    DRV_BM64_EVENT_PORT3_INPUT_CHANGED,
    DRV_BM64_EVENT_BLESPP_MSG_RECEIVED,
    DRV_BM64_EVENT_BLE_STATUS_CHANGED
} DRV_BM64_EVENT;
```

Description

events that can be returned to a client via callback

DRV_BM64_EVENT_HANDLER Type

File

drv_bm64.h

C

```
typedef void (* DRV_BM64_EVENT_HANDLER)(DRV_BM64_EVENT event, uint32_t param, uintptr_t contextHandle);
```

Description

prototype for callback for DRV_BM64_EventHandlerSet

DRV_BM64_LINKSTATUS Enumeration

File

drv_bm64.h

C

```
typedef enum {
  DRV_BM64_NO_LINK_STATUS = 0,
  DRV_BM64_SCO_LINK_STATUS = 0x01,
```

```
  DRV_BM64_ACL_LINK_STATUS = 0x02,
  DRV_BM64_HFP_LINK_STATUS = 0x04,
  DRV_BM64_A2DP_LINK_STATUS = 0x08,
  DRV_BM64_AVRCP_LINK_STATUS = 0x10,
  DRV_BM64_SPP_LINK_STATUS = 0x20,
  DRV_BM64_IAP_LINK_STATUS = 0x40,
  DRV_BM64_MAP_LINK_STATUS = 0x80
} DRV_BM64_LINKSTATUS;
```

Description

BM64 link status

DRV_BM64_PLAYINGSTATUS Enumeration

File

drv_bm64.h

C

```
typedef enum {
  DRV_BM64_PLAYING_STOPPED,
  DRV_BM64_PLAYING_PLAYING,
  DRV_BM64_PLAYING_PAUSED,
  DRV_BM64_PLAYING_FF,
  DRV_BM64_PLAYING_FR,
  DRV_BM64_PLAYING_ERROR
} DRV_BM64_PLAYINGSTATUS;
```

Description

This is type DRV_BM64_PLAYINGSTATUS.

DRV_BM64_PROTOCOL Enumeration

File

drv_bm64.h

C

```
typedef enum {
  DRV_BM64_PROTOCOL_A2DP = 1,
  DRV_BM64_PROTOCOL_AVRCP = 2,
  DRV_BM64_PROTOCOL_HFP_HSP = 4,
  DRV_BM64_PROTOCOL_SPP = 8,
  DRV_BM64_PROTOCOL_BLE = 16,
  DRV_BM64_PROTOCOL_ALL = 31
} DRV_BM64_PROTOCOL;
```

Description

BM64 protocols

DRV_BM64_REQUEST Enumeration

File

drv_bm64.h

C

```
typedef enum {
  DRV_BM64_REQ_NONE = 0,
  DRV_BM64_REQ_SYSTEM_ON,
  DRV_BM64_REQ_SYSTEM_OFF
} DRV_BM64_REQUEST;
```

Description

BM64 power on/off request

DRV_BM64_SAMPLE_FREQUENCY Enumeration

File

drv_bm64.h

C

```
typedef enum {
  DRV_BM64_SAMPLEFREQ_8000 = 0,
  DRV_BM64_SAMPLEFREQ_12000,
  DRV_BM64_SAMPLEFREQ_16000,
  DRV_BM64_SAMPLEFREQ_24000,
  DRV_BM64_SAMPLEFREQ_32000,
  DRV_BM64_SAMPLEFREQ_48000,
  DRV_BM64_SAMPLEFREQ_44100,
  DRV_BM64_SAMPLEFREQ_88000,
  DRV_BM64_SAMPLEFREQ_96000
} DRV_BM64_SAMPLE_FREQUENCY;
```

Description

BM64 sample frequency

DRV_BM64_BLE_STATUS Enumeration

File

drv_bm64.h

C

```
typedef enum {
  DRV_BM64_BLE_STATUS_STANDBY,
  DRV_BM64_BLE_STATUS_ADVERTISING,
  DRV_BM64_BLE_STATUS_SCANNING,
  DRV_BM64_BLE_STATUS_CONNECTED
} DRV_BM64_BLE_STATUS;
```

Description

This is type DRV_BM64_BLE_STATUS.

Files

Files

Name	Description
drv_bm64.h	BM64 Bluetooth Static Driver main header file
drv_bm64_config_template.h	BM64 Bluetooth Driver Configuration Template.

Description

This section lists the source and header files used by the BM64 Bluetooth Driver Library.

drv_bm64.h

BM64 Bluetooth Static Driver main header file

Enumerations

	Name	Description
	DRV_BM64_BLE_STATUS	This is type DRV_BM64_BLE_STATUS.
	DRV_BM64_DRVR_STATUS	BM64 driver status
	DRV_BM64_EVENT	events that can be returned to a client via callback
	DRV_BM64_LINKSTATUS	BM64 link status
	DRV_BM64_PLAYINGSTATUS	This is type DRV_BM64_PLAYINGSTATUS.
	DRV_BM64_PROTOCOL	BM64 protocols
	DRV_BM64_REQUEST	BM64 power on/off request

	DRV_BM64_SAMPLE_FREQUENCY	BM64 sample frequency

Functions

	Name	Description
	DRV_BM64_BLE_EnableAdvertising	Enable or disable advertising.
	DRV_BM64_BLE_QueryStatus	Query BM64 LE status.
	DRV_BM64_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_BM64_BufferEventHandlerSet	This function allows a client to identify a event handling function for the driver to call back.
	DRV_BM64_CancelForwardOrRewind	Cancel previous fast forward or rewind request.
	DRV_BM64_ClearBLEData	Clear the BLE receive buffer.
	DRV_BM64_Close	Close an opened-instance of the BM64 Bluetooth driver.
	DRV_BM64_DisconnectAllLinks	Disconnect all links.
	DRV_BM64_EnterBTPairingMode	Enter Bluetooth pairing mode.
	DRV_BM64_EventHandlerSet	This function allows a client to identify an event handling function for the driver to call back.
	DRV_BM64_FastForward	Fast forward the media.
	DRV_BM64_ForgetAllLinks	Forget all pairings.
	DRV_BM64_GetBDAddress	Return the Bluetooth address.
	DRV_BM64_GetBDName	Return Bluetooth device name.
	DRV_BM64_GetLinkStatus	Return link status.
	DRV_BM64_GetPlayingStatus	Return the current playing status of the device.
	DRV_BM64_GetPowerStatus	Gets the current status of the BM64 Bluetooth driver module (BM64-specific).
	DRV_BM64_Initialize	Initializes hardware and data for the instance of the BM64 Bluetooth module
	DRV_BM64_LinkLastDevice	Link last device.
	DRV_BM64_Open	Open the specified BM64 driver instance and returns a handle to it
	DRV_BM64_Pause	Pause playback.
	DRV_BM64_Play	Start playback.
	DRV_BM64_PlayNextSong	Play the next song.
	DRV_BM64_PlayPause	Toggle play/pause mode.
	DRV_BM64_PlayPreviousSong	Play the previous song.
	DRV_BM64_ReadByteFromBLE	Read a byte over BLE.
	DRV_BM64_ReadDataFromBLE	Read data over BLE.
	DRV_BM64_Rewind	Rewind the media.
	DRV_BM64_SamplingRateGet	Return the current sampling rate.
	DRV_BM64_SamplingRateSet	Set the current sampling rate.
	DRV_BM64_SendByteOverBLE	Send a byte over BLE.
	DRV_BM64_SendDataOverBLE	Send data over BLE.
	DRV_BM64_SetBDName	Set the Bluetooth device name.
	DRV_BM64_Status	Gets the current system status of the BM64 Bluetooth driver module.
	DRV_BM64_Stop	Stop playback.
	DRV_BM64_TaskReq	Make a power on/power off task request.
	DRV_BM64_Tasks	Maintains the driver's control and data interface state machine.
	DRV_BM64_volumeDown	Turn the volume down on the host device.
	DRV_BM64_VolumeGet	returns 7-bit value 0-127
	DRV_BM64_VolumeSet	Set current volume.
	DRV_BM64_volumeUp	Turn the volume up on the host device.

Macros

	Name	Description
	DRV_BM64_BUFFER_EVENT	This is macro DRV_BM64_BUFFER_EVENT.
	DRV_BM64_BUFFER_EVENT_COMPLETE	This is macro DRV_BM64_BUFFER_EVENT_COMPLETE.
	DRV_BM64_BUFFER_HANDLE	This is macro DRV_BM64_BUFFER_HANDLE.
	DRV_BM64_BUFFER_HANDLE_INVALID	This is macro DRV_BM64_BUFFER_HANDLE_INVALID.
	DRV_BM64_DATA32	BM64 defines based on I2S interface
	DRV_BM64_MAXBDNAMESIZE	

Types

	Name	Description
	DRV_BM64_BUFFER_EVENT_HANDLER	prototype for callback for DRV_BM64_BufferEventHandlerSet
	DRV_BM64_EVENT_HANDLER	prototype for callback for DRV_BM64_EventHandlerSet

Description

BM64 Bluetooth Static Driver implementation

This file is the header file for the external (public) API of the static implementation of the BM64 driver.

The BM64 is a Bluetooth 4.2 Stereo Module that supports classic A2DP, AVRCP, HFP, HSP, and SPP protocols as well as BLE (Bluetooth Low Energy).

The BM64 streams I2S audio at up to 24-bit, 96 kHz. It uses a UART to receive commands from the host microcontroller (PIC32) and and send events back.

All functions and constants in this file are named with the format DRV_BM64_xxx, where xxx is a function name or constant. These names are redefined in the appropriate configuration?s system_config.h file to the format DRV_BT_xxx using #defines so that Bluetooth code in the application can be written as generically as possible (e.g. by writing DRV_BT_Open instead of DRV_BM64_Open etc.).

File Name

drv_bm64.h

Company

Microchip Technology Inc.

drv_bm64_config_template.h

BM64 Bluetooth Driver Configuration Template.

Macros

	Name	Description
	INCLUDE_BM64_BLE	Identifies whether the driver should include BLE
	INCLUDE_BM64_I2S	Identifies whether the driver should include HFP,A2DP,AVRCP functionality.
	INCLUDE_DEPRECATED_MMI_COMMANDS	Identifies whether the driver should use deprecated MMI commands.

Description

BM64 Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_bm64_config_template.h

Company

Microchip Technology Inc.

Camera Driver Libraries

This section describes the Camera Driver Libraries.

Introduction

This section provides information on the Camera Driver libraries that are provided in MPLAB Harmony and describes the APIs that are common to all drivers.

Library Interface

a) Common Driver Functions

	Name	Description
	DRV_CAMERA_Close	Closes an opened instance of an CAMERA module driver.
	DRV_CAMERA_Deinitialize	Deinitializes the index instance of the CAMERA module.
	DRV_CAMERA_Initialize	Initializes hardware and data for the index instance of the CAMERA module.

	DRV_CAMERA_Open	Opens the specified instance of the Camera driver for use and provides an "open instance" handle.
	DRV_CAMERA_Reinitialize	Reinitializes hardware and data for the index instance of the CAMERA module.
	DRV_CAMERA_Status	Provides the current status of the index instance of the CAMERA module.
	DRV_CAMERA_Tasks	This is function DRV_CAMERA_Tasks.

b) Common Data Types and Constants

	Name	Description
	DRV_CAMERA_INIT	Defines the data required to initialize or reinitialize the CAMERA driver.
	DRV_CAMERA_INTERRUPT_PORT_REMAP	Defines the data required to initialize the CAMERA driver interrupt port remap.
	DRV_CAMERA_INDEX_0	Camera driver index definitions.
	DRV_CAMERA_INDEX_1	This is macro DRV_CAMERA_INDEX_1.
	DRV_CAMERA_INDEX_COUNT	Number of valid CAMERA driver indices.
	CAMERA_MODULE_ID	This is type CAMERA_MODULE_ID.

Description

Camera Driver APIs that are common to all Camera drivers.

a) Common Driver Functions

DRV_CAMERA_Close Function

Closes an opened instance of an CAMERA module driver.

File

drv_camera.h

C

```
void DRV_CAMERA_Close(DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened instance of an CAMERA module driver, making the specified handle invalid.

Remarks

None.

Preconditions

The DRV_CAMERA_Initialize routine must have been called for the specified CAMERA device instance and the DRV_CAMERA_Status must have returned SYS_STATUS_READY.

DRV_CAMERA_Open must have been called to obtain a valid opened device handle.

Example

```
myCameraHandle = DRV_CAMERA_Open(DRV_CAMERA_ID_1, DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE);

DRV_CAMERA_Close(myCameraHandle);
```

Parameters

Parameters	Description
drvHandle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_CAMERA_Close (const DRV_HANDLE drvHandle)

DRV_CAMERA_Deinitialize Function

Deinitializes the index instance of the CAMERA module.

File

drv_camera.h

C

```
void DRV_CAMERA_Deinitialize(const SYS_MODULE_INDEX index);
```

Returns

None.

Description

This function deinitializes the index instance of the CAMERA module, disabling its operation (and any hardware for driver modules). It deinitializes only the specified module instance. It also resets all the internal data structures and fields for the specified instance to the default settings.

Remarks

None.

Preconditions

The DRV_CAMERA_Initialize function should have been called before calling this function.

Example

```
SYS_STATUS   cameraStatus;

DRV_CAMERA_Deinitialize(DRV_CAMERA_ID_1);

cameraStatus = DRV_CAMERA_Status(DRV_CAMERA_ID_1);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the CAMERA module to be deinitialized

Function

void DRV_CAMERA_Deinitialize (const SYS_MODULE_ID index)

DRV_CAMERA_Initialize Function

Initializes hardware and data for the index instance of the CAMERA module.

File

drv_camera.h

C

```
SYS_MODULE_OBJ DRV_CAMERA_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

None.

Description

This function initializes hardware for the index instance of the CAMERA module, using the hardware initialization given data. It also initializes any internal driver data structures making the driver ready to be opened.

Remarks

None.

Preconditions

None.

Example

```
DRV_CAMERA_INIT_DATA        cameraInitData;
SYS_STATUS                  cameraStatus;

// Populate the cameraInitData structure
cameraInitData.moduleInit.powerState = SYS_MODULE_POWER_RUN_FULL;
```

```
cameraInitData.moduleInit.moduleCode = (DRV_CAMERA_INIT_DATA_MASTER | DRV_CAMERA_INIT_DATA_SLAVE);

DRV_CAMERA_Initialize(DRV_CAMERA_ID_1, (SYS_MODULE_INIT*)&cameraInitData);
cameraStatus = DRV_CAMERA_Status(DRV_CAMERA_ID_1);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the CAMERA module to be initialized
data	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and the default initialization is to be used.

Function

void DRV_CAMERA_Initialize (const CAMERA_MODULE_ID index,

const SYS_MODULE_INIT *const data)

DRV_CAMERA_Open Function

Opens the specified instance of the Camera driver for use and provides an "open instance" handle.

File

drv_camera.h

C

```
DRV_HANDLE DRV_CAMERA_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Returns

If successful, the routine returns a valid open-instance handle (a value identifying both the caller and the module instance). If an error occurs, the returned value is DRV_HANDLE_INVALID.

Description

This function opens the specified instance of the Camera module for use and provides a handle that is required to use the remaining driver routines.

This function opens a specified instance of the Camera module driver for use by any client module and provides an "open instance" handle that must be provided to any of the other Camera driver operations to identify the caller and the instance of the Camera driver/hardware module.

Preconditions

The DRV_CAMERA_Initialize routine must have been called for the specified CAMERA device instance and the DRV_CAMERA_Status must have returned SYS_STATUS_READY.

Example

```
DRV_HANDLE                cameraHandle;
DRV_CAMERA_CLIENT_STATUS   cameraClientStatus;

cameraHandle = DRV_CAMERA_Open(DRV_CAMERA_ID_1, DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE);
if (DRV_HANDLE_INVALID == cameraHandle)
{
    // Handle open error
}

cameraClientStatus = DRV_CAMERA_ClientStatus(cameraHandle);

// Close the device when it is no longer needed.
DRV_CAMERA_Close(cameraHandle);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the CAMERA module to be opened.
intent	Flags parameter identifying the intended usage and behavior of the driver. Multiple flags may be ORed together to specify the intended usage of the device. See the DRV_IO_INTENT definition.

Function

DRV_HANDLE DRV_CAMERA_Open (const SYS_MODULE_INDEX index,

const DRV_IO_INTENT intent)

DRV_CAMERA_Reinitialize Function

Reinitializes hardware and data for the index instance of the CAMERA module.

File

drv_camera.h

C

```
void DRV_CAMERA_Reinitialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT *const data);
```

Returns

None.

Description

This function reinitializes hardware for the index instance of the CAMERA module, using the hardware initialization given data. It also reinitializes any internal driver data structures making the driver ready to be opened.

Remarks

None.

Preconditions

The DRV_CAMERA_Initialize function should have been called before calling this function.

Example

```
SYS_MODULE_INIT cameraInit;
SYS_STATUS      cameraStatus;

DRV_CAMERA_Reinitialize(DRV_CAMERA_ID_1, &cameraStatus);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the CAMERA module to be reinitialized
data	Pointer to the data structure containing any data necessary to reinitialize the hardware. This pointer may be null if no data is required and default configuration is to be used.

Function

void DRV_CAMERA_Reinitialize(const SYS_MODULE_ID index,

const SYS_MODULE_INIT *const data)

DRV_CAMERA_Status Function

Provides the current status of the index instance of the CAMERA module.

File

drv_camera.h

C

```
SYS_STATUS DRV_CAMERA_Status(const SYS_MODULE_INDEX index);
```

Returns

The current status of the index instance.

Description

This function provides the current status of the index instance of the CAMERA module.

Remarks

None.

Preconditions

The DRV_CAMERA_Initialize function should have been called before calling this function.

Function

SYS_STATUS DRV_CAMERA_Status (const CAMERA_MODULE_ID index)

DRV_CAMERA_Tasks Function

File

drv_camera.h

C

```
void DRV_CAMERA_Tasks(SYS_MODULE_OBJ object);
```

Description

This is function DRV_CAMERA_Tasks.

b) Common Data Types and Constants

DRV_CAMERA_INIT Structure

Defines the data required to initialize or reinitialize the CAMERA driver.

File

drv_camera.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  int cameraId;
  SYS_MODULE_OBJ (* drvInitialize)(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
  DRV_HANDLE (* drvOpen)(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
  INT_SOURCE interruptSource;
  DRV_CAMERA_INTERRUPT_PORT_REMAP interruptPort;
  uint16_t orientation;
  uint16_t horizontalResolution;
  uint16_t verticalResolution;
} DRV_CAMERA_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
int cameraId;	ID
uint16_t orientation;	Orientation of the display (given in degrees of 0,90,180,270)
uint16_t horizontalResolution;	Horizontal Resolution of the displayed orientation in Pixels

Description

CAMERA Driver Initialization Data

This data type defines the data required to initialize or reinitialize the CAMERA driver. If the driver is built statically, the members of this data structure are statically over-ridden by static override definitions in the system_config.h file.

Remarks

None.

DRV_CAMERA_INTERRUPT_PORT_REMAP Structure

Defines the data required to initialize the CAMERA driver interrupt port remap.

File

drv_camera.h

C

```
typedef struct {
```

```
    PORTS_REMAP_INPUT_FUNCTION inputFunction;
    PORTS_REMAP_INPUT_PIN inputPin;
    PORTS_ANALOG_PIN analogPin;
    PORTS_PIN_MODE pinMode;
    PORTS_CHANNEL channel;
    PORTS_DATA_MASK dataMask;
} DRV_CAMERA_INTERRUPT_PORT_REMAP;
```

Description

CAMERA Driver Interrupt Port Remap Initialization Data

This data type defines the data required to initialize the CAMERA driver interrupt port remap.

Remarks

None.

DRV_CAMERA_INDEX_0 Macro

Camera driver index definitions.

File

drv_camera.h

C

```
#define DRV_CAMERA_INDEX_0 0
```

Description

Camera Driver Module Index Numbers

These constants provide the Camera driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_CAMERA_Initialize and DRV_CAMERA_Open functions to identify the driver instance in use.

DRV_CAMERA_INDEX_1 Macro

File

drv_camera.h

C

```
#define DRV_CAMERA_INDEX_1 1
```

Description

This is macro DRV_CAMERA_INDEX_1.

DRV_CAMERA_INDEX_COUNT Macro

Number of valid CAMERA driver indices.

File

drv_camera.h

C

```
#define DRV_CAMERA_INDEX_COUNT 1
```

Description

CAMERA Driver Module Index Count

This constant identifies the number of valid CAMERA driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific header files defined as part of the peripheral libraries.

CAMERA_MODULE_ID Enumeration

File

drv_camera.h

C

```
typedef enum {
  CAMERA_MODULE_OVM7690
} CAMERA_MODULE_ID;
```

Description

This is type CAMERA_MODULE_ID.

Files

Files

Name	Description
drv_camera.h	Camera device driver interface file.

Description

drv_camera.h

Camera device driver interface file.

Enumerations

	Name	Description
	CAMERA_MODULE_ID	This is type CAMERA_MODULE_ID.

Functions

	Name	Description
	DRV_CAMERA_Close	Closes an opened instance of an CAMERA module driver.
	DRV_CAMERA_Deinitialize	Deinitializes the index instance of the CAMERA module.
	DRV_CAMERA_Initialize	Initializes hardware and data for the index instance of the CAMERA module.
	DRV_CAMERA_Open	Opens the specified instance of the Camera driver for use and provides an "open instance" handle.
	DRV_CAMERA_Reinitialize	Reinitializes hardware and data for the index instance of the CAMERA module.
	DRV_CAMERA_Status	Provides the current status of the index instance of the CAMERA module.
	DRV_CAMERA_Tasks	This is function DRV_CAMERA_Tasks.

Macros

	Name	Description
	DRV_CAMERA_INDEX_0	Camera driver index definitions.
	DRV_CAMERA_INDEX_1	This is macro DRV_CAMERA_INDEX_1.
	DRV_CAMERA_INDEX_COUNT	Number of valid CAMERA driver indices.

Structures

	Name	Description
	DRV_CAMERA_INIT	Defines the data required to initialize or reinitialize the CAMERA driver.
	DRV_CAMERA_INTERRUPT_PORT_REMAP	Defines the data required to initialize the CAMERA driver interrupt port remap.

Description

Camera Driver Interface

The Camera driver provides a abstraction to all camera drivers.

File Name

drv_camera.h

Company

Microchip Technology Inc.

OVM7690 Camera Driver Library

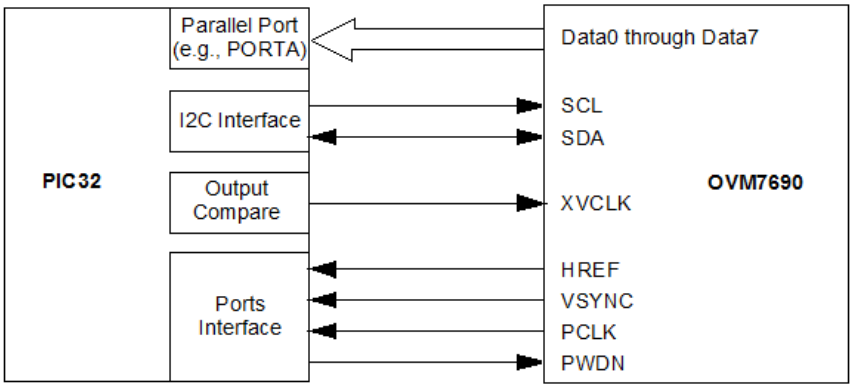
This topic describes the OVM7690 Camera Driver Library.

Introduction

The OVM7690 Camera Driver provides a high-level interface to manage the OmniVision Technologies, Inc. OVM7690 640x480 CameraCube™ device (referred to as the OVM7690) that is interfaced with serial and parallel ports to a Microchip microcontroller for providing camera solutions.

Description

The OVM7690 640x480 CameraCube™ device (referred to as the OVM7690) can be interfaced to a Microchip microcontroller using the I2C serial interface and parallel port interface. The I2C serial interface is used for control command transfer. The I2C module from the microcontroller is connected to the SCCB serial interface of the OVM7690. The parallel port interface is used to transfer pixel data from the OVM7690 to the microcontroller. There are few other signals from the camera to be interfaced with the microcontroller. The XVCLK pin of the camera is driven by the Output Compare module. Frame synchronization signals such as HREF and VSYNC from the camera are connected to suitable pins supporting change notification within the microcontroller. The PCLK pin of the camera drives the pixel clock and is connected at the pin of the microcontroller supporting external interrupts. The PWDN pin of the camera supports camera power-down mode and is connected at any output port pin of the microcontroller. A typical interface of the OVM7690 to a PIC32 device is provided in the following diagram:



Using the Library

This topic describes the basic architecture of the OVM7690 Camera Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_camera_ovm7690.h`

The interface to the Camera Driver Library is defined in the `drv_camera_ovm7690.h` header file.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address the overall operation of the OVM7690 Camera Driver Library.

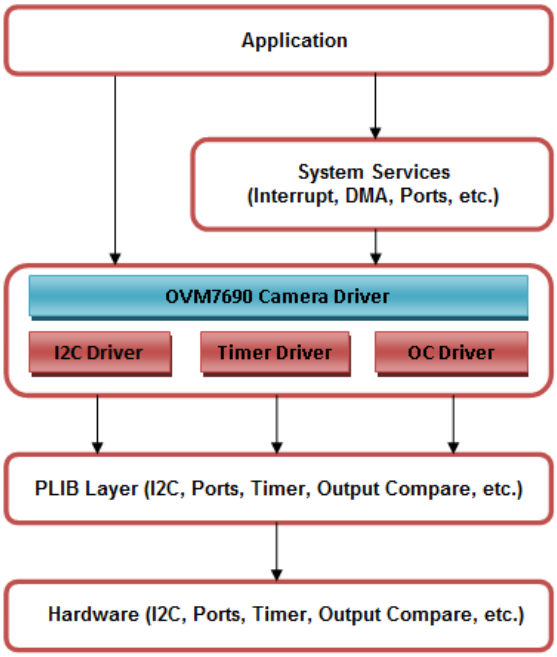
Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization and deinitialization.
Client Setup Functions	Provides open and close functions.
Camera-specific Functions	Provides APIs that are camera-specific.
Other Functions	Provides miscellaneous driver-specific functions such as register set functions, among others.

Abstraction Model

This library provides a low-level abstraction of the OVM7690 Camera Driver Library on Microchip's microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The OVM7690 Camera Driver is modeled using the abstraction model, as shown in the following diagram.



How the Library Works

Provides information on how the OVM7690 Camera Driver Library works.

Description

The library provides interfaces to support:

- System functionality
- Client functionality

System Initialization

The system performs the Initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the OVM7690 would be initialized with the following configuration settings that are supported by the specific OVM7690 device hardware:

- Camera ID: OVM7690 ID
- Source Port: Address of source port to which the pixel data is received
- Horizontal Sync Channel: Channel of the pin to be configured as horizontal sync
- Horizontal Sync Position: Horizontal sync port pin position from selected port channel
- Vertical Sync Channel: Channel the pin to be configured as vertical sync
- Vertical Sync Position: Vertical sync port pin position from selected port channel
- Horizontal Sync Interrupt Source
- Vertical Sync Interrupt Source
- DMA Channel: DMA channel to transfer pixel data from camera to frame buffer
- DMA Channel Trigger Source
- Bits Per Pixel: Bits per pixel to define the size of frame line

The DRV_CAMERA_OVM7690_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handler returned by the Initialize Interface would be used by the other interfaces such as DRV_CAMERA_OVM7690_Deinitialize.

Client Access

For the application to start using an instance of the module, it must call the DRV_CAMERA_OVM7690_Open function. The DRV_CAMERA_OVM7690_Open function provides a driver handle to the OVM7690 Camera Driver instance for operations. If the driver is deinitialized using the function DRV_CAMERA_OVM7690_Deinitialize function, the application must call the DRV_CAMERA_OVM7690_Open function again to set up the instance of the driver.

Client Operations

Client operations provide the API interface for control command and pixel data transfer from the OVM7690 Camera Driver to the Graphics Frame Buffer.

Configuring the Library

Macros

	Name	Description
	DRV_OVM7690_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.

Description

The configuration of the OVM7690 Camera Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the OVM7690 Camera Driver build. Based on the selections made here and the system setup, the OVM7690 Camera Driver may support the selected features. These configuration settings will apply to all instances of the driver.

This header can be placed anywhere in the application specific folders and the path of this header needs to be presented to the include search for a successful build. Refer to the Applications Help section for more details.

Control Commands

The following OVM7690-specific control commands are provided:

- DRV_CAMERA_OVM7690_FrameBufferAddressSet
- DRV_CAMERA_OVM7690_Start
- DRV_CAMERA_OVM7690_Stop
- DRV_CAMERA_OVM7690_FrameRectSet

Application Process

An application needs to perform following steps:

1. The system should have completed necessary setup initializations.
2. The I2C driver object should have been initialized by calling DRV_I2C_Initialize.
3. The Timer driver object should have been initialized by calling DRV_Timer_Initialize,
4. The Output Control driver object should have been initialized by calling DRV_OC_Initialize,
5. The OVM7690 Camera Driver object should have been initialized by calling DRV_CAMERA_OVM7690_Initialize,
6. Open the OVM7690 Camera Driver client by calling DRV_CAMERA_OVM7690_Open.
7. Pass the Graphics Frame buffer address to OVM7690 Camera Driver by calling DRV_CAMERA_OVM7690_FrameBufferAddressSet.
8. Set the Frame Rectangle area by calling DRV_CAMERA_OVM7690_FrameRectSet.
9. Set Other Camera settings such as: soft reset, enabling pclk, enabling href, enabling vsync, output color format, reversing HREF polarity, gating clock to the HREF, pixel clock frequency, sub-sampling mode by calling DRV_CAMERA_OVM7690_RegisterSet.
10. Start the OVM7690 Camera by calling DRV_CAMERA_OVM7690_Start.

DRV_OVM7690_INTERRUPT_MODE Macro

Controls operation of the driver in the interrupt or polled mode.

File

drv_ovm7690_config_template.h

C

```
#define DRV_OVM7690_INTERRUPT_MODE false
```

Description

OVM7690 Interrupt And Polled Mode Operation Control

This macro controls the operation of the driver in the interrupt mode of operation. The possible values of this macro are:

- true - Select if interrupt mode of OVM7690 operation is desired

- false - Select if polling mode of OVM7690 operation is desired

Not defining this option to true or false will result in a build error.

Remarks

None.

Building the Library

This section lists the files that are available in the OVM7690 Camera Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/camera/ovm7690`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_camera_ovm7690.h`	This file provides the interface definitions of the OVM7690 Camera Driver.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/drv_camera_ovm7690.c`	This file contains the implementation of the OVM7690 Camera Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The OVM7690 Camera Driver Library depends on the following modules:

- I2C Driver Library
- Output Compare Driver Library
- Timer Driver Library

Library Interface

a) System Functions

	Name	Description
	DRV_CAMERA_OVM7690_Initialize	Initializes the OVM7690 Camera instance for the specified driver index.
	DRV_CAMERA_OVM7690_Deinitialize	Deinitializes the specified instance of the OVM7690 Camera Driver module.
	DRV_CAMERA_OVM7690_RegisterSet	Sets the camera OVM7690 configuration registers.
	DRV_CAMERA_OVM7690_Tasks	Maintains the OVM7690 state machine.

b) Client Setup Functions

	Name	Description
	DRV_CAMERA_OVM7690_Open	Opens the specified OVM7690 Camera Driver instance and returns a handle to it.
	DRV_CAMERA_OVM7690_Close	Closes an opened instance of the OVM7690 Camera Driver.

c) Camera-specific Functions

	Name	Description
	DRV_CAMERA_OVM7690_FrameBufferAddressSet	Sets the framebuffer address.

	DRV_CAMERA_OVM7690_FrameRectSet	Sets the frame rectangle set.
	DRV_CAMERA_OVM7690_Start	Starts camera rendering to the display.
	DRV_CAMERA_OVM7690_Stop	Stops rendering the camera Pixel data.

d) Other Functions

	Name	Description
	DRV_CAMERA_OVM7690_HsyncEventHandler	Horizontal synchronization event handler.
	DRV_CAMERA_OVM7690_VsyncEventHandler	Vertical synchronization event handler .
	_DRV_CAMERA_OVM7690_DMAEventHandler	This is function _DRV_CAMERA_OVM7690_DMAEventHandler.
	_DRV_CAMERA_OVM7690_delayMS	This is function _DRV_CAMERA_OVM7690_delayMS.
	_DRV_CAMERA_OVM7690_HardwareSetup	This is function _DRV_CAMERA_OVM7690_HardwareSetup.

e) Data Types and Constants

	Name	Description
	DRV_CAMERA_OVM7690_CLIENT_OBJ	OVM7690 Camera Driver client object.
	DRV_CAMERA_OVM7690_CLIENT_STATUS	Identifies OVM7690 Camera possible client status.
	DRV_CAMERA_OVM7690_ERROR	Identifies OVM7690 Camera possible errors.
	DRV_CAMERA_OVM7690_INIT	OVM7690 Camera Driver initialization parameters.
	DRV_CAMERA_OVM7690_OBJ	OVM7690 Camera Driver instance object.
	DRV_CAMERA_OVM7690_RECT	OVM7690 Camera window rectangle coordinates.
	DRV_CAMERA_OVM7690_REG12_OP_FORMAT	Lists OVM7690 Camera device register addresses.
	DRV_CAMERA_OVM7690_INDEX_0	OVM7690 driver index definitions.
	DRV_CAMERA_OVM7690_INDEX_1	This is macro DRV_CAMERA_OVM7690_INDEX_1.
	DRV_CAMERA_OVM7690_REG12_SOFT_RESET	OVM7690 Camera Driver Register 0x12 Soft reset flag.
	DRV_CAMERA_OVM7690_SCCB_READ_ID	OVM7690 Camera SCCB Interface device Read Slave ID.
	DRV_CAMERA_OVM7690_SCCB_WRITE_ID	OVM7690 Camera SCCB Interface device Write Slave ID.

Description

This section describes the Application Programming Interface (API) functions of the Camera Driver Library.

a) System Functions

DRV_CAMERA_OVM7690_Initialize Function

Initializes the OVM7690 Camera instance for the specified driver index.

File

drv_camera_ovm7690.h

C

```
SYS_MODULE_OBJ DRV_CAMERA_OVM7690_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes the OVM7690 Camera Driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the OVM7690 Camera module ID. Refer to the description of the DRV_CAMERA_OVM7690_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This function must be called before any other OVM7690 Camera Driver function is called.

This function should only be called once during system initialization unless DRV_CAMERA_OVM7690_Deinitialize is called to deinitialize the driver instance. This function will NEVER block for hardware access.

Preconditions

None.

Example

```
// The following code snippet shows an example OVM7690 driver initialization.

DRV_CAMERA_OVM7690_INIT     cameraInit;
SYS_MODULE_OBJ              objectHandle;

cameraInit.cameraID                = CAMERA_MODULE_OVM7690;
cameraInit.sourcePort              = (void *)&PORTK,
cameraInit.hsyncInterruptSource    = INT_SOURCE_CHANGE_NOTICE_A,
cameraInit.vsyncInterruptSource    = INT_SOURCE_CHANGE_NOTICE_J,
cameraInit.dmaChannel              = DRV_CAMERA_OVM7690_DMA_CHANNEL_INDEX,
cameraInit.dmaTriggerSource        = DMA_TRIGGER_EXTERNAL_2,
cameraInit.bpp                     = GFX_CONFIG_COLOR_DEPTH,

objectHandle = DRV_CAMERA_OVM7690_Initialize( DRV_CAMERA_OVM7690_INDEX_0,
                                              (SYS_MODULE_INIT*)&cameraInit);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_CAMERA_OVM7690_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

)

DRV_CAMERA_OVM7690_Deinitialize Function

Deinitializes the specified instance of the OVM7690 Camera Driver module.

File

drv_camera_ovm7690.h

C

```
void DRV_CAMERA_OVM7690_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the specified instance of the OVM7690 Camera Driver module, disabling its operation (and any hardware), and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This function will NEVER block waiting for hardware.

Preconditions

Function DRV_CAMERA_OVM7690_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object; //  Returned from DRV_CAMERA_OVM7690_Initialize
```

```
SYS_STATUS          status;

DRV_CAMERA_OVM7690_Deinitialize(object);

status = DRV_CAMERA_OVM7690_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_CAMERA_OVM7690_Initialize function

Function

void DRV_CAMERA_OVM7690_Deinitialize(SYS_MODULE_OBJ object)

DRV_CAMERA_OVM7690_RegisterSet Function

Sets the camera OVM7690 configuration registers.

File

drv_camera_ovm7690.h

C

```
DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_RegisterSet(DRV_CAMERA_OVM7690_REGISTER_ADDRESS regIndex,
uint8_t regValue);
```

Returns

- DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE - Invalid driver Handle.
- DRV_CAMERA_OVM7690_ERROR_NONE - No error.

Description

This function sets the OVM7690 Camera configuration registers using the SCCB interface.

Remarks

This function can be used separately or within an interface.

Preconditions

The DRV_CAMERA_OVM7690_Initialize function must have been called for the specified OVM7690 Camera Driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

The SCCB interface also must have been initialized to configure the OVM7690 Camera Driver.

Example

```
DRV_HANDLE handle;
uint8_t reg12 = DRV_CAMERA_OVM7690_REG12_SOFT_RESET;

 handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
 if (DRV_HANDLE_INVALID == handle)
 {
     //error
     return;
 }

 if ( DRV_CAMERA_OVM7690_RegisterSet( DRV_CAMERA_OVM7690_REG12_REG_ADDR,
                                      reg12 ) !=
                                      DRV_CAMERA_OVM7690_ERROR_NONE )
 {
     //error
     return;
 }
```

Parameters

Parameters	Description
regIndex	Defines the OVM7690 configuration register addresses.
regValue	Defines the register value to be set.

Function

DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_RegisterSet
(
DRV_CAMERA_OVM7690_REGISTER_ADDRESS regIndex,
uint8_t regValue
)

DRV_CAMERA_OVM7690_Tasks Function

Maintains the OVM7690 state machine.

File

drv_camera_ovm7690.h

C

```
void DRV_CAMERA_OVM7690_Tasks(SYS_MODULE_OBJ object);
```

Function

void DRV_CAMERA_OVM7690_Tasks(SYS_MODULE_OBJ object);

b) Client Setup Functions

DRV_CAMERA_OVM7690_Open Function

Opens the specified OVM7690 Camera Driver instance and returns a handle to it.

File

drv_camera_ovm7690.h

C

```
DRV_HANDLE DRV_CAMERA_OVM7690_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the function returns a valid open instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Errors can occur:

- if the number of client objects allocated via DRV_CAMERA_OVM7690_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client
- if the driver is not ready to be opened, typically when the initialize function has not completed execution

Description

This function opens the specified OVM7690 Camera Driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

Remarks

The handle returned is valid until the DRV_CAMERA_OVM7690_Close function is called. This function will NEVER block waiting for hardware.If the requested intent flags are not supported, the function will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application.

Preconditions

Function DRV_CAMERA_OVM7690_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;
```

```
handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

DRV_HANDLE DRV_CAMERA_OVM7690_Open

(

const SYS_MODULE_INDEX index,

const DRV_IO_INTENT ioIntent

)

DRV_CAMERA_OVM7690_Close Function

Closes an opened instance of the OVM7690 Camera Driver.

File

drv_camera_ovm7690.h

C

```
void DRV_CAMERA_OVM7690_Close(DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened instance of the OVM7690 Camera Driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this function, the handle passed in "handle" must not be used with any of the remaining driver routines (with one possible exception described in the "Remarks" section). A new handle must be obtained by calling DRV_CAMERA_OVM7690_Open before the caller may use the driver again

Remarks

Usually there is no need for the client to verify that the Close operation has completed. The driver will abort any ongoing operations when this function is called.

Preconditions

The DRV_CAMERA_OVM7690_Initialize function must have been called for the specified OVM7690 Camera Driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_USART_Open
DRV_CAMERA_OVM7690_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's Open function

Function

void DRV_CAMERA_OVM7690_Close(DRV_Handle handle)

c) Camera-specific Functions

DRV_CAMERA_OVM7690_FrameBufferAddressSet Function

Sets the framebuffer address.

File

drv_camera_ovm7690.h

C

```
DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_FrameBufferAddressSet(DRV_HANDLE handle, void * frameBuffer);
```

Returns

- DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE - Invalid driver Handle.
- DRV_CAMERA_OVM7690_ERROR_NONE - No error.

Description

This function will set the framebuffer address. This framebuffer address will point to the location at which frame data is to be rendered. This buffer is shared with the display controller to display the frame on the display.

Remarks

This function is mandatory. A valid framebuffer address must be set to display the camera data.

Preconditions

The DRV_CAMERA_OVM7690_Initialize function must have been called for the specified OVM7690 Camera Driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;
uint16_t frameBuffer[DISP_VER_RESOLUTION][DISP_HOR_RESOLUTION];

handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    //error
    return;
}

if ( DRV_CAMERA_OVM7690_FrameBufferAddressSet( handle, (void *) frameBuffer ) !=
                                    DRV_CAMERA_OVM7690_ERROR_NONE )
{
    //error
    return;
}
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's Open function

Function

DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_FrameBufferAddressSet

(

DRV_HANDLE handle,

void * frameBuffer

)

DRV_CAMERA_OVM7690_FrameRectSet Function

Sets the frame rectangle set.

File

drv_camera_ovm7690.h

C

```
DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_FrameRectSet(DRV_HANDLE handle, uint32_t left, uint32_t top,
uint32_t right, uint32_t bottom);
```

Returns

- DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE - Invalid driver Handle.
- DRV_CAMERA_OVM7690_ERROR_NONE - No error.

Description

This function sets the frame rectangle coordinates. The frame within the rectangle is copied to the framebuffer. The left and top values are expected to be less than right and bottom respectively. Left, top, right, and bottom values are also expected to be within range of screen coordinates. Internally it calls the DRV_CAMERA_OVM7690_RegisterSet function to set the respective registers. The rectangle coordinates are also maintained in the driver object.

Remarks

This function is optional if default values are expected to be used.

Preconditions

The DRV_CAMERA_OVM7690_Initialize function must have been called for the specified OVM7690 Camera Driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

The SCCB interface also must have been initialized to configure the OVM7690 Camera Driver.

Example

```
DRV_HANDLE handle;
uint32_t left   = 0x69;
uint32_t top    = 0x0E;
uint32_t right  = DISP_HOR_RESOLUTION + 0x69;
uint32_t bottom = DISP_VER_RESOLUTION + 0x69;

handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    //error
    return;
}

if ( DRV_CAMERA_OVM7690_FrameRectSet( handle, left, top, right, bottom ) !=
                                      DRV_CAMERA_OVM7690_ERROR_NONE )
{
    //error
    return;
}
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's Open function
left	left frame coordinate
top	top frame coordinate
right	right frame coordinate
bottom	bottom frame coordinate

Function

```
DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_FrameRectSet
(
    DRV_HANDLE handle,
uint32_t  left,
uint32_t  top,
uint32_t  right,
uint32_t  bottom
)
```

DRV_CAMERA_OVM7690_Start Function

Starts camera rendering to the display.

File

drv_camera_ovm7690.h

C

```
DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_Start(DRV_HANDLE handle);
```

Returns

- DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE - Invalid driver Handle.
- DRV_CAMERA_OVM7690_ERROR_NONE - No error.

Description

This function starts the camera rendering to the display by writing the pixel data to the framebuffer. The framebuffer is shared between the OVM7690 Camera and the display controller.

Remarks

This function is mandatory. Camera module will not update the framebuffer without calling this function.

Preconditions

The DRV_CAMERA_OVM7690_Initialize function must have been called for the specified OVM7690 Camera Driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

DRV_CAMERA_OVM7690_FrameBufferAddressSet must have been called to set a valid framebuffer address.

Example

```
DRV_HANDLE handle;
uint16_t frameBuffer[DISP_VER_RESOLUTION][DISP_HOR_RESOLUTION];

handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    //error
    return;
}

if ( DRV_CAMERA_OVM7690_FrameBufferAddressSet( handle, (void *) frameBuffer ) !=
                                    DRV_CAMERA_OVM7690_ERROR_NONE )
{
    //error
    return;
}

if ( DRV_CAMERA_OVM7690_Start( handle ) !=
                                    DRV_CAMERA_OVM7690_ERROR_NONE )
{
    //error
    return;
}
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's Open function

Function

```
DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_Start
(
    DRV_HANDLE handle
);
```

DRV_CAMERA_OVM7690_Stop Function

Stops rendering the camera Pixel data.

File

drv_camera_ovm7690.h

C

```
DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_Stop(DRV_HANDLE handle);
```

Returns

- DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE - Invalid driver Handle.
- DRV_CAMERA_OVM7690_ERROR_NONE - No error.

Description

This function starts the camera rendering to the display by writing the pixel data to the framebuffer. The framebuffer is shared between the OVM7690 Camera and the display controller.

Remarks

This function only disables the interrupt for HSYNC and VSYNC. To stop the camera the power-down pin needs to be toggled to an active-high value., which will stop the camera internal clock and maintain the register values.

Preconditions

The DRV_CAMERA_OVM7690_Initialize function must have been called for the specified OVM7690 Camera Driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;

handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    //error
    return;
}

if ( DRV_CAMERA_OVM7690_Stop( handle ) !=
                                    DRV_CAMERA_OVM7690_ERROR_NONE )
{
    //error
    return;
}
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's Open function.

Function

DRV_CAMERA_OVM7690_ERROR DRV_CAMERA_OVM7690_Stop

(

DRV_HANDLE handle

);

d) Other Functions

DRV_CAMERA_OVM7690_HsyncEventHandler Function

Horizontal synchronization event handler.

File

drv_camera_ovm7690.h

C

```
void DRV_CAMERA_OVM7690_HsyncEventHandler(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is called when the OVM7690 Camera sends a Horizontal Sync Pulse on the HSYNC line. It sets the next line address in the DMA module.

Remarks

This function is mandatory.

Preconditions

The DRV_CAMERA_OVM7690_Initialize function must have been called for the specified OVM7690 Camera Driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_CAMERA_OVM7690_INIT     cameraInit;
SYS_MODULE_OBJ              objectHandle;

cameraInit.cameraID                 = CAMERA_MODULE_OVM7690;
cameraInit.sourcePort               = (void *)&PORTK,
cameraInit.hsyncInterruptSource     = INT_SOURCE_CHANGE_NOTICE_A,
cameraInit.vsyncInterruptSource     = INT_SOURCE_CHANGE_NOTICE_J,
cameraInit.dmaChannel               = DRV_CAMERA_OVM7690_DMA_CHANNEL_INDEX,
cameraInit.dmaTriggerSource         = DMA_TRIGGER_EXTERNAL_2,
cameraInit.bpp                      = GFX_CONFIG_COLOR_DEPTH,

objectHandle = DRV_CAMERA_OVM7690_Initialize( DRV_CAMERA_OVM7690_INDEX_0,
                                              (SYS_MODULE_INIT*)&cameraInit);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}

handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    //error
    return;
}

void __ISR( HSYNC_ISR_VECTOR) _Ovm7690HSyncHandler(void)
{
    DRV_CAMERA_OVM7690_HsyncEventHandler(objectHandle);

    SYS_INT_SourceStatusClear(HSYNC_INTERRUPT_SOURCE);
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_CAMERA_OVM7690_Initialize function

Function

void DRV_CAMERA_OVM7690_HsyncEventHandler(SYS_MODULE_OBJ object)

DRV_CAMERA_OVM7690_VsyncEventHandler Function

Vertical synchronization event handler .

File

drv_camera_ovm7690.h

C

```
void DRV_CAMERA_OVM7690_VsyncEventHandler(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is called when the OVM7690 Camera sends a Vertical Sync Pulse on the VSYNC line. It clears the number of lines drawn variable.

Remarks

This function is mandatory.

Preconditions

The DRV_CAMERA_OVM7690_Initialize function must have been called for the specified OVM7690 Camera Driver instance.

DRV_CAMERA_OVM7690_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_CAMERA_OVM7690_INIT     cameraInit;
SYS_MODULE_OBJ              objectHandle;

cameraInit.cameraID                 = CAMERA_MODULE_OVM7690;
cameraInit.sourcePort               = (void *)&PORTK,
cameraInit.hsyncInterruptSource     = INT_SOURCE_CHANGE_NOTICE_A,
cameraInit.vsyncInterruptSource     = INT_SOURCE_CHANGE_NOTICE_J,
cameraInit.dmaChannel               = DRV_CAMERA_OVM7690_DMA_CHANNEL_INDEX,
cameraInit.dmaTriggerSource         = DMA_TRIGGER_EXTERNAL_2,
cameraInit.bpp                      = GFX_CONFIG_COLOR_DEPTH,

objectHandle = DRV_CAMERA_OVM7690_Initialize( DRV_CAMERA_OVM7690_INDEX_0,
                                              (SYS_MODULE_INIT*)&cameraInit);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}

handle = DRV_CAMERA_OVM7690_Open(DRV_CAMERA_OVM7690_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    //error
    return;
}

void __ISR( VSYNC_ISR_VECTOR) _Ovm7690VSyncHandler(void)
{
    DRV_CAMERA_OVM7690_VsyncEventHandler(objectHandle);

    SYS_INT_SourceStatusClear(VSYNC_INTERRUPT_SOURCE);
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_CAMERA_OVM7690_Initialize function

Function

void DRV_CAMERA_OVM7690_VsyncEventHandler(SYS_MODULE_OBJ object)

_DRV_CAMERA_OVM7690_DMAEventHandler Function

File

drv_camera_ovm7690.h

C

```
void _DRV_CAMERA_OVM7690_DMAEventHandler(SYS_DMA_TRANSFER_EVENT event, SYS_DMA_CHANNEL_HANDLE handle,
uintptr_t contextHandle);
```

Description

This is function _DRV_CAMERA_OVM7690_DMAEventHandler.

_DRV_CAMERA_OVM7690_delayMS Function

File

drv_camera_ovm7690.h

C

```
void _DRV_CAMERA_OVM7690_delayMS(unsigned int delayMs);
```

Description

This is function _DRV_CAMERA_OVM7690_delayMS.

_DRV_CAMERA_OVM7690_HardwareSetup Function

File

drv_camera_ovm7690.h

C

```
void _DRV_CAMERA_OVM7690_HardwareSetup(DRV_CAMERA_OVM7690_OBJ * dObj);
```

Description

This is function _DRV_CAMERA_OVM7690_HardwareSetup.

e) Data Types and Constants

DRV_CAMERA_OVM7690_CLIENT_OBJ Structure

OVM7690 Camera Driver client object.

File

drv_camera_ovm7690.h

C

```
typedef struct {
  DRV_CAMERA_OVM7690_OBJ * hDriver;
  DRV_IO_INTENT ioIntent;
  bool inUse;
  DRV_CAMERA_OVM7690_ERROR error;
  DRV_CAMERA_OVM7690_CLIENT_STATUS status;
} DRV_CAMERA_OVM7690_CLIENT_OBJ;
```

Members

Members	Description
DRV_CAMERA_OVM7690_OBJ * hDriver;	The hardware instance object associated with the client
DRV_IO_INTENT ioIntent;	The I/O intent with which the client was opened
bool inUse;	This flags indicates if the object is in use or is available
DRV_CAMERA_OVM7690_ERROR error;	Driver Error

DRV_CAMERA_OVM7690_CLIENT_STATUS status;	Client status

Description

OVM7690 Camera Driver Client Object.

This structure provides a definition of the OVM7690 Camera Driver client object.

Remarks

These values are been updated into the DRV_CAMERA_OVM7690_Open function.

DRV_CAMERA_OVM7690_CLIENT_STATUS Enumeration

Identifies OVM7690 Camera possible client status.

File

drv_camera_ovm7690.h

C

```
typedef enum {
  DRV_CAMERA_OVM7690_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR,
  DRV_CAMERA_OVM7690_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED,
  DRV_CAMERA_OVM7690_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY,
  DRV_CAMERA_OVM7690_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY
} DRV_CAMERA_OVM7690_CLIENT_STATUS;
```

Members

Members	Description
DRV_CAMERA_OVM7690_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR	An error has occurred.
DRV_CAMERA_OVM7690_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED	The driver is closed, no operations for this client are ongoing, and/or the given handle is invalid.
DRV_CAMERA_OVM7690_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY	The driver is currently busy and cannot start additional operations.
DRV_CAMERA_OVM7690_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY	The module is running and ready for additional operations

Description

OVM7690 Camera Client Status.

This enumeration defines possible OVM7690 Camera Client Status.

Remarks

This enumeration values are set by driver interfaces: DRV_CAMERA_OVM7690_Open and DRV_CAMERA_OVM7690_Close.

DRV_CAMERA_OVM7690_ERROR Enumeration

Identifies OVM7690 Camera possible errors.

File

drv_camera_ovm7690.h

C

```
typedef enum {
  DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE,
  DRV_CAMERA_OVM7690_ERROR_NONE
} DRV_CAMERA_OVM7690_ERROR;
```

Members

Members	Description
DRV_CAMERA_OVM7690_ERROR_INVALID_HANDLE	OVM7690 Camera Driver Invalid Handle
DRV_CAMERA_OVM7690_ERROR_NONE	OVM7690 Camera Driver error none

Description

OVM7690 Camera Error flag

This enumeration defines possible OVM7690 Camera errors.

Remarks

This enumeration values are returned by driver interfaces in case of errors.

DRV_CAMERA_OVM7690_INIT Structure

OVM7690 Camera Driver initialization parameters.

File

drv_camera_ovm7690.h

C

```
typedef struct {
  CAMERA_MODULE_ID cameraID;
  void * sourcePort;
  PORTS_CHANNEL hsyncChannel;
  PORTS_BIT_POS hsyncPosition;
  PORTS_CHANNEL vsyncChannel;
  PORTS_BIT_POS vsyncPosition;
  INT_SOURCE hsyncInterruptSource;
  INT_SOURCE vsyncInterruptSource;
  DMA_CHANNEL dmaChannel;
  DMA_TRIGGER_SOURCE dmaTriggerSource;
  uint16_t bpp;
} DRV_CAMERA_OVM7690_INIT;
```

Members

Members	Description
CAMERA_MODULE_ID cameraID;	Camera module ID
void * sourcePort;	Source Port Address
PORTS_CHANNEL hsyncChannel;	HSYNC pin channel
PORTS_BIT_POS hsyncPosition;	HSYNC pin bit position
PORTS_CHANNEL vsyncChannel;	VSYNC pin channel
PORTS_BIT_POS vsyncPosition;	VSYNC pin bit position
INT_SOURCE hsyncInterruptSource;	HSYNC Interrupt Source
INT_SOURCE vsyncInterruptSource;	VSYNC Interrupt Source
DMA_CHANNEL dmaChannel;	DMA channel
DMA_TRIGGER_SOURCE dmaTriggerSource;	DMA trigger source
uint16_t bpp;	Bits per pixel

Description

OVM7690 Camera Initialization parameters

This structure defines OVM7690 Camera Driver initialization parameters.

Remarks

These values should be passed into the DRV_CAMERA_OVM7690_Initialize function.

DRV_CAMERA_OVM7690_OBJ Structure

OVM7690 Camera Driver instance object.

File

drv_camera_ovm7690.h

C

```
typedef struct {
  CAMERA_MODULE_ID moduleId;
  SYS_STATUS status;
  bool inUse;
  bool isExclusive;
  size_t nClients;
  PORTS_CHANNEL hsyncChannel;
  PORTS_BIT_POS hsyncPosition;
```

```
  PORTS_CHANNEL vsyncChannel;
  PORTS_BIT_POS vsyncPosition;
  INT_SOURCE hsyncInterruptSource;
  INT_SOURCE vsyncInterruptSource;
  SYS_DMA_CHANNEL_HANDLE dmaHandle;
  DMA_CHANNEL dmaChannel;
  DMA_TRIGGER_SOURCE dmaTriggerSource;
  bool dmaTransferComplete;
  void * sourcePort;
  uint32_t frameLineCount;
  uint32_t frameLineSize;
  void * frameLineAddress;
  void * frameBufferAddress;
  DRV_CAMERA_OVM7690_RECT rect;
  uint16_t bpp;
} DRV_CAMERA_OVM7690_OBJ;
```

Members

Members	Description
CAMERA_MODULE_ID moduleId;	The module index associated with the object
SYS_STATUS status;	The status of the driver
bool inUse;	Flag to indicate this object is in use
bool isExclusive;	Flag to indicate that driver has been opened exclusively.
size_t nClients;	Keeps track of the number of clients • that have opened this driver
PORTS_CHANNEL hsyncChannel;	HSYNC pin channel
PORTS_BIT_POS hsyncPosition;	HSYNC pin bit position
PORTS_CHANNEL vsyncChannel;	VSYNC pin channel
PORTS_BIT_POS vsyncPosition;	VSYNC pin bit position
INT_SOURCE hsyncInterruptSource;	HSYNC Interrupt Source
INT_SOURCE vsyncInterruptSource;	VSYNC Interrupt Source
SYS_DMA_CHANNEL_HANDLE dmaHandle;	DMA Handle
DMA_CHANNEL dmaChannel;	Read DMA channel
DMA_TRIGGER_SOURCE dmaTriggerSource;	DMA Trigger Source
bool dmaTransferComplete;	DMA Transfer Complete Flag
void * sourcePort;	Source Port Address
uint32_t frameLineCount;	Frame Line Count
uint32_t frameLineSize;	Frame Line Size
void * frameLineAddress;	Frame Line Address
void * frameBufferAddress;	Framebuffer Address
DRV_CAMERA_OVM7690_RECT rect;	Window Rectangle
uint16_t bpp;	Bits per pixel supported

Description

OVM7690 Camera Driver Instance Object

This structure provides a definition of the OVM7690 Camera Driver instance object.

Remarks

These values are been updated into the DRV_CAMERA_OVM7690_Initialize function.

DRV_CAMERA_OVM7690_RECT Structure

OVM7690 Camera window rectangle coordinates.

File

drv_camera_ovm7690.h

C

```
typedef struct {
  uint32_t left;
  uint32_t top;
  uint32_t right;
  uint32_t bottom;
```

```
} DRV_CAMERA_OVM7690_RECT;
```

Members

Members	Description
uint32_t left;	OVM7690 Camera Window left coordinate
uint32_t top;	OVM7690 Camera Window top coordinate
uint32_t right;	OVM7690 Camera Window right coordinate
uint32_t bottom;	OVM7690 Camera Window bottom coordinate

Description

OVM7690 Camera Window Rect

This structure defines window rectangle co-ordinates as left, right, top, and bottom.

Remarks

These values should be passed into the DRV_CAMERA_OVM7690_FrameRectSet function.

DRV_CAMERA_OVM7690_REG12_OP_FORMAT Enumeration

Lists OVM7690 Camera device register addresses.

File

drv_camera_ovm7690.h

C

```
typedef enum {
  DRV_CAMERA_OVM7690_REG12_OP_FORMAT_RAW_2
} DRV_CAMERA_OVM7690_REG12_OP_FORMAT;
```

Members

Members	Description
DRV_CAMERA_OVM7690_REG12_OP_FORMAT_RAW_2	Bayer Raw Format

Description

OVM7690 Camera Device Register Addresses.

This enumeration defines the list of device register addresses.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_CAMERA_OVM7690_RegisterSet function. Refer to the specific device data sheet for more information.

DRV_CAMERA_OVM7690_INDEX_0 Macro

OVM7690 driver index definitions.

File

drv_camera_ovm7690.h

C

```
#define DRV_CAMERA_OVM7690_INDEX_0 0
```

Description

OVM7690 Camera Driver Module Index

These constants provide OVM7690 Camera Driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_CAMERA_OVM7690_Initialize and DRV_CAMERA_OVM7690_Open routines to identify the driver instance in use.

DRV_CAMERA_OVM7690_INDEX_1 Macro

File

drv_camera_ovm7690.h

C

```
#define DRV_CAMERA_OVM7690_INDEX_1 1
```

Description

This is macro DRV_CAMERA_OVM7690_INDEX_1.

DRV_CAMERA_OVM7690_REG12_SOFT_RESET Macro

OVM7690 Camera Driver Register 0x12 Soft reset flag.

File

drv_camera_ovm7690.h

C

```
#define DRV_CAMERA_OVM7690_REG12_SOFT_RESET
```

Description

OVM7690 Camera Driver Soft reset flag.

This macro provides a definition of the OVM7690 Camera Register 0x12 Soft reset flag.

Remarks

These constants should be used in place of hard-coded numeric literals.

DRV_CAMERA_OVM7690_SCCB_READ_ID Macro

OVM7690 Camera SCCB Interface device Read Slave ID.

File

drv_camera_ovm7690.h

C

```
#define DRV_CAMERA_OVM7690_SCCB_READ_ID
```

Description

OVM7690 Camera Driver SCCB Read ID

This macro provides a definition of the OVM7690 Camera SCCB Interface device Read Slave ID.

Remarks

These constants should be used in place of hard-coded numeric literals.

DRV_CAMERA_OVM7690_SCCB_WRITE_ID Macro

OVM7690 Camera SCCB Interface device Write Slave ID.

File

drv_camera_ovm7690.h

C

```
#define DRV_CAMERA_OVM7690_SCCB_WRITE_ID
```

Description

OVM7690 Camera Driver SCCB Write ID

This macro provides a definition of the OVM7690 Camera SCCB Interface device Write Slave ID.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_CAMERA_OVM7690_RegisterSet function to identify the OVM7690 Camera SCCB Interface device Write Slave ID.

Files

Files

Name	Description
drv_camera_ovm7690.h	OVM7690 Camera Driver local data structures.
drv_ovm7690_config_template.h	OVM7690 Device Driver configuration template.

Description

drv_camera_ovm7690.h

OVM7690 Camera Driver local data structures.

Enumerations

	Name	Description
	DRV_CAMERA_OVM7690_CLIENT_STATUS	Identifies OVM7690 Camera possible client status.
	DRV_CAMERA_OVM7690_ERROR	Identifies OVM7690 Camera possible errors.
	DRV_CAMERA_OVM7690_REG12_OP_FORMAT	Lists OVM7690 Camera device register addresses.

Functions

	Name	Description
	_DRV_CAMERA_OVM7690_delayMS	This is function _DRV_CAMERA_OVM7690_delayMS.
	_DRV_CAMERA_OVM7690_DMAEventHandler	This is function _DRV_CAMERA_OVM7690_DMAEventHandler.
	_DRV_CAMERA_OVM7690_HardwareSetup	This is function _DRV_CAMERA_OVM7690_HardwareSetup.
	DRV_CAMERA_OVM7690_Close	Closes an opened instance of the OVM7690 Camera Driver.
	DRV_CAMERA_OVM7690_Deinitialize	Deinitializes the specified instance of the OVM7690 Camera Driver module.
	DRV_CAMERA_OVM7690_FrameBufferAddressSet	Sets the framebuffer address.
	DRV_CAMERA_OVM7690_FrameRectSet	Sets the frame rectangle set.
	DRV_CAMERA_OVM7690_HsyncEventHandler	Horizontal synchronization event handler.
	DRV_CAMERA_OVM7690_Initialize	Initializes the OVM7690 Camera instance for the specified driver index.
	DRV_CAMERA_OVM7690_Open	Opens the specified OVM7690 Camera Driver instance and returns a handle to it.
	DRV_CAMERA_OVM7690_RegisterSet	Sets the camera OVM7690 configuration registers.
	DRV_CAMERA_OVM7690_Start	Starts camera rendering to the display.
	DRV_CAMERA_OVM7690_Stop	Stops rendering the camera Pixel data.
	DRV_CAMERA_OVM7690_Tasks	Maintains the OVM7690 state machine.
	DRV_CAMERA_OVM7690_VsyncEventHandler	Vertical synchronization event handler .

Macros

	Name	Description
	DRV_CAMERA_OVM7690_INDEX_0	OVM7690 driver index definitions.
	DRV_CAMERA_OVM7690_INDEX_1	This is macro DRV_CAMERA_OVM7690_INDEX_1.
	DRV_CAMERA_OVM7690_REG12_SOFT_RESET	OVM7690 Camera Driver Register 0x12 Soft reset flag.
	DRV_CAMERA_OVM7690_SCCB_READ_ID	OVM7690 Camera SCCB Interface device Read Slave ID.
	DRV_CAMERA_OVM7690_SCCB_WRITE_ID	OVM7690 Camera SCCB Interface device Write Slave ID.

Structures

	Name	Description
	DRV_CAMERA_OVM7690_CLIENT_OBJ	OVM7690 Camera Driver client object.
	DRV_CAMERA_OVM7690_INIT	OVM7690 Camera Driver initialization parameters.
	DRV_CAMERA_OVM7690_OBJ	OVM7690 Camera Driver instance object.

	DRV_CAMERA_OVM7690_RECT	OVM7690 Camera window rectangle coordinates.

Description

OVM7690 Camera Driver Local Data Structures

This header file provides the local data structures for the OVM7690 Camera Driver Library.

File Name

drv_camera_ovm7690.h

Company

Microchip Technology Inc.

drv_ovm7690_config_template.h

OVM7690 Device Driver configuration template.

Macros

	Name	Description
	DRV_OVM7690_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.

Description

OVM7690 Device Driver Configuration Template

This header file contains the build-time configuration selections for the OVM7690 device driver. This is the template file which give all possible configurations that can be made. This file should not be included in any project.

File Name

drv_ovm7690_config_template.h

Company

Microchip Technology Inc.

CAN Driver Library

This section describes the CAN Driver Library.

Introduction

The CAN Static Driver provides a high-level interface to manage the CAN module on the Microchip family of microcontrollers.

Description

Through MHC, this driver provides an API to initialize the CAN module, as well as the baud rate. The API also allows simple transmit and receive functionality.

Library Interface

Function(s)

	Name	Description
	DRV_CAN_ChannelMessageReceive	Receives a message on a channel for the specified driver index. **Implementation:** Static
	DRV_CAN_ChannelMessageTransmit	Transmits a message on a channel for the specified driver index. **Implementation:** Static
	DRV_CAN_Close	Closes the CAN instance for the specified driver index. **Implementation:** Static
	DRV_CAN_Deinitialize	Deinitializes the DRV_CAN_Initialize instance that has been called for the specified driver index. **Implementation:** Static
	DRV_CAN_Initialize	Initializes the CAN instance for the specified driver index. **Implementation:** Static

	DRV_CAN_Open	Opens the CAN instance for the specified driver index. **Implementation:** Static

Description

This section describes the Application Programming Interface (API) functions of the CAN Driver Library.

Function(s)

DRV_CAN_ChannelMessageReceive Function

Receives a message on a channel for the specified driver index.

Implementation: Static

File

help_drv_can.h

C

```
bool DRV_CAN_ChannelMessageReceive(CAN_CHANNEL channelNum, int address, uint8_t DLC, uint8_t* message);
```

Returns

- true - When a message has been received
- false - When a message has not been received

Description

This routine receives data into a buffer from the CAN bus according to the channel, address, and data length given.

Remarks

This routine receives a standard or extended messages based upon the CAN Driver setup.

Preconditions

DRV_CAN_Initialize has been called.

Parameters

Parameters	Description
CAN_CHANNEL channelNum	CAN channel to use
int address	CAN address to receive on
uint8_t DLC	Data Length Code of Message
uint8_t* message	Pointer to put the message data to receive

Function

bool DRV_CAN_ChannelMessageReceive(CAN_CHANNEL channelNum, int address,
uint8_t DLC, uint8_t* message);

DRV_CAN_ChannelMessageTransmit Function

Transmits a message on a channel for the specified driver index.

Implementation: Static

File

help_drv_can.h

C

```
bool DRV_CAN_ChannelMessageTransmit(CAN_CHANNEL channelNum, int address, uint8_t DLC, uint8_t* message);
```

Returns

Boolean "true" when a message has been transmitted.

Description

This routine transmits a data buffer on the CAN bus according to the channel, address, and data length given.

Remarks

This routine receives a standard or extended messages based upon the CAN Driver setup.

Preconditions

DRV_CAN_Initialize has been called.

Parameters

Parameters	Description
CAN_CHANNEL channelNum	CAN channel to use
int address	CAN address to transmit on
uint8_t DLC	Data Length Code of Message
uint8_t* message	Pointer to the message data to send

Function

bool DRV_CAN_ChannelMessageTransmit(CAN_CHANNEL channelNum, int address,
uint8_t DLC, uint8_t* message);

DRV_CAN_Close Function

Closes the CAN instance for the specified driver index.

Implementation: Static

File

help_drv_can.h

C

```
void DRV_CAN_Close();
```

Returns

None.

Description

This routine closes the CAN driver instance for the specified driver instance, making it ready for clients to use it.

Preconditions

DRV_CAN_Initialize has been called.

Function

void DRV_CAN_Close(void)

DRV_CAN_Deinitialize Function

Deinitializes the DRV_CAN_Initialize instance that has been called for the specified driver index.

Implementation: Static

File

help_drv_can.h

C

```
void DRV_CAN_Deinitialize();
```

Returns

None.

Description

This routine deinitializes the CAN Driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Preconditions

None.

Function

void DRV_CAN_Deinitialize(void)

DRV_CAN_Initialize Function

Initializes the CAN instance for the specified driver index.

Implementation: Static

File

help_drv_can.h

C

```
void DRV_CAN_Initialize();
```

Returns

None.

Description

This routine initializes the CAN Driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Remarks

This routine must be called before any other CAN routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Function

void DRV_CAN_Initialize(void)

DRV_CAN_Open Function

Opens the CAN instance for the specified driver index.

Implementation: Static

File

help_drv_can.h

C

```
void DRV_CAN_Open();
```

Returns

None.

Description

This routine opens the CAN Driver instance for the specified driver instance, making it ready for clients to use it.

Preconditions

DRV_CAN_Initialize has been called.

Function

void DRV_CAN_Open(void)

Codec Driver Libraries

This section describes the Codec Driver Libraries available in MPLAB Harmony.

AK4384 Codec Driver Library

This topic describes the AK4384 Codec Driver Library.

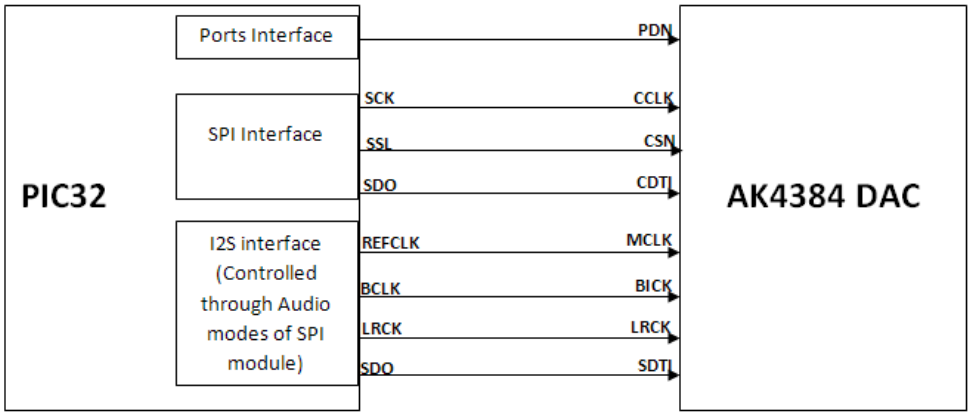
Introduction

This library provides an interface to manage the AK4384 106 dB 192 kHz 24-Bit DAC that is serially interfaced to a Microchip microcontroller for providing Audio Solutions.

Description

The AK4384 module is 24-bit Audio DAC from Asahi Kasei Microdevices Corporation. The AK4384 can be interfaced to Microchip microcontrollers through SPI and I2S serial interfaces. SPI interface is used for control command transfer. The I2S interface is used for Audio data output.

A typical interface of AK4384 to a Microchip PIC32 device is provided in the following diagram:



Features

The AK4384 Codec Driver supports the following features:

- Sampling Rate Ranging from 8 kHz to 192 kHz
- 128 times Oversampling (Normal Speed mode)
- 64 times Oversampling (Double Speed mode)
- 32 times Oversampling (Quad Speed mode)
- Digital de-emphasis for 32k, 44.1k and 48 kHz sampling
- Soft mute
- Digital Attenuator (Linear 256 steps)
- I/F format:
  - 24-bit MSB justified
  - 24/20/16-bit LSB justified
  - I2S
- Master clock:
  - 256 fs, 384 fs, 512 fs, 768 fs, or 1152 fs (Normal Speed mode)
  - 128 fs, 192 fs, 256 fs, or 384 fs (Double Speed mode)
  - 128 fs or 192 fs (Quad Speed mode)

Using the Library

This topic describes the basic architecture of the AK4384 Codec Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_ak4384.h`

The interface to the AK4384 Codec Driver library is defined in the `drv_ak4384.h` header file. Any C language source (`.c`) file that uses the AK4384 Codec Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

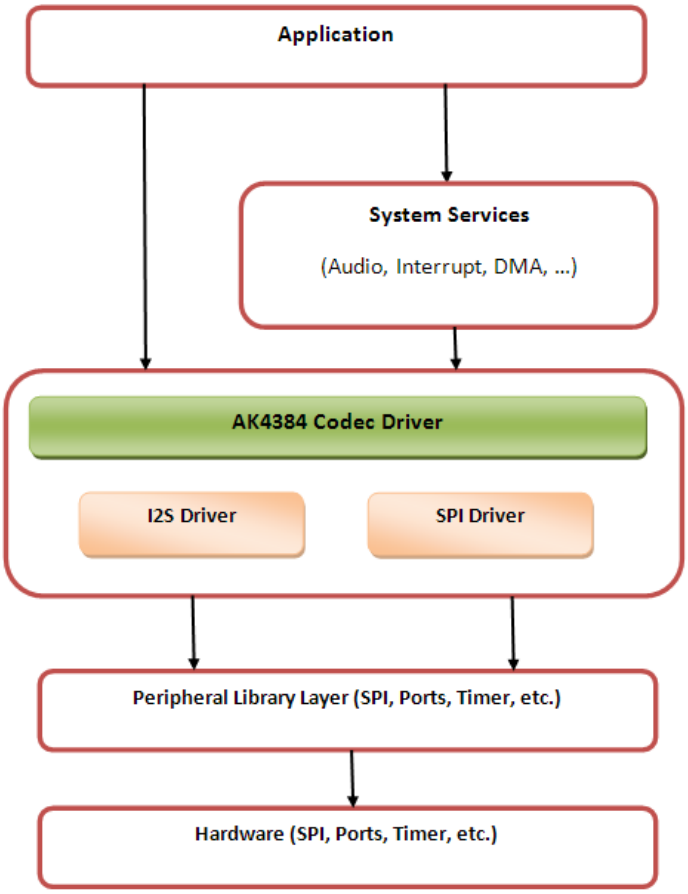
Abstraction Model

This library provides a low-level abstraction of the AK4384 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The abstraction model shown in the following diagram depicts how the AK4384 Codec Driver is positioned in the MPLAB Harmony framework. The AK4384 Codec Driver uses the SPI and I2S drivers for control and audio data transfers to the AK4384 module.

AK4384 Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The AK4384 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced AK4384 DAC module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the AK4384 Codec Driver Library.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open and close functions.
Codec Specific Functions	Provides functions that are Codec-specific.
Data Transfer Functions	Provides data transfer functions.
Other Functions	Provides driver specific miscellaneous functions such as sampling rate setting, control command functions, etc.
Data Types and Constants	These data types and constants are required while interacting and setting up the AK4384 Codec Driver Library.

How the Library Works

The library provides interfaces to support:

- System Functionality

- Client Functionality

System Access

This topic provides information on system initialization, implementations, and provides a system access code example.

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the AK4384 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_AK4384_INIT or by using Initialization Overrides) that are supported by the specific AK4384 device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- SPI driver module index. The module index should be same as the one used in initializing the SPI Driver.
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver.
- Sampling rate
- Master clock detection mode
- Power down pin port initialization
- Queue size for the audio data transmit buffer

The DRV_AK4384_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ AK4384_Deinitialize, DRV_ AK4384_Status and DRV_I2S_Tasks.

Implementations

The AK4384 Codec Driver can have the following implementations:

Implementation	Description	MPLAB Harmony Components
Implementation 1	Dedicated hardware for control (SPI) and data (I2S) interface.	Standard MPLAB Harmony drivers for SPI and I2S interfaces.
Implementation 2	Dedicated hardware for data (I2S) interface. Ports pins for control interface.	Standard MPLAB Harmony drivers for I2S interface. Virtual MPLAB Harmony drivers for SPI interface.
Implementation 3	Dedicated hardware for data (I2S) interface. Ports pins for control.	Standard MPLAB Harmony drivers for I2S interface. An internal bit-banged implementation of control interface in the AK4384 Codec Driver.

If Implementation 3 is in use, while initializing fields of DRV_AK4384_INIT structure, the SPI Driver module index initialization is redundant. The user can pass a dummy value.

For Implementation 3, the user has to additionally initialize parameters to support bit-banged control interface implementation. These additional parameters can be passed by assigning values to the respective macros in `system_config.h`.

Example:

```
DRV_AK4384_INIT drvak4384Init =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .volume = 120,
    .mclkMode = DRV_AK4384_MCLK_MODE_MANUAL,
    .queueSizeTransmit = 2,
};

/*
 The SPI module index should be same as the one used in
 initializing the SPI driver.
 The SPI module index initialization is redundant
 if Implementation 3 is in use.
 */
drvak4384Init.spiDriverModuleIndex = DRV_SPI_INDEX_0;

/*
 The I2S module index should be same as the one used in
 initializing the I2S driver.
 */
drvak4384Init.i2sDriverModuleIndex = DRV_I2S_INDEX_0;

ak4384DevObject = DRV_AK4384_Initialize(DRV_AK4384_INDEX_0, (SYS_MODULE_INIT *) &drvak4384Init);
```

```
if (SYS_MODULE_OBJ_INVALID == ak4384DevObject)
{
    // Handle error
}
```

Task Routine

The DRV_AK4384_Tasks will be called from the System Task Service.

Client Access

This topic describes client access and includes a code example.

Description

For the application to start using an instance of the module, it must call the DRV_AK4384_Open function. The DRV_AK4384_Open provides a driver handle to the AK4384 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_AK4384_Deinitialize, the application must call the DRV_AK4384_Open function again to set up the instance of the driver.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

Note: It is necessary to check the status of driver initialization before opening a driver instance. The status of the AK4384 Codec Driver can be known by calling DRV_AK4384_Status.

Example:

```
DRV_HANDLE handle;
SYS_STATUS ak4384Status;
    ak4384Status = DRV_AK4384_Status(sysObjects.ak4384DevObject);
        if (SYS_STATUS_READY == ak4384Status)
        {
                // The driver can now be opened.
                appData.ak4384Client.handle = DRV_AK4384_Open
                                              (DRV_AK4384_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
                if(appData.ak4384Client.handle != DRV_HANDLE_INVALID)
                {
                    appData.state = APP_STATE_AK4384_SET_BUFFER_HANDLER;
                }
                else
                {
                    SYS_DEBUG(0, "Find out what's wrong \r\n");
                }
    }
    else
    {
        /* AK4384 Driver Is not ready */
                ;
    }
```

Client Operations

This topic describes client operations and provides a code example.

Description

Client operations provide the API interface for control command and audio data transfer to the AK4384 Codec.

The following AK4384 Codec specific control command functions are provided:

Notes:

1. The calling and execution of the following functions does not guarantee that the function (and its associated Codec command) has been set in the Codec peer interfaced through the SPI. It just means that the submission of the command has started over the SPI.
2. Regarding Note 1, the user should not call the following functions consecutively, which could result in unexpected behavior. If needed, the user should confirm the completion status of a function before calling any of the other functions.
3. To know the completion status of the following functions, users can register a command event callback handler by calling the function 'DRV_AK4384_CommandEventHandlerSet'. The callback handler will be called when the last submitted command (submitted by calling one of the following functions) has completed.

- DRV_AK4384_SamplingRateSet
- DRV_AK4384_SamplingRateGet
- DRV_AK4384_VolumeSet

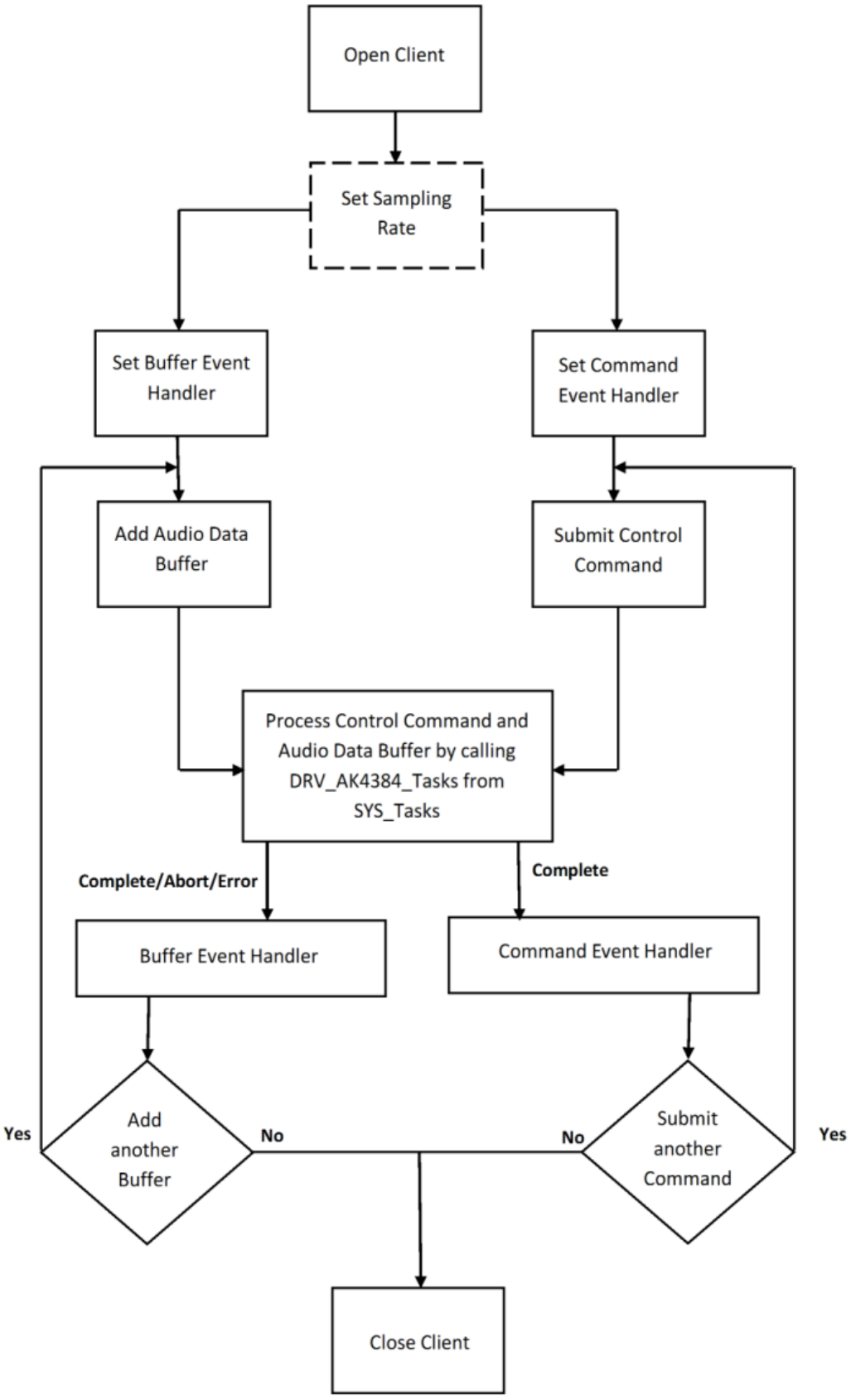
- DRV_AK4384_VolumeGet
- DRV_AK4384_MuteOn
- DRV_AK4384_MuteOff
- DRV_AK4384_ZeroDetectEnable
- DRV_AK4384_ZeroDetectDisable
- DRV_AK4384_ZeroDetectModeSet
- DRV_AK4384_ZeroDetectInvertEnable
- DRV_AK4384_ZeroDetectInvertDisable
- DRV_AK4384_ChannelOutputInvertEnable
- DRV_AK4384_ChannelOutputInvertDisable
- DRV_AK4384_SlowRollOffFilterEnable
- DRV_AK4384_SlowRollOffFilterDisable
- DRV_AK4384_DeEmphasisFilterSet

These functions schedule a non-blocking control command transfer operation. These functions submit the control command request to the AK4384 Codec. A notification for the submitted requests can be received by registering a command callback event with the driver. The driver notifies by calling the callback on successfully transmitting the command to the AK4384 Codec module.

The function DRV_AK4384_BufferAddWrite is a buffered data operation functions. This function schedules non-blocking audio data transfer operation. The function adds the request to the hardware instance queues and returns a buffer handle. The requesting client also registers a callback event with the driver. The driver notifies the client with DRV_AK4384_BUFFER_EVENT_COMPLETE, DRV_AK4384_BUFFER_EVENT_ERROR, or DRV_AK4384_BUFFER_EVENT_ABORT events.

The submitted control commands and audio buffer add requests are processed under DRV_AK4384_Tasks function. This function is called from the SYS_Tasks routine.

The following diagram illustrates the control commands and audio buffered data operations.



Note: It is not necessary to close and reopen the client between multiple transfers.

An application using the buffered functionality needs to perform the following steps:

1. The system should have completed necessary setup and initializations.

2. The I2S Driver object should have been initialized by calling DRV_I2S_Initialize.
3. The SPI Driver object should have been initialized by calling DRV_SPI_Initialize.
4. The AK4384 Codec Driver object should be initialized by calling DRV_AK4384_Initialize.
5. The necessary sampling rate value should be set up by calling DRV_AK4384_ SamplingRateSet.
6. Register buffer event handler for the client handle by calling DRV_AK4384_BufferEventHandlerSet.
7. Register command event handler for the client handle by calling DRV_AK4384_CommandEventHandlerSet.
8. Submit a command by calling specific command API.
9. Add a buffer to initiate the data transfer by calling DRV_AK4384_BufferAddWrite.
10. The submitted command and Audio data processing happens b calling DRV_AK4384_Tasks from SYS_Tasks.
11. Repeat steps 9 through 10 to handle multiple buffer transmission and reception.
12. When the client is done, it can use DRV_AK4384_Close to close the client handle.

Example:

```
typedef enum
{
    APP_STATE_AK4384_OPEN,
    APP_STATE_AK4384_SET_COMMAND_HANDLER,
    APP_STATE_AK4384_SET_BUFFER_HANDLER,
    APP_STATE_AK4384_SET_SAMPLING_RATE_COMMAND,
    APP_STATE_AK4384_ADD_BUFFER,
    APP_STATE_AK4384_WAIT_FOR_BUFFER_COMPLETE,
    APP_STATE_AK4384_BUFFER_COMPLETE
} APP_STATES;

typedef struct
{
    DRV_HANDLE handle;
    DRV_AK4384_BUFFER_HANDLE writeBufHandle;
    DRV_AK4384_BUFFER_EVENT_HANDLER bufferHandler;
    DRV_AK4384_COMMAND_EVENT_HANDLER commandHandler;
    uintptr_t context;
    uint8_t *txbufferObject;
    size_t bufferSize;

} APP_AK4384_CLIENT;

typedef struct
{
    /* Application's current state*/
    APP_STATES state;
    /* USART client handle */
    APP_AK4384_CLIENT ak4384Client;
} APP_DATA;
APP_DATA appData;
SYS_MODULE_OBJ ak4384DevObject;
DRV_AK4384_INIT drvak4384Init =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .volume = 120,
    .mclkMode = DRV_AK4384_MCLK_MODE_MANUAL,
    .queueSizeTransmit = 2,
};

void SYS_Initialize(void * data)
{
    /*
     The SPI module index should be same as the one used in
     initializing the SPI driver.
     The SPI module index initialization is redundant
     if Implementation 3 (Described in System Access) is in use.
     */
    drvak4384Init.spiDriverModuleIndex = DRV_SPI_INDEX_0;

    /*
     The I2S module index should be same as the one used in
     initializing the I2S driver.
     */
    drvak4384Init.i2sDriverModuleIndex = DRV_I2S_INDEX_0;
```

```
    ak4384DevObject = DRV_AK4384_Initialize(DRV_AK4384_INDEX_0, (SYS_MODULE_INIT *) & drvak4384Init);
    if (SYS_MODULE_OBJ_INVALID == ak4384DevObject) {
        // Handle error
    }
}

void APP_Tasks (void )
{
    switch(appData.state)
    {
        /* Open the ak4384 client and get an Handle */
        case APP_STATE_AK4384_OPEN:
        {
            SYS_STATUS ak4384Status;
            ak4384Status = DRV_AK4384_Status(sysObjects.ak4384DevObject);
            if (SYS_STATUS_READY == ak4384Status)
            {
                // This means the driver can now be opened.
                appData.ak4384Client.handle = DRV_AK4384_Open(DRV_AK4384_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
                if(appData.ak4384Client.handle != DRV_HANDLE_INVALID)
                {
                    appData.state = APP_STATE_AK4384_SET_COMMAND_HANDLER;
                }
                else
                {
                    SYS_DEBUG(0, "Find out what is wrong \r\n");
                }
            }
            else
            {
                /* Wait for AK4384 to Initialize */
                ;
            }
        }
        break;

        /* Register a command event handler */
        case APP_STATE_AK4384_SET_COMMAND_HANDLER:
        {
            DRV_AK4384_CommandEventHandlerSet(appData.ak4384Client.handle,
                    appData.ak4384Client.commandHandler,
                    appData.ak4384Client.context);
            appData.state = APP_STATE_AK4384_SET_BUFFER_HANDLER;
        }
        break;


        /* Register a buffer event handler */
        case APP_STATE_AK4384_SET_BUFFER_HANDLER:
        {
            DRV_AK4384_BufferEventHandlerSet(appData.ak4384Client.handle,
                    appData.ak4384Client.bufferHandler,
                    appData.ak4384Client.context);
            appData.state = APP_STATE_AK4384_SET_SAMPLING_RATE_COMMAND;
        }
        break;

        /* Submit a set sampling rate command */
        case APP_STATE_AK4384_SET_SAMPLING_RATE_COMMAND:
        {
            DRV_AK4384_SamplingRateSet(appData.ak4384Client.handle,48000);
            appData.state = APP_STATE_AK4384_ADD_BUFFER;
        }
        break;

        /* Add the Audio buffer to be transmitted */
        case APP_STATE_AK4384_ADD_BUFFER:
        {
```

```
            DRV_AK4384_BufferAddWrite(appData.ak4384Client.handle, &appData.ak4384Client.writeBufHandle,
            appData.ak4384Client.txbufferObject, appData.ak4384Client.bufferSize);
            if(appData.ak4384Client.writeBufHandle != DRV_AK4384_BUFFER_HANDLE_INVALID)
            {
                appData.state = APP_STATE_AK4384_WAIT_FOR_BUFFER_COMPLETE;
            }
            else
            {
                SYS_DEBUG(0, "Find out what is wrong \r\n");
            }
        }
        break;

    /* Audio Buffer transmission under process */
        case APP_STATE_AK4384_WAIT_FOR_BUFFER_COMPLETE:
        {
        }
        break;

    /* Audio Buffer transmission completed */
        case APP_STATE_AK4384_BUFFER_COMPLETE:
        {
            /* Add another buffer */
            appData.state = APP_STATE_AK4384_ADD_BUFFER;
        }
        break;

        default:
        {
        }
        break;
    }

}

void APP_AK4384CommandEventHandler(uintptr_t context )
{

    // Last submitted command successful. Take action as needed.
}

void APP_AK4384BufferEventHandler(DRV_AK4384_BUFFER_EVENT event,
        DRV_AK4384_BUFFER_HANDLE handle, uintptr_t context )
{

    switch(event)
    {
        case DRV_AK4384_BUFFER_EVENT_COMPLETE:
        {
            // Can set appData.state = APP_STATE_AK4384_BUFFER_COMPLETE;
        // Take Action as needed

        }
        break;
        case DRV_AK4384_BUFFER_EVENT_ERROR:
        {
        // Take Action as needed

        } break;

        case DRV_AK4384_BUFFER_EVENT_ABORT:
        {
        // Take Action as needed

        } break;

    }
}
```

```
void SYS_Tasks(void)
{
    DRV_AK4384_Tasks(ak4384DevObject);
    APP_Tasks();
}
```

Configuring the Library

Macros

	Name	Description
	DRV_AK4384_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4384_CONTROL_CLOCK	Sets up clock frequency for the control interface (SPI)
	DRV_AK4384_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4384_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4384_TIMER_DRIVER_MODULE_INDEX	Identifies the Timer Module Index for custom virtual SPI driver implementation.
	DRV_AK4384_TIMER_PERIOD	Identifies the period for the bit bang timer.
	DRV_AK4384_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1, and 48K sampling frequency
	DRV_AK4384_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1 and 48K sampling frequency

Description

The configuration of the AK4384 Codec Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the AK4384 Codec Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the AK4384 Codec Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_AK4384_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_CLIENTS_NUMBER DRV_AK4384_INSTANCES_NUMBER
```

Description

AK4384 Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. Typically only one client could be connected to one hardware instance. This value represents the total number of clients to be supported across all hardware instances. Therefore, if there are five AK4384 hardware interfaces, this number will be 5.

Remarks

None.

DRV_AK4384_CONTROL_CLOCK Macro

Sets up clock frequency for the control interface (SPI)

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_CONTROL_CLOCK
```

Description

AK4384 Control Interface Clock Speed configuration

Sets up clock frequency for the control interface (SPI). The maximum value supported is 5MHZ.

Remarks

1. This Macro is useful only when a hardware SPI module is not available(used) or a virtual SPI driver is not available(used) for the control interface to the AK4384 CODEC.
2. This constant needs to defined only for a bit banged implementation of control interface with in the driver.

DRV_AK4384_INPUT_REFCLOCK Macro

Identifies the input REFCLOCK source to generate the MCLK to codec.

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_INPUT_REFCLOCK
```

Description

AK4384 Input reference clock

Identifies the input REFCLOCK source to generate the MCLK to codec.

Remarks

None.

DRV_AK4384_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_INSTANCES_NUMBER
```

Description

AK4384 driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of AK4384 CODEC modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None.

DRV_AK4384_TIMER_DRIVER_MODULE_INDEX Macro

Identifies the Timer Module Index for custom virtual SPI driver implementation.

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_TIMER_DRIVER_MODULE_INDEX
```

Description

AK4384 Timer Module Index

Identifies the Timer Module Index for custom virtual SPI driver implementation. The AK4384 uses SPI protocol for control interface. The Timer Module Index is needed by AK4384 driver to implement a virtual SPI driver for control command exchange with the AK4384 CODEC.

Remarks

1. This Macro is useful only when a hardware SPI module is not available(used) or a virtual SPI driver is not available(used) for the control interface to the AK4384 CODEC.
2. This constant needs to defined only for a bit banged implementation of control interface with in the driver.

DRV_AK4384_TIMER_PERIOD Macro

Identifies the period for the bit bang timer.

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_TIMER_PERIOD
```

Description

AK4384 Timer Period

Identifies the period for the bit bang timer after which the timer interrupt should occur. The value assigned should align with the expected control interface clock defined by AK4384_CONTROL_CLOCK.

Remarks

1. This Macro is useful only when a hardware SPI module is not available(used) or a virtual SPI driver is not available(used) for the control interface to the AK4384 CODEC.
2. This constant needs to defined only for a bit banged implementation of control interface with in the driver.

DRV_AK4384_BCLK_BIT_CLK_DIVISOR Macro

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1, and 48K sampling frequency

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_BCLK_BIT_CLK_DIVISOR
```

Description

AK4384 BCLK to LRCK Ratio to Generate Audio Stream

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1 and 48K I2S sampling frequency

Following BCLK to LRCK ratios are supported 16bit LSB Justified >=32fs 20bit LSB Justified >=40fs 24bit MSB Justified >=48fs 24bit I2S Compatible >=48fs 24bit LSB Justified >=48fs

Typical values for the divisor are 1,2,4 and 8

Remarks

None.

DRV_AK4384_MCLK_SAMPLE_FREQ_MULTPLIER Macro

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1 and 48K sampling frequency

File

drv_ak4384_config_template.h

C

```
#define DRV_AK4384_MCLK_SAMPLE_FREQ_MULTPLIER
```

Description

AK4384 MCLK to LRCK Ratio to Generate Audio Stream

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1, and 48K I2S sampling frequency

Supported MCLK to LRCK Ratios are as below 256fs, 384fs, 512fs, 768fs or 1152fs [Normal Speed Mode(8kHz~48kHz)] 128fs, 192fs, 256fs or 384fs [Double Speed Mode(60kHz~96kHz)] 128fs, 192fs [Quad Speed Mode(120kHz~192kHz)]

Remarks

None

Configuring the MHC

Provides examples on how to configure the MPLAB Harmony Configurator (MHC) for a specific driver.

Description

The following three figures show examples of MHC configurations for the AK4384 Codec Driver, I2S Driver, and the Timer Driver.

Figure 1: AK4384 Codec Driver MHC Configuration

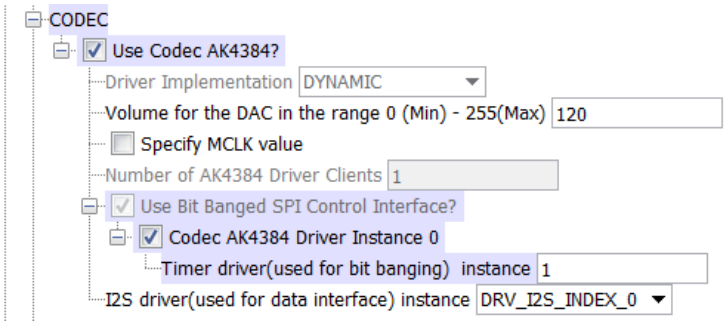


Figure 2: I2S Driver MHC Configuration

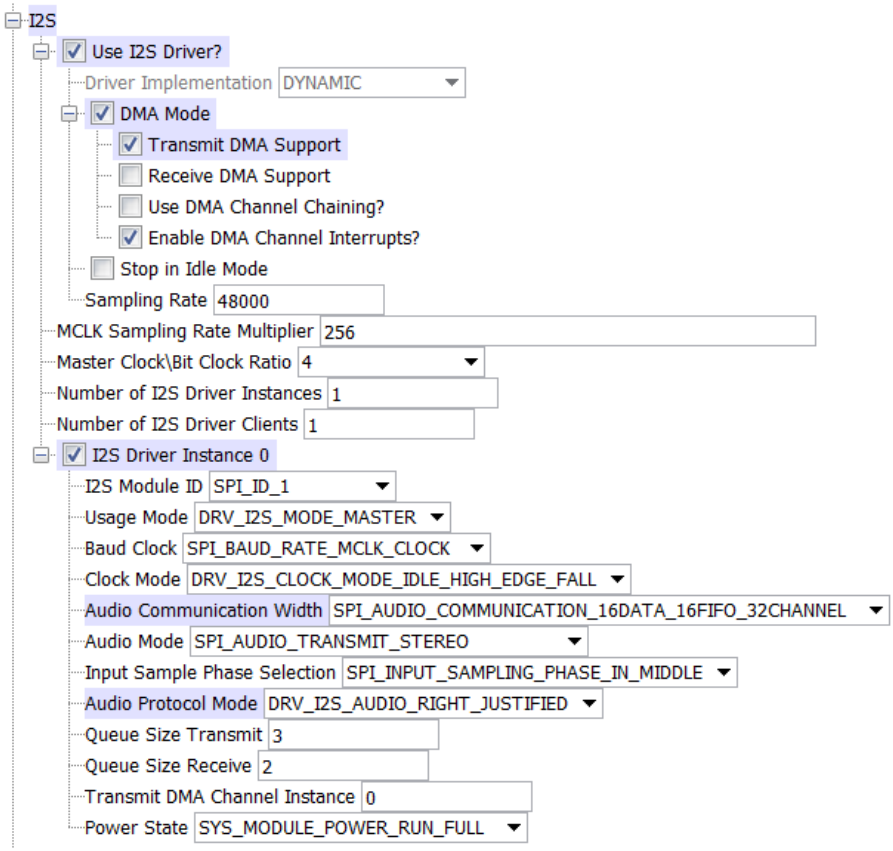
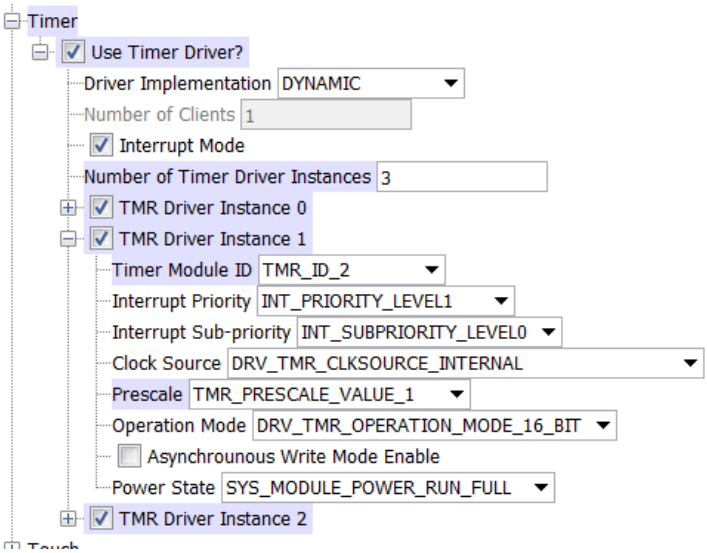


Figure 3: Timer Driver MHC Configuration



Building the Library

This section lists the files that are available in the AK4384 Codec Driver Library.

Description

This section list the files that are available in the `/src` folder of the AK4384 Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/codec/ak4384`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_ak4384.h`	Header file that exports the driver API.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_ak4384_bit_banged_control_interface.c`	This file contains implementation of the AK4384 Codec Driver with a custom bit-banged implementation for control interface driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
`/src/dynamic/drv_ak4384_virtual_control_interface.c`	This file contains implementation of the AK4384 Codec Driver with a virtual SPI driver as control interface driver. **Note:** This file is currently unsupported.
`/src/dynamic/drv_ak4384.c`	This file contains the core implementation of the AK4384 Codec Driver **Note:** This file currently unsupported.

Module Dependencies

The AK4384 Driver Library depends on the following modules:

- I2S Driver Library
- SPI Driver Library

- Timer Driver Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_AK4384_Initialize	Initializes hardware and data for the instance of the AK4384 DAC module. **Implementation:** Dynamic
	DRV_AK4384_Deinitialize	Deinitializes the specified instance of the AK4384 driver module. **Implementation:** Dynamic
	DRV_AK4384_Status	Gets the current status of the AK4384 driver module. **Implementation:** Dynamic
	DRV_AK4384_Tasks	Maintains the driver's control and data interface state machine. **Implementation:** Dynamic
	DRV_AK4384_SetAudioCommunicationMode	This function provides a run time audio format configuration

b) Client Setup Functions

	Name	Description
	DRV_AK4384_Open	Opens the specified AK4384 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_AK4384_Close	Closes an opened-instance of the AK4384 driver. **Implementation:** Dynamic

c) Codec Specific Functions

	Name	Description
	DRV_AK4384_ChannelOutputInvertDisable	Disables output polarity of the selected Channel. **Implementation:** Dynamic
	DRV_AK4384_ChannelOutputInvertEnable	Enables output polarity of the selected channel. **Implementation:** Dynamic
	DRV_AK4384_DeEmphasisFilterSet	Allows specifies enabling of digital de-emphasis filter. **Implementation:** Dynamic
	DRV_AK4384_MuteOff	Disables AK4384 output for soft mute. **Implementation:** Dynamic
	DRV_AK4384_MuteOn	Allows AK4384 output for soft mute on. **Implementation:** Dynamic
	DRV_AK4384_SamplingRateGet	This function gets the sampling rate set on the DAC AK4384. **Implementation:** Dynamic
	DRV_AK4384_SamplingRateSet	This function sets the sampling rate of the media stream. **Implementation:** Dynamic
	DRV_AK4384_SlowRollOffFilterDisable	Disables Slow Roll-off filter function. **Implementation:** Dynamic
	DRV_AK4384_SlowRollOffFilterEnable	Enables Slow Roll-off filter function. **Implementation:** Dynamic
	DRV_AK4384_VolumeGet	This function gets the volume for AK4384 Codec. **Implementation:** Dynamic
	DRV_AK4384_VolumeSet	This function sets the volume for AK4384 Codec. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectDisable	Disables AK4384 channel-independent zeros detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectEnable	Enables AK4384 channel-independent zeros detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectInvertDisable	Disables inversion of polarity for zero detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectInvertEnable	Enables inversion of polarity for zero detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectModeSet	Sets mode of AK4384 channel-independent zeros detect function. **Implementation:** Dynamic

d) Data Transfer Functions

	Name	Description
	DRV_AK4384_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_AK4384_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_AK4384_BufferCombinedQueueSizeGet	This function returns the number of bytes queued (to be processed) in the buffer queue. **Implementation:** Dynamic
	DRV_AK4384_BufferQueueFlush	This function flushes off the buffers associated with the client object. **Implementation:** Dynamic
	DRV_AK4384_BufferProcessedSizeGet	This function returns number of bytes that have been processed for the specified buffer. **Implementation:** Dynamic

e) Other Functions

	Name	Description
	DRV_AK4384_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished. **Implementation:** Dynamic
	DRV_AK4384_VersionGet	Returns the version of the AK4384 driver. **Implementation:** Dynamic
	DRV_AK4384_VersionStrGet	Returns the version of AK4384 driver in string format. **Implementation:** Dynamic

f) Data Types and Constants

	Name	Description
	DRV_AK4384_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4384_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4384_BUFFER_EVENT_HANDLER	Pointer to a AK4384 Driver Buffer Event handler function.
	DRV_AK4384_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4384_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4384_COMMAND_EVENT_HANDLER	Pointer to a AK4384 Driver Command Event Handler Function
	DRV_AK4384_DEEMPHASIS_FILTER	Identifies de-emphasis filter function.
	DRV_AK4384_INIT	Defines the data required to initialize or reinitialize the AK4384 driver.
	DRV_AK4384_MCLK_MODE	Identifies the mode of master clock to AK4384 DAC.
	DRV_AK4384_ZERO_DETECT_MODE	Identifies Zero Detect Function mode
	DRV_AK4384_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4384_COUNT	Number of valid AK4384 driver indices.
	DRV_AK4384_INDEX_0	AK4384 driver index definitions.
	DRV_AK4384_INDEX_1	This is macro DRV_AK4384_INDEX_1.
	DRV_AK4384_INDEX_2	This is macro DRV_AK4384_INDEX_2.
	DRV_AK4384_INDEX_3	This is macro DRV_AK4384_INDEX_3.
	DRV_AK4384_INDEX_4	This is macro DRV_AK4384_INDEX_4.
	DRV_AK4384_INDEX_5	This is macro DRV_AK4384_INDEX_5.

Description

This section describes the API functions of the AK4384 Codec Driver library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_AK4384_Initialize Function

Initializes hardware and data for the instance of the AK4384 DAC module.

Implementation: Dynamic

File

drv_ak4384.h

C

```
SYS_MODULE_OBJ DRV_AK4384_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the AK4384 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the 'init' parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

Remarks

This routine must be called before any other AK4384 routine is called.

This routine should only be called once during system initialization unless DRV_AK4384_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this CODEC driver. DRV_SPI_Initialize must be called if SPI driver is used for handling the control interface of this CODEC driver.

Example

```
DRV_AK4384_INIT                init;
SYS_MODULE_OBJ                 objectHandle;

init.moduleInit.value          = SYS_MODULE_POWER_RUN_FULL;
init.spiDriverModuleIndex      = DRV_SPI_INDEX_0;  // This will be ignored for a custom
                                                   // control interface driver implementation
init.i2sDriverModuleIndex      = DRV_I2S_INDEX_0;
init.mclkMode                  = DRV_AK4384_MCLK_MODE_MANUAL;
init.audioDataFormat           = DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_I2S;
init.powerDownPortChannel      = PORT_CHANNEL_G;
init.powerDownBitPosition      = PORTS_BIT_POS_15;

objectHandle = DRV_AK4384_Initialize(DRV_AK4384_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized
init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

SYS_MODULE_OBJ DRV_AK4384_Initialize

(

const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT *const init

);

DRV_AK4384_Deinitialize Function

Deinitializes the specified instance of the AK4384 driver module.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the AK4384 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_AK4384_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_AK4384_Initialize
SYS_STATUS          status;


DRV_AK4384_Deinitialize(object);

status = DRV_AK4384_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4384_Initialize routine

Function

void DRV_AK4384_Deinitialize(SYS_MODULE_OBJ object)

DRV_AK4384_Status Function

Gets the current status of the AK4384 driver module.

Implementation: Dynamic

File

drv_ak4384.h

C

```
SYS_STATUS DRV_AK4384_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This routine provides the current status of the AK4384 driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_AK4384_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4384_Initialize
SYS_STATUS          ak4384Status;

ak4384Status = DRV_AK4384_Status(object);
if (SYS_STATUS_READY == ak4384Status)
{
    // This means the driver can be opened using the
    // DRV_AK4384_Open function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4384_Initialize routine

Function

SYS_STATUS DRV_AK4384_Status(SYS_MODULE_OBJ object)

DRV_AK4384_Tasks Function

Maintains the driver's control and data interface state machine.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks function.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4384_Initialize

while (true)
{
    DRV_AK4384_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4384_Initialize)

Function

void DRV_AK4384_Tasks(SYS_MODULE_OBJ object);

DRV_AK4384_SetAudioCommunicationMode Function

This function provides a run time audio format configuration

File

drv_ak4384.h

C

```
void DRV_AK4384_SetAudioCommunicationMode(DRV_HANDLE handle, const DATA_LENGTH dl, const SAMPLE_LENGTH sl);
```

Returns

None

Description

This function sets up audio mode in I2S protocol

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
dl	Data length for I2S audio interface
sl	Left/Right Sample Length for I2S audio interface

Function

void DRV_AK4384_SetAudioCommunicationMode
(
DRV_HANDLE handle,
const DATA_LENGTH dl,
const SAMPLE_LENGTH sl
)

b) Client Setup Functions

DRV_AK4384_Open Function

Opens the specified AK4384 driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_ak4384.h

C

```
DRV_HANDLE DRV_AK4384_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Errors can occur under following conditions:

- if the number of client objects allocated via DRV_AK4384_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid
- if the ioIntent options passed are not relevant to this driver

Description

This routine opens the specified AK4384 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

Only DRV_IO_INTENT_WRITE is a valid ioIntent option as AK4384 is DAC only.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_AK4384_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

Function DRV_AK4384_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_AK4384_Open(DRV_AK4384_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

DRV_HANDLE DRV_AK4384_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

)

DRV_AK4384_Close Function

Closes an opened-instance of the AK4384 driver.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This routine closes an opened-instance of the AK4384 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_AK4384_Open before the caller may use the driver again

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_AK4384_Open

DRV_AK4384_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_Close(DRV_Handle handle)

c) Codec Specific Functions

DRV_AK4384_ChannelOutputInvertDisable Function

Disables output polarity of the selected Channel.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ChannelOutputInvertDisable(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan);
```

Returns

None.

Description

This function disables output polarity of the selected Channel.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ChannelOutputInvertDisable(myAK4384Handle, DRV_AK4384_CHANNEL_LEFT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Left or Right channel

Function

void DRV_AK4384_ChannelOutputInvertDisable(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan)

DRV_AK4384_ChannelOutputInvertEnable Function

Enables output polarity of the selected channel.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ChannelOutputInvertEnable(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan);
```

Returns

None.

Description

This function enables output polarity of the selected channel.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ChannelOutputInvertEnable(myAK4384Handle, DRV_AK4384_CHANNEL_LEFT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Left or Right channel

Function

void DRV_AK4384_ChannelOutputInvertEnable(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan)

DRV_AK4384_DeEmphasisFilterSet Function

Allows specifies enabling of digital de-emphasis filter.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_DeEmphasisFilterSet(DRV_HANDLE handle, DRV_AK4384_DEEMPHASIS_FILTER filter);
```

Returns

None.

Description

This function allows specifies enabling of digital de-emphasis for 32, 44.1 or 48 kHz sampling rates (tc = 50/15 µs)

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_DeEmphasisFilterSet(myAK4384Handle, DRV_AK4384_DEEMPHASIS_FILTER_44_1KHZ)
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
filter	Specifies Enable of de-emphasis filter

Function

void DRV_AK4384_DeEmphasisFilterSet

(

DRV_HANDLE handle,

DRV_AK4384_DEEMPHASIS_FILTER filter

)

DRV_AK4384_MuteOff Function

Disables AK4384 output for soft mute.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_MuteOff(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables AK4384 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

    DRV_AK4384_MuteOff(myAK4384Handle); //AK4384 output soft mute disabled
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_MuteOff(DRV_HANDLE handle)

DRV_AK4384_MuteOn Function

Allows AK4384 output for soft mute on.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_MuteOn(DRV_HANDLE handle);
```

Returns

None.

Description

This function Enables AK4384 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_MuteOn(myAK4384Handle);  //AK4384 output soft muted
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_MuteOn(DRV_HANDLE handle);

DRV_AK4384_SamplingRateGet Function

This function gets the sampling rate set on the DAC AK4384.

Implementation: Dynamic

File

drv_ak4384.h

C

```
uint32_t DRV_AK4384_SamplingRateGet(DRV_HANDLE handle);
```

Returns

None.

Description

This function gets the sampling rate set on the DAC AK4384.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
uint32_t baudRate;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

baudRate = DRV_AK4384_SamplingRateGet(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_AK4384_SamplingRateGet(DRV_HANDLE handle)

DRV_AK4384_SamplingRateSet Function

This function sets the sampling rate of the media stream.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

Returns

None.

Description

This function sets the media sampling rate for the client handle.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_SamplingRateSet(myAK4384Handle, 48000);  //Sets 48000 media sampling rate
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
baudRate	Baud Rate to be set

Function

void DRV_AK4384_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

DRV_AK4384_SlowRollOffFilterDisable Function

Disables Slow Roll-off filter function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_SlowRollOffFilterDisable(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables Slow Roll-off filter function. Sharp Roll-off filter function gets enabled.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_SlowRollOffFilterDisable(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_SlowRollOffFilterDisable(DRV_HANDLE handle);

DRV_AK4384_SlowRollOffFilterEnable Function

Enables Slow Roll-off filter function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_SlowRollOffFilterEnable(DRV_HANDLE handle);
```

Returns

None.

Description

This function enables Slow Roll-off filter function.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_SlowRollOffFilterEnable(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_SlowRollOffFilterEnable(DRV_HANDLE handle);

DRV_AK4384_VolumeGet Function

This function gets the volume for AK4384 Codec.

Implementation: Dynamic

File

drv_ak4384.h

C

```
uint8_t DRV_AK4384_VolumeGet(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan);
```

Returns

None.

Description

This functions gets the current volume programmed to the DAC AK4384.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

  volume = DRV_AK4384_VolumeGet(myAK4384Handle, DRV_AK4384_CHANNEL_LEFT_RIGHT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to get.

Function

uint8_t DRV_AK4384_VolumeGet(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan)

DRV_AK4384_VolumeSet Function

This function sets the volume for AK4384 Codec.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_VolumeSet(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan, uint8_t volume);
```

Returns

None.

Description

This functions sets the volume value from 0-255, which can attenuate from 0 dB to –48 dB and mute.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

    DRV_AK4384_VolumeSet(myAK4384Handle, DRV_AK4384_CHANNEL_LEFT_RIGHT, 120); //Step 120 volume
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to be set
volume	volume value from 0-255, which can attenuate from 0 dB to –48 dB and mute

Function

void DRV_AK4384_VolumeSet(DRV_HANDLE handle, DRV_AK4384_CHANNEL chan, uint8_t volume)

DRV_AK4384_ZeroDetectDisable Function

Disables AK4384 channel-independent zeros detect function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ZeroDetectDisable(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables AK4384 channel-independent zeros detect function.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ZeroDetectDisable(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_ZeroDetectDisable(DRV_HANDLE handle)

DRV_AK4384_ZeroDetectEnable Function

Enables AK4384 channel-independent zeros detect function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ZeroDetectEnable(DRV_HANDLE handle);
```

Returns

None.

Description

This function enables AK4384 channel-independent zeros detect function.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ZeroDetectEnable(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_ZeroDetectEnable(DRV_HANDLE handle)

DRV_AK4384_ZeroDetectInvertDisable Function

Disables inversion of polarity for zero detect function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ZeroDetectInvertDisable(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables inversion of polarity for zero detect function. DZF goes “H” at Zero Detection.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ZeroDetectInvertDisable(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_ZeroDetectInvertDisable(DRV_HANDLE handle)

DRV_AK4384_ZeroDetectInvertEnable Function

Enables inversion of polarity for zero detect function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ZeroDetectInvertEnable(DRV_HANDLE handle);
```

Returns

None.

Description

This function enables inversion of polarity for zero detect function. DZF goes “L” at Zero Detection

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ZeroDetectInvertEnable(myAK4384Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4384_ZeroDetectInvertEnable(DRV_HANDLE handle)

DRV_AK4384_ZeroDetectModeSet Function

Sets mode of AK4384 channel-independent zeros detect function.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_ZeroDetectModeSet(DRV_HANDLE handle, DRV_AK4384_ZERO_DETECT_MODE zdMode);
```

Returns

None.

Description

This function sets mode of AK4384 channel-independent zeros detect function

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

DRV_AK4384_ZeroDetectModeSet(myAK4384Handle, DRV_AK4384_ZERO_DETECT_MODE_ANDED);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
zdMode	Specifies zero detect function mode.

Function

void DRV_AK4384_ZeroDetectModeSet
(
DRV_HANDLE handle,
DRV_AK4384_ZERO_DETECT_MODE zdMode
)

d) Data Transfer Functions

DRV_AK4384_BufferAddWrite Function

Schedule a non-blocking driver write operation.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_BufferAddWrite(const DRV_HANDLE handle, DRV_AK4384_BUFFER_HANDLE * bufferHandle, void * buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4384_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4384_BUFFER_HANDLE_INVALID if:

- a buffer could not be allocated to the request
- the input buffer pointer is NULL
- the buffer size is '0'
- the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4384_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4384_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4384 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4384 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 device instance and the DRV_AK4384_Status must have returned SYS_STATUS_READY.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_AK4384_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4384_BUFFER_HANDLE bufferHandle;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

// Client registers an event handler with driver

DRV_AK4384_BufferEventHandlerSet(myAK4384Handle,
                APP_AK4384BufferEventHandler, (uintptr_t)&myAppObj);
```

```
DRV_AK4384_BufferAddWrite(myAK4384handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4384_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4384BufferEventHandler(DRV_AK4384_BUFFER_EVENT event,
        DRV_AK4384_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4384_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4384_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the AK4384 instance as return by the DRV_AK4384_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK4384_BufferAddWrite
(
const DRV_HANDLE handle,
DRV_AK4384_BUFFER_HANDLE *bufferHandle,
void *buffer, size_t size
)

DRV_AK4384_BufferEventHandlerSet Function

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_AK4384_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_AK4384_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4384_BUFFER_HANDLE bufferHandle;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

// Client registers an event handler with driver

DRV_AK4384_BufferEventHandlerSet(myAK4384Handle,
                APP_AK4384BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4384_BufferAddWrite(myAK4384handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4384_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4384BufferEventHandler(DRV_AK4384_BUFFER_EVENT event,
        DRV_AK4384_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4384_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4384_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.

context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4384_BufferEventHandlerSet
(
DRV_HANDLE handle,
const DRV_AK4384_BUFFER_EVENT_HANDLER eventHandler,
const uintptr_t contextHandle
)

DRV_AK4384_BufferCombinedQueueSizeGet Function

This function returns the number of bytes queued (to be processed) in the buffer queue.

Implementation: Dynamic

File

drv_ak4384.h

C

```
size_t DRV_AK4384_BufferCombinedQueueSizeGet(DRV_HANDLE handle);
```

Returns

Returns the number of the bytes that have been processed for this buffer. Returns 0 for an invalid or an expired client handle.

Description

This function returns the number of bytes queued (to be processed) in the buffer queue associated with the driver instance to which the calling client belongs. The client can use this function to know number of bytes that is in the queue to be transmitted.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

One of DRV_AK4384_BufferAddRead/DRV_AK4384_BufferAddWrite function must have been called and buffers should have been queued for transmission.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
size_t bufferQueuedSize;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4384_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_AK4384_Open function.

// Client registers an event handler with driver. This is done once

DRV_AK4384_BufferEventHandlerSet(myAK4384Handle, APP_AK4384BufferEventHandle,
                                                 (uintptr_t)&myAppObj);

DRV_AK4384_BufferAddRead(myAK4384handle,&bufferHandle,
                                 myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4384_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// The data is being processed after adding the buffer to the queue.
```

```
// The user can get to know dynamically available data in the queue to be
// transmitted by calling DRV_AK4384_BufferCombinedQueueSizeGet
bufferQueuedSize = DRV_AK4384_BufferCombinedQueueSizeGet(myAK4384Handle);
```

Parameters

Parameters	Description
handle	Opened client handle associated with a driver object.

Function

size_t DRV_AK4384_BufferCombinedQueueSizeGet(DRV_HANDLE handle)

DRV_AK4384_BufferQueueFlush Function

This function flushes off the buffers associated with the client object.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_BufferQueueFlush(const DRV_HANDLE handle);
```

Returns

None.

Description

This function flushes off the buffers associated with the client object and disables the DMA channel used for transmission.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

One of DRV_AK4384_BufferAddRead/DRV_AK4384_BufferAddWrite function must have been called and buffers should have been queued for transmission.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
size_t bufferQueuedSize;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4384_BUFFER_HANDLE bufferHandle;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

// Client registers an event handler with driver. This is done once

DRV_AK4384_BufferEventHandlerSet(myAK4384Handle, APP_AK4384BufferEventHandle,
                                                    (uintptr_t)&myAppObj);

DRV_AK4384_BufferAddRead(myAK4384handle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4384_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// The data is being processed after adding the buffer to the queue.
// The user can stop the data processing and flushoff the data
// in the queue by calling DRV_AK4384_BufferQueueFlush
DRV_AK4384_BufferQueueFlush(myAK4384Handle);
```

Parameters

Parameters	Description
handle	Opened client handle associated with a driver object.

Function

void DRV_AK4384_BufferQueueFlush(DRV_HANDLE handle)

DRV_AK4384_BufferProcessedSizeGet Function

This function returns number of bytes that have been processed for the specified buffer.

Implementation: Dynamic

File

drv_ak4384.h

C

```
size_t DRV_AK4384_BufferProcessedSizeGet(DRV_HANDLE handle);
```

Returns

Returns the number of the bytes that have been processed for this buffer. Returns 0 for an invalid or an expired buffer handle.

Description

This function returns number of bytes that have been processed for the specified buffer. The client can use this function, in a case where the buffer has terminated due to an error, to obtain the number of bytes that have been processed. If this function is called on a invalid buffer handle, or if the buffer handle has expired, the function returns 0.

Remarks

None.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified I2S driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

One of DRV_AK4384_BufferAddRead, DRV_AK4384_BufferAddWrite function must have been called and a valid buffer handle returned.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4384_BUFFER_HANDLE bufferHandle;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

// Client registers an event handler with driver. This is done once

DRV_AK4384_BufferEventHandlerSet(myAK4384Handle, APP_AK4384BufferEventHandle,
                                                    (uintptr_t)&myAppObj);

DRV_AK4384_BufferAddRead(myAK4384handle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4384_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_AK4384BufferEventHandler(DRV_AK4384_BUFFER_EVENT event,
        DRV_AK4384_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
```

```
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;

    switch(event)
    {
        case DRV_AK4384_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4384_BUFFER_EVENT_ERROR:

            // Error handling here.
            // We can find out how many bytes were processed in this
            // buffer before the error occurred.

            processedBytes = DRV_AK4384_BufferProcessedSizeGet(myAK4384Handle);

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
bufferhandle	Handle of the buffer of which the processed number of bytes to be obtained.

Function

size_t DRV_AK4384_BufferProcessedSizeGet(DRV_HANDLE handle)

e) Other Functions

DRV_AK4384_CommandEventHandlerSet Function

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

Implementation: Dynamic

File

drv_ak4384.h

C

```
void DRV_AK4384_CommandEventHandlerSet(DRV_HANDLE handle, const DRV_AK4384_COMMAND_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

When a client calls DRV_AK4384_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "AK4384 CODEC Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Preconditions

The DRV_AK4384_Initialize routine must have been called for the specified AK4384 driver instance.

DRV_AK4384_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;

// myAK4384Handle is the handle returned
// by the DRV_AK4384_Open function.

// Client registers an event handler with driver

DRV_AK4384_CommandEventHandlerSet(myAK4384Handle,
                APP_AK4384CommandEventHandler, (uintptr_t)&myAppObj);

DRV_AK4384_DeEmphasisFilterSet(myAK4384Handle, DRV_AK4384_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_AK4384CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4384_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4384_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

DRV_AK4384_VersionGet Function

Returns the version of the AK4384 driver.

Implementation: Dynamic

File

drv_ak4384.h

C

```
uint32_t DRV_AK4384_VersionGet();
```

Returns

Returns the version of AK4384 driver.

Description

The version number returned from the DRV_AK4384_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

Remarks

None.

Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

Example 2

```
uint32_t ak4384version;
ak4384version = DRV_AK4384_VersionGet();
```

Function

uint32_t DRV_AK4384_VersionGet(void)

DRV_AK4384_VersionStrGet Function

Returns the version of AK4384 driver in string format.

Implementation: Dynamic

File

drv_ak4384.h

C

```
int8_t* DRV_AK4384_VersionStrGet();
```

Returns

returns a string containing the version of AK4384 driver.

Description

The DRV_AK4384_VersionStrGet function returns a string in the format: ".[.][]" Where: is the AK4384 driver's version number. is the AK4384 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals '00'). is an optional release type ('a' for alpha, 'b' for beta not the entire word spelled out) that is not included if the release is a production version (i.e., not an alpha or beta).

The String does not contain any spaces.

Remarks

None.

Preconditions

None.

Example 1

"0.03a" "1.00"

Example 2

```
int8_t *ak4384string;
ak4384string = DRV_AK4384_VersionStrGet();
```

Function

int8_t* DRV_AK4384_VersionStrGet(void)

f) Data Types and Constants

DRV_AK4384_AUDIO_DATA_FORMAT Enumeration

Identifies the Serial Audio data interface format.

File

drv_ak4384.h

C

```
typedef enum {
  DRV_AK4384_AUDIO_DATA_FORMAT_16BIT_RIGHT_JUSTIFIED = 0,
  DRV_AK4384_AUDIO_DATA_FORMAT_20BIT_RIGHT_JUSTIFIED,
  DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_LEFT_JUSTIFIED,
  DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_I2S,
  DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_RIGHT_JUSTIFIED
} DRV_AK4384_AUDIO_DATA_FORMAT;
```

Members

Members	Description
DRV_AK4384_AUDIO_DATA_FORMAT_16BIT_RIGHT_JUSTIFIED = 0	16 bit Right Justified Audio data format
DRV_AK4384_AUDIO_DATA_FORMAT_20BIT_RIGHT_JUSTIFIED	20 bit Right Justified Audio data format
DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_LEFT_JUSTIFIED	24 bit Left Justified Audio data format
DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_I2S	24 bit I2S Audio data format
DRV_AK4384_AUDIO_DATA_FORMAT_24BIT_RIGHT_JUSTIFIED	24 bit Right Justified Audio data format

Description

AK4384 Audio data format

This enumeration identifies Serial Audio data interface format.

Remarks

None.

DRV_AK4384_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_ak4384.h

C

```
typedef enum {
  DRV_AK4384_BUFFER_EVENT_COMPLETE,
  DRV_AK4384_BUFFER_EVENT_ERROR,
  DRV_AK4384_BUFFER_EVENT_ABORT
} DRV_AK4384_BUFFER_EVENT;
```

Members

Members	Description
DRV_AK4384_BUFFER_EVENT_COMPLETE	Data was transferred successfully.
DRV_AK4384_BUFFER_EVENT_ERROR	Error while processing the request
DRV_AK4384_BUFFER_EVENT_ABORT	Data transfer aborted (Applicable in DMA mode)

Description

AK4384 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_AK4384_BufferAddWrite function.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_AK4384_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_AK4384_BUFFER_EVENT_HANDLER Type

Pointer to a AK4384 Driver Buffer Event handler function.

File

drv_ak4384.h

C

```
typedef void (* DRV_AK4384_BUFFER_EVENT_HANDLER)(DRV_AK4384_BUFFER_EVENT event, DRV_AK4384_BUFFER_HANDLE
bufferHandle, uintptr_t contextHandle);
```

Returns

None.

Description

AK4384 Driver Buffer Event Handler Function

This data type defines the required function signature for the AK4384 driver buffer event handling callback function. A client must register a pointer to a buffer event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_AK4384_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_AK4384_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_AK4384_BufferProcessedSizeGet function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4384_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver (I2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_AK4384_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void APP_MyBufferEventHandler( DRV_AK4384_BUFFER_EVENT event,
                               DRV_AK4384_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_AK4384_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_AK4384_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4384_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

File

drv_ak4384.h

C

```
typedef uintptr_t DRV_AK4384_BUFFER_HANDLE;
```

Description

AK4384 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_AK4384_BufferAddWrite function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_AK4384_CHANNEL Enumeration

Identifies Left/Right Audio channel

File

drv_ak4384.h

C

```
typedef enum {
  DRV_AK4384_CHANNEL_LEFT,
  DRV_AK4384_CHANNEL_RIGHT,
  DRV_AK4384_CHANNEL_LEFT_RIGHT,
  DRV_AK4384_NUMBER_OF_CHANNELS
} DRV_AK4384_CHANNEL;
```

Description

AK4384 Audio Channel

This enumeration identifies Left/Right Audio channel

Remarks

None.

DRV_AK4384_COMMAND_EVENT_HANDLER Type

Pointer to a AK4384 Driver Command Event Handler Function

File

drv_ak4384.h

C

```
typedef void (* DRV_AK4384_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

Returns

None.

Description

AK4384 Driver Command Event Handler Function

This data type defines the required function signature for the AK4384 driver command event handling callback function.

A command is a control instruction to the AK4384 Codec. For example, Mute ON/OFF, Zero Detect Enable/Disable, etc.

A client must register a pointer to a command event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4384_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_AK4384CommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4384_DEEMPHASIS_FILTER Enumeration

Identifies de-emphasis filter function.

File

drv_ak4384.h

C

```
typedef enum {
  DRV_AK4384_DEEMPHASIS_FILTER_44_1KHZ,
  DRV_AK4384_DEEMPHASIS_FILTER_OFF,
  DRV_AK4384_DEEMPHASIS_FILTER_48KHZ,
  DRV_AK4384_DEEMPHASIS_FILTER_32KHZ
} DRV_AK4384_DEEMPHASIS_FILTER;
```

Members

Members	Description
DRV_AK4384_DEEMPHASIS_FILTER_44_1KHZ	De-Emphasis filter for 44.1kHz.
DRV_AK4384_DEEMPHASIS_FILTER_OFF	De-Emphasis filter Off This is the default setting.
DRV_AK4384_DEEMPHASIS_FILTER_48KHZ	De-Emphasis filter for 48kHz.
DRV_AK4384_DEEMPHASIS_FILTER_32KHZ	De-Emphasis filter for 32kHz.

Description

AK4384 De-Emphasis Filter

This enumeration identifies the settings for de-emphasis filter function.

Remarks

None.

DRV_AK4384_INIT Structure

Defines the data required to initialize or reinitialize the AK4384 driver.

File

drv_ak4384.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX spiDriverModuleIndex;
  SYS_MODULE_INDEX i2sDriverModuleIndex;
  uint8_t volume;
  DRV_AK4384_MCLK_MODE mclkMode;
  bool delayDriverInitialization;
} DRV_AK4384_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX spiDriverModuleIndex;	Identifies control module(SPI) driver ID for control interface of Codec
SYS_MODULE_INDEX i2sDriverModuleIndex;	Identifies data module(I2S) driver ID for data interface of Codec
uint8_t volume;	Volume
DRV_AK4384_MCLK_MODE mclkMode;	Set MCLK mode.
bool delayDriverInitialization;	true if driver initialization should be delayed due to shared RESET pin

Description

AK4384 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the AK4384 Codec driver.

Remarks

None.

DRV_AK4384_MCLK_MODE Enumeration

Identifies the mode of master clock to AK4384 DAC.

File

drv_ak4384.h

C

```
typedef enum {
  DRV_AK4384_MCLK_MODE_MANUAL,
  DRV_AK4384_MCLK_MODE_AUTO
} DRV_AK4384_MCLK_MODE;
```

Members

Members	Description
DRV_AK4384_MCLK_MODE_MANUAL	Master clock frequency mode Manual
DRV_AK4384_MCLK_MODE_AUTO	Master clock frequency mode Auto This is the default mode.

Description

AK4384 Master clock frequency mode

This enumeration identifies mode of master clock to AK4384 DAC. In Manual Setting Mode, the sampling speed is set by setting DFS0/1 bits in Control Register 2. The frequency of MCLK at each sampling speed is set automatically. In Auto Setting Mode, the MCLK frequency is detected automatically

Remarks

None.

DRV_AK4384_ZERO_DETECT_MODE Enumeration

Identifies Zero Detect Function mode

File

drv_ak4384.h

C

```
typedef enum {
  DRV_AK4384_ZERO_DETECT_MODE_CHANNEL_SEPARATED,
  DRV_AK4384_ZERO_DETECT_MODE_ANDED
} DRV_AK4384_ZERO_DETECT_MODE;
```

Members

Members	Description
DRV_AK4384_ZERO_DETECT_MODE_CHANNEL_SEPARATED	Zero Detect channel separated. When the input data at each channel is continuously zeros for 8192 LRCK cycles, DZF pin of each channel goes to “H” This is the default mode.

DRV_AK4384_ZERO_DETECT_MODE_ANDED	Zero Detect Anded DZF pins of both channels go to “H” only when the input data at both channels are continuously zeros for 8192 LRCK cycles

Description

AK4384 Zero Detect mode

This enumeration identifies the mode of zero detect function

Remarks

None.

DRV_AK4384_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_ak4384.h

C

```
#define DRV_AK4384_BUFFER_HANDLE_INVALID ((DRV_AK4384_BUFFER_HANDLE)(-1))
```

Description

AK4384 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_AK4384_BufferAddWrite function if the buffer add request was not successful.

Remarks

None.

DRV_AK4384_COUNT Macro

Number of valid AK4384 driver indices.

File

drv_ak4384.h

C

```
#define DRV_AK4384_COUNT
```

Description

AK4384 Driver Module Count

This constant identifies the maximum number of AK4384 Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of AK4384 instances on this microcontroller.

Remarks

This value is device-specific.

DRV_AK4384_INDEX_0 Macro

AK4384 driver index definitions.

File

drv_ak4384.h

C

```
#define DRV_AK4384_INDEX_0 0
```

Description

Driver AK4384 Module Index

These constants provide AK4384 driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_AK4384_Initialize and DRV_AK4384_Open routines to identify the driver instance in use.

DRV_AK4384_INDEX_1 Macro

File

drv_ak4384.h

C

```
#define DRV_AK4384_INDEX_1 1
```

Description

This is macro DRV_AK4384_INDEX_1.

DRV_AK4384_INDEX_2 Macro

File

drv_ak4384.h

C

```
#define DRV_AK4384_INDEX_2 2
```

Description

This is macro DRV_AK4384_INDEX_2.

DRV_AK4384_INDEX_3 Macro

File

drv_ak4384.h

C

```
#define DRV_AK4384_INDEX_3 3
```

Description

This is macro DRV_AK4384_INDEX_3.

DRV_AK4384_INDEX_4 Macro

File

drv_ak4384.h

C

```
#define DRV_AK4384_INDEX_4 4
```

Description

This is macro DRV_AK4384_INDEX_4.

DRV_AK4384_INDEX_5 Macro

File

drv_ak4384.h

C

```
#define DRV_AK4384_INDEX_5 5
```

Description

This is macro DRV_AK4384_INDEX_5.

Files

Files

Name	Description
drv_ak4384.h	AK4384 Codec Driver Interface header file
drv_ak4384_config_template.h	AK4384 Codec Driver Configuration Template.

Description

This section lists the source and header files used by the AK4384Codec Driver Library.

drv_ak4384.h

AK4384 Codec Driver Interface header file

Enumerations

	Name	Description
	DRV_AK4384_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4384_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4384_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4384_DEEMPHASIS_FILTER	Identifies de-emphasis filter function.
	DRV_AK4384_MCLK_MODE	Identifies the mode of master clock to AK4384 DAC.
	DRV_AK4384_ZERO_DETECT_MODE	Identifies Zero Detect Function mode

Functions

	Name	Description
	DRV_AK4384_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_AK4384_BufferCombinedQueueSizeGet	This function returns the number of bytes queued (to be processed) in the buffer queue. **Implementation:** Dynamic
	DRV_AK4384_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_AK4384_BufferProcessedSizeGet	This function returns number of bytes that have been processed for the specified buffer. **Implementation:** Dynamic
	DRV_AK4384_BufferQueueFlush	This function flushes off the buffers associated with the client object. **Implementation:** Dynamic
	DRV_AK4384_ChannelOutputInvertDisable	Disables output polarity of the selected Channel. **Implementation:** Dynamic
	DRV_AK4384_ChannelOutputInvertEnable	Enables output polarity of the selected channel. **Implementation:** Dynamic
	DRV_AK4384_Close	Closes an opened-instance of the AK4384 driver. **Implementation:** Dynamic
	DRV_AK4384_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished. **Implementation:** Dynamic
	DRV_AK4384_DeEmphasisFilterSet	Allows specifies enabling of digital de-emphasis filter. **Implementation:** Dynamic
	DRV_AK4384_Deinitialize	Deinitializes the specified instance of the AK4384 driver module. **Implementation:** Dynamic
	DRV_AK4384_Initialize	Initializes hardware and data for the instance of the AK4384 DAC module. **Implementation:** Dynamic
	DRV_AK4384_MuteOff	Disables AK4384 output for soft mute. **Implementation:** Dynamic
	DRV_AK4384_MuteOn	Allows AK4384 output for soft mute on. **Implementation:** Dynamic

	DRV_AK4384_Open	Opens the specified AK4384 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_AK4384_SamplingRateGet	This function gets the sampling rate set on the DAC AK4384. **Implementation:** Dynamic
	DRV_AK4384_SamplingRateSet	This function sets the sampling rate of the media stream. **Implementation:** Dynamic
	DRV_AK4384_SetAudioCommunicationMode	This function provides a run time audio format configuration
	DRV_AK4384_SlowRollOffFilterDisable	Disables Slow Roll-off filter function. **Implementation:** Dynamic
	DRV_AK4384_SlowRollOffFilterEnable	Enables Slow Roll-off filter function. **Implementation:** Dynamic
	DRV_AK4384_Status	Gets the current status of the AK4384 driver module. **Implementation:** Dynamic
	DRV_AK4384_Tasks	Maintains the driver's control and data interface state machine. **Implementation:** Dynamic
	DRV_AK4384_VersionGet	Returns the version of the AK4384 driver. **Implementation:** Dynamic
	DRV_AK4384_VersionStrGet	Returns the version of AK4384 driver in string format. **Implementation:** Dynamic
	DRV_AK4384_VolumeGet	This function gets the volume for AK4384 Codec. **Implementation:** Dynamic
	DRV_AK4384_VolumeSet	This function sets the volume for AK4384 Codec. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectDisable	Disables AK4384 channel-independent zeros detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectEnable	Enables AK4384 channel-independent zeros detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectInvertDisable	Disables inversion of polarity for zero detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectInvertEnable	Enables inversion of polarity for zero detect function. **Implementation:** Dynamic
	DRV_AK4384_ZeroDetectModeSet	Sets mode of AK4384 channel-independent zeros detect function. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_AK4384_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4384_COUNT	Number of valid AK4384 driver indices.
	DRV_AK4384_INDEX_0	AK4384 driver index definitions.
	DRV_AK4384_INDEX_1	This is macro DRV_AK4384_INDEX_1.
	DRV_AK4384_INDEX_2	This is macro DRV_AK4384_INDEX_2.
	DRV_AK4384_INDEX_3	This is macro DRV_AK4384_INDEX_3.
	DRV_AK4384_INDEX_4	This is macro DRV_AK4384_INDEX_4.
	DRV_AK4384_INDEX_5	This is macro DRV_AK4384_INDEX_5.

Structures

	Name	Description
	DRV_AK4384_INIT	Defines the data required to initialize or reinitialize the AK4384 driver.

Types

	Name	Description
	DRV_AK4384_BUFFER_EVENT_HANDLER	Pointer to a AK4384 Driver Buffer Event handler function.
	DRV_AK4384_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4384_COMMAND_EVENT_HANDLER	Pointer to a AK4384 Driver Command Event Handler Function

Description

AK4384 Codec Driver Interface

The AK4384 Codec device driver interface provides a simple interface to manage the AK4384 106 dB 192 kHz 24-Bit DAC that can be interfaced Microchip Microcontroller. This file provides the interface definition for the AK4384 Codec device driver.

File Name

drv_ak4384.h

Company

Microchip Technology Inc.

drv_ak4384_config_template.h

AK4384 Codec Driver Configuration Template.

Macros

	Name	Description
	DRV_AK4384_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1, and 48K sampling frequency
	DRV_AK4384_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4384_CONTROL_CLOCK	Sets up clock frequency for the control interface (SPI)
	DRV_AK4384_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4384_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4384_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for 32, 44.1 and 48K sampling frequency
	DRV_AK4384_TIMER_DRIVER_MODULE_INDEX	Identifies the Timer Module Index for custom virtual SPI driver implementation.
	DRV_AK4384_TIMER_PERIOD	Identifies the period for the bit bang timer.

Description

AK4384 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_ak4384_config_template.h

Company

Microchip Technology Inc.

AK4642 Codec Driver Library

This topic describes the AK4642 Codec Driver Library.

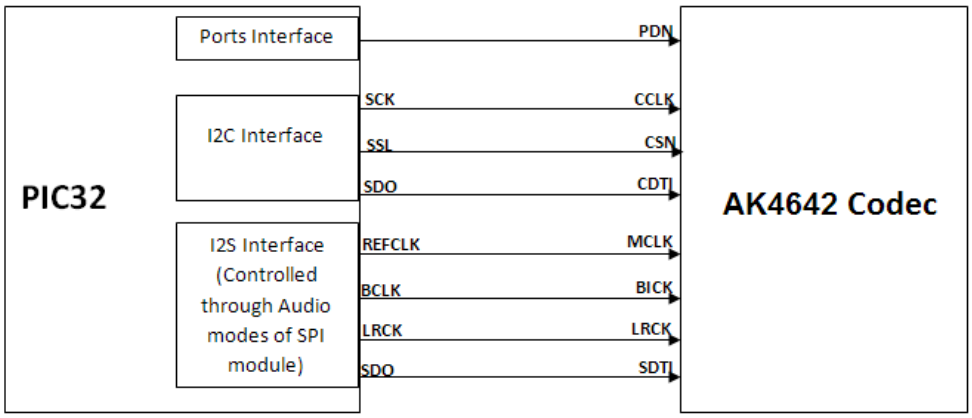
Introduction

This library provides an interface to manage the AK4642 Codec that is serially interfaced to a Microchip microcontroller for providing Audio Solutions.

Description

The AK4642 module is 16/24-bit Audio Codec from Asahi Kasei Microdevices Corporation. The AK4642 can be interfaced to Microchip microcontrollers through I2C and I2S serial interfaces. The I2C interface is used for control command transfer. The I2S interface is used for Audio data output.

A typical interface of AK4642 to a Microchip PIC32 device is provided in the following diagram:



Features

The AK4642 Codec Driver supports the following features:

- Audio Interface Format: MSB first
- ADC: 16-bit MSB justified, 16/24-bit I2S
- DAC: 16-bit MSB justified, 16bit LSB justified, 16/24-bit I2S
- Sampling Frequency Range: 8 kHz to 48 kHz
- Digital Volume Control: +12dB ~ .115dB, 0.5dB Step
- SoftMute: On and Off
- Master Clock Frequencies: 32 fs/64 fs/128fs/256fs

Using the Library

This topic describes the basic architecture of the AK4642 Codec Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_ak4642.h`

The interface to the AK4642 Codec Driver library is defined in the `drv_ak4642.h` header file. Any C language source (`.c`) file that uses the AK4642 Codec Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

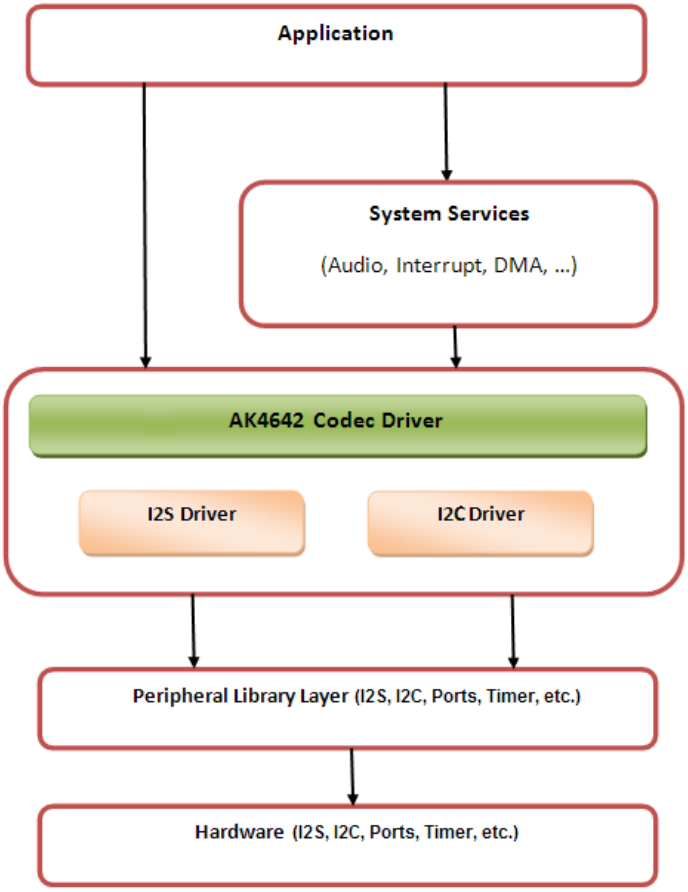
Abstraction Model

This library provides a low-level abstraction of the AK4642 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The abstraction model shown in the following diagram depicts how the AK4642 Codec Driver is positioned in the MPLAB Harmony framework. The AK4642 Codec Driver uses the SPI and I2S drivers for control and audio data transfers to the AK4642 module.

AK4642 Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The AK4642 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced AK4642 DAC module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the AK4642 Codec Driver Library.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open and close functions.
Codec Specific Functions	Provides functions that are codec specific.
Data Transfer Functions	Provides data transfer functions.
Other Functions	Provides driver specific miscellaneous functions such as sampling rate setting, control command functions, etc.
Data Types and Constants	These data types and constants are required while interacting and setting up the AK4642 Codec Driver Library.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

System Access

This topic provides information on system initialization, implementations, and provides a system access code example.

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the AK4642 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_AK4642_INIT or by using Initialization Overrides) that are supported by the specific AK4642 device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver.
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver.
- Sampling rate
- Master clock detection mode
- Power down pin port initialization

The DRV_AK4642_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ AK4642_Deinitialize, DRV_ AK4642_Status and DRV_I2S_Tasks.

Implementations

The AK4642 Codec Driver can have the following implementations:

Implementation	Description	MPLAB Harmony Components
Implementation 1	Dedicated hardware for control (I2C) and data (I2S) interface.	Standard MPLAB Harmony drivers for I2C and I2S interfaces.
Implementation 2	Dedicated hardware for data (I2S) interface. Ports pins for control interface.	Standard MPLAB Harmony drivers for I2S interface. Virtual MPLAB Harmony drivers for I2C interface.

Example:

```
DRV_AK4642_INIT drvak4642Init =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .i2sDriverModuleIndex = DRV_AK4642_I2S_DRIVER_MODULE_INDEX_IDX0,
    .i2cDriverModuleIndex = DRV_AK4642_I2C_DRIVER_MODULE_INDEX_IDX0,
    .volume = DRV_AK4642_VOLUME,
};

/*
  The I2C and I2S module index should be same as the one used in
  initializing the I2C and I2S drivers.
 */

ak4642DevObject = DRV_AK4642_Initialize(DRV_AK4642_INDEX_0, (SYS_MODULE_INIT *) &drvak4642Init);
if (SYS_MODULE_OBJ_INVALID == ak4642DevObject)
{
    // Handle error
}
```

Task Routine

The DRV_AK4642_Tasks will be called from the System Task Service.

Client Access

This topic describes client access and includes a code example.

Description

For the application to start using an instance of the module, it must call the DRV_AK4642_Open function. The DRV_AK4642_Open provides a driver handle to the AK4642 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_AK4642_Deinitialize, the application must call the DRV_AK4642_Open function again to set up the instance of the driver.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

Note: It is necessary to check the status of driver initialization before opening a driver instance. The status of the AK4642 Codec Driver can be known by calling DRV_AK4642_Status.

Example:

```
DRV_HANDLE handle;
SYS_STATUS ak4642Status;
    ak4642Status = DRV_AK4642_Status(sysObjects.ak4642DevObject);
        if (SYS_STATUS_READY == ak4642Status)
        {
               // The driver can now be opened.
               appData.ak4642Client.handle = DRV_AK4642_Open
                                            (DRV_AK4642_INDEX_0,
                                             DRV_IO_INTENT_WRITE |
                                             DRV_IO_INTENT_EXCLUSIVE );
               if(appData.ak4642Client.handle != DRV_HANDLE_INVALID)
               {
                   appData.state = APP_STATE_AK4642_SET_BUFFER_HANDLER;
               }
               else
               {
                   SYS_DEBUG(0, "Find out what's wrong \r\n");
               }
    }
    else
    {
        /* AK4642 Driver Is not ready */
                ;
    }
```

Client Operations

This topic describes client operations and provides a code example.

Description

Client operations provide the API interface for control command and audio data transfer to the AK4642 Codec.

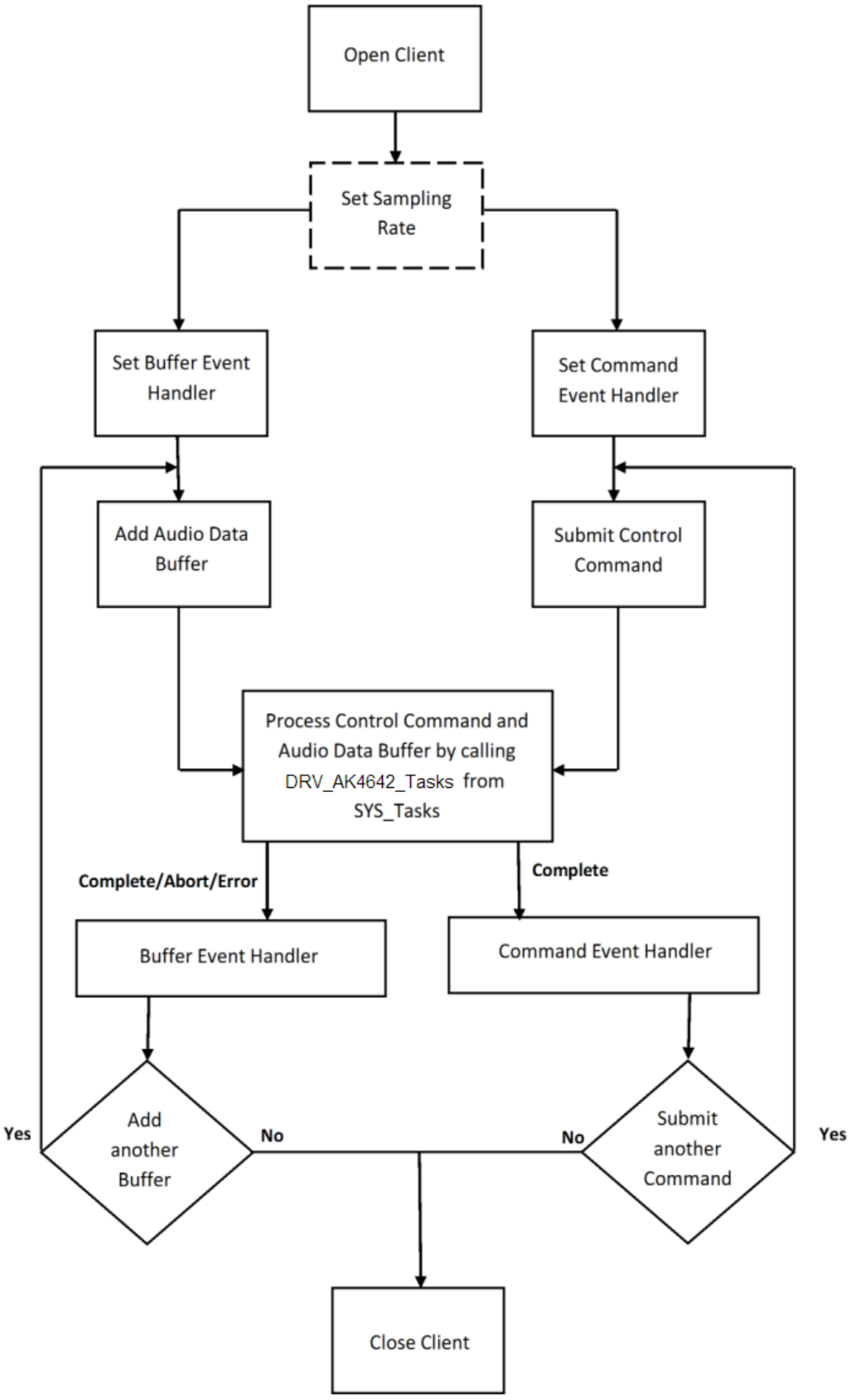
The following AK4642 Codec specific control command functions are provided:

- DRV_AK4642_SamplingRateSet
- DRV_AK4642_SamplingRateGet
- DRV_AK4642_VolumeSet
- DRV_AK4642_VolumeGet
- DRV_AK4642_MuteOn
- DRV_AK4642_MuteOff
- DRV_AK4642_IntExtMicSet
- DRV_AK4642_MonoStereoMicSet

These functions schedule a non-blocking control command transfer operation. These functions submit the control command request to the I2C Driver transmit queue, where the request is processed immediately if it is the first request, or it is processed when the previous request is complete.

DRV_AK4642_BufferAddWrite, DRV_AK4642_BufferAddRead, and DRV_AK4642_BufferAddWriteRead are buffered data operation functions. These functions schedule non-blocking audio data transfer operations. These functions add the request to the I2S Driver transmit or receive buffer queue depending on the request type, and are executed immediately if it is the first buffer, or executed later when the previous buffer is complete. The driver notifies the client with DRV_AK4642_BUFFER_EVENT_COMPLETE, DRV_AK4642_BUFFER_EVENT_ERROR, or DRV_AK4642_BUFFER_EVENT_ABORT events.

The following diagram illustrates the control commands and audio buffered data operations.



Note: It is not necessary to close and reopen the client between multiple transfers.

An application using the buffered functionality needs to perform the following steps:

1. The system should have completed necessary setup and initializations.

2. The I2S driver object should have been initialized by calling DRV_I2S_Initialize.
3. The I2C driver object should have been initialized by calling DRV_I2C_Initialize.
4. The AK4642 driver object should be initialized by calling DRV_AK4642_Initialize.
5. The necessary sampling rate value should be set up by calling DRV_AK4642_ SamplingRateSet.
6. Register buffer event handler for the client handle by calling DRV_AK4642_BufferEventHandlerSet.
7. Submit a command by calling specific command API.
8. Add a buffer to initiate the data transfer by calling DRV_AK4642_BufferAddWrite, DRV_AK4642_BufferAddRead, and DRV_AK4642_BufferAddWriteRead.
9. Call the DRV_AK4642_BufferAddWrite, DRV_AK4642_BufferAddRead, or DRV_AK4642_BufferAddWriteRead function for handling multiple buffer transmissions or receptions.
10. When the client is done, it can use DRV_AK4642_Close to close the client handle.

Example:

```
typedef enum
{
    APP_STATE_AK4642_OPEN,
    APP_STATE_AK4642_SET_BUFFER_HANDLER,
    APP_STATE_AK4642_ADD_FIRST_BUFFER_READ,
    APP_STATE_AK4642_ADD_BUFFER_OUT,
    APP_STATE_AK4642_ADD_BUFFER_IN,
    APP_STATE_AK4642_WAIT_FOR_BUFFER_COMPLETE,
} APP_STATES;

typedef struct
{
    DRV_HANDLE handle;
    DRV_AK4642_BUFFER_HANDLE writereadBufHandle;
    DRV_AK4642_BUFFER_EVENT_HANDLER bufferEventHandler;
    uintptr_t context;
    uint8_t *txbufferObject;
    uint8_t *rxbufferObject;
    size_t bufferSize;

} APP_AK4642_CLIENT;

typedef struct
{
    /* Application's current state*/
    APP_STATES state;
    /* USART client handle */
    APP_AK4642_CLIENT ak4642Client;
} APP_DATA;
APP_DATA appData;
SYS_MODULE_OBJ ak4642DevObject;
DRV_AK4642_INIT drvak4642Init =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .i2sDriverModuleIndex = DRV_AK4642_I2S_DRIVER_MODULE_INDEX_IDX0,
    .i2cDriverModuleIndex = DRV_AK4642_I2C_DRIVER_MODULE_INDEX_IDX0,
    .volume = DRV_AK4642_VOLUME,
};

void SYS_Initialize(void * data)
{
    /* Initialize Drivers */
    DRV_I2C0_Initialize();
    sysObj.drvI2S0 = DRV_I2S_Initialize(DRV_I2S_INDEX_0, (SYS_MODULE_INIT *)
                                        &drvI2S0InitData);

    sysObj.drvak4642Codec0 = DRV_AK4642_Initialize(DRV_AK4642_INDEX_0,
                             (SYS_MODULE_INIT *)&drvak4642Codec0InitData);

    /* Initialize System Services */
    SYS_INT_Initialize();
}

void APP_Tasks (void )
{
```

```
switch(appData.state)
{
    case APP_STATE_AK4642_OPEN:
    {
        SYS_STATUS status;
        status = DRV_CODEC_Status(sysObjdrvCodec0);
        if (SYS_STATUS_READY == status)
        {
        /* A client opens the driver object to get an Handle */
        appData.ak4642Client.handle = DRV_AK4642_Open(DRV_AK4642_INDEX_0,
                          DRV_IO_INTENT_WRITE|DRV_IO_INTENT_EXCLUSIVE);
        if(appData.ak4642Client.handle != DRV_HANDLE_INVALID)
        {
            appData.state = APP_STATE_AK4642_SET_BUFFER_HANDLER;
        }
        else
        {
           /* Got an Invalid Handle. Wait for AK4642 to Initialize */
        }
      }
    }
    break;

    /* Set a handler for the audio buffer completion event */
    case APP_STATE_AK4642_SET_BUFFER_HANDLER:
    {

       DRV_AK4642_BufferEventHandlerSet(appData.ak4642Client.handle,
                appData.ak4642Client.bufferEventHandler,
                appData.ak4642Client.context);

        appData.state = APP_STATE_AK4642_ADD_FIRST_BUFFER_READ;
    }
    break;

    case APP_STATE_AK4642_ADD_FIRST_BUFFER_READ:
    {
        DRV_AK4642_BufferAddWriteRead(appData.ak4642Client.handle,
                &appData.ak4642Client.writeReadBufHandle,
                appData.ak4642Client.txbufferObject,
                appData.ak4642Client.rxbufferObject,
                appData.ak4642Client.bufferSize);
        if(appData.ak4642Client.writeReadBufHandle != DRV_AK4642_BUFFER_HANDLE_INVALID)
        {
            appData.state = APP_STATE_AK4642_WAIT_FOR_BUFFER_COMPLETE;
        }
        else
        {
            SYS_DEBUG(0, "Find out what is wrong \r\n");
        }

    }
    break;
    /* Add an audio buffer to the ak4642 driver to be transmitted to
     * AK4642 CODEC */
    case APP_STATE_AK4642_ADD_BUFFER_OUT:
    {

        DRV_AK4642_BufferAddWrite(appData.ak4642Client.handle, &appData.ak4642Client.writeBufHandle,
        appData.ak4642Client.txbufferObject, appData.ak4642Client.bufferSize);
        if(appData.ak4642Client.writeBufHandle != DRV_AK4642_BUFFER_HANDLE_INVALID)
        {

            appData.state = APP_STATE_AK4642_WAIT_FOR_BUFFER_COMPLETE;
        }
        else
        {
            SYS_DEBUG(0, "Find out what is wrong \r\n");
        }
```

```
        }
        break;
        /* Add an audio buffer to the ak4642 driver to be received
         * AK4642 CODEC */
        case APP_STATE_AK4642_ADD_BUFFER_IN:
        {

            DRV_AK4642_BufferAddRead(appData.ak4642Client.handle, &appData.ak4642Client.readBufHandle,
            appData.ak4642Client.rxbufferObject, appData.ak4642Client.bufferSize);


            if(appData.ak4642Client.readBufHandle != DRV_AK4642_BUFFER_HANDLE_INVALID)
            {
                appData.state = APP_STATE_AK4642_ADD_BUFFER_OUT;
            }
            else
            {
                SYS_DEBUG(0, "Find out what is wrong \r\n");
            }
        }
        break;
        /* Audio data Transmission under process */
        case APP_STATE_AK4642_WAIT_FOR_BUFFER_COMPLETE:
        {
            /*Do nothing*/
        }
        break;

        default:
        {
        }
        break;
    }

}

/*******************************************************
 * Application AK4642 buffer Event handler.
 * This function is called back by the AK4642 driver when
 * a AK4642 data buffer RX completes.
 *******************************************************/
void APP_AK4642MicBufferEventHandler(DRV_AK4642_BUFFER_EVENT event,
        DRV_AK4642_BUFFER_HANDLE handle, uintptr_t context )
{
    static uint8_t cnt = 0;

    switch(event)
    {
        case DRV_AK4642_BUFFER_EVENT_COMPLETE:
        {

            bufnum ^= 1;

            if(bufnum ==0)
            {
                appData.ak4642Client.rxbufferObject = (uint8_t *) micbuf1;
                appData.ak4642Client.txbufferObject = (uint8_t *) micbuf2;
            }
            else if(bufnum ==1)
            {
                appData.ak4642Client.rxbufferObject = (uint8_t *) micbuf2;
                appData.ak4642Client.txbufferObject = (uint8_t *) micbuf1;
            }

                DRV_AK4642_BufferAddWriteRead(appData.ak4642Client.handle,
                &appData.ak4642Client.writeReadBufHandle,
                appData.ak4642Client.txbufferObject,
                appData.ak4642Client.rxbufferObject,
                appData.ak4642Client.bufferSize);
```

```
                appData.state = APP_STATE_AK4642_WAIT_FOR_BUFFER_COMPLETE;

            }
            break;
            case DRV_AK4642_BUFFER_EVENT_ERROR:
            {
            } break;

            case DRV_AK4642_BUFFER_EVENT_ABORT:
            {
            } break;

        }
    }

    void SYS_Tasks(void)
    {
        DRV_AK4642_Tasks(ak4642DevObject);
        APP_Tasks();
    }
```

Configuring the Library

Macros

	Name	Description
	DRV_AK4642_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4642_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4642_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4642_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4642_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4642_MCLK_SOURCE	Indicate the input clock frequency to generate the MCLK to codec.

Description

The configuration of the AK4642 Codec Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the AK4642 Codec Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the AK4642 Codec Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_AK4642_BCLK_BIT_CLK_DIVISOR Macro

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

File

drv_ak4642_config_template.h

C

```
#define DRV_AK4642_BCLK_BIT_CLK_DIVISOR
```

Description

AK4642 BCLK to LRCK Ratio to Generate Audio Stream

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

Following BCLK to LRCK ratios are supported 16bit data 16 bit channel :- 32fs, hence divisor would be 8 16bit data 32 bit channel :- 64fs, hence divisor would be 4

Remarks

None.

DRV_AK4642_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_ak4642_config_template.h

C

```
#define DRV_AK4642_CLIENTS_NUMBER DRV_AK4642_INSTANCES_NUMBER
```

Description

AK4642 Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. Typically only one client could be connected to one hardware instance. This value represents the total number of clients to be supported across all hardware instances. Therefore, if there are five AK4642 hardware interfaces, this number will be 5.

Remarks

None.

DRV_AK4642_INPUT_REFCLOCK Macro

Identifies the input REFCLOCK source to generate the MCLK to codec.

File

drv_ak4642_config_template.h

C

```
#define DRV_AK4642_INPUT_REFCLOCK
```

Description

AK4642 Input reference clock

Identifies the input REFCLOCK source to generate the MCLK to codec.

Remarks

None.

DRV_AK4642_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_ak4642_config_template.h

C

```
#define DRV_AK4642_INSTANCES_NUMBER
```

Description

AK4642 driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of AK4642 CODEC modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None.

DRV_AK4642_MCLK_SAMPLE_FREQ_MULTPLIER Macro

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

File

drv_ak4642_config_template.h

C

```
#define DRV_AK4642_MCLK_SAMPLE_FREQ_MULTPLIER
```

Description

AK4642 MCLK to LRCK Ratio to Generate Audio Stream

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency I2S sampling frequency

Supported MCLK to Sampling frequency Ratios are as below 256fs, 384fs, 512fs, 768fs or 1152fs

Remarks

None

DRV_AK4642_MCLK_SOURCE Macro

Indicate the input clock frequency to generate the MCLK to codec.

File

drv_ak4642_config_template.h

C

```
#define DRV_AK4642_MCLK_SOURCE
```

Description

AK4642 Data Interface Master Clock Speed configuration

Indicate the input clock frequency to generate the MCLK to codec.

Remarks

None.

Configuring the MHC

Provides examples on how to configure the MPLAB Harmony Configurator (MHC) for a specific driver.

Description

The following three figures show examples of MHC configurations for the AK4642 Codec Driver, I2S Driver, and the I2C Driver.

Figure 1: AK4642 Codec Driver MHC Configuration

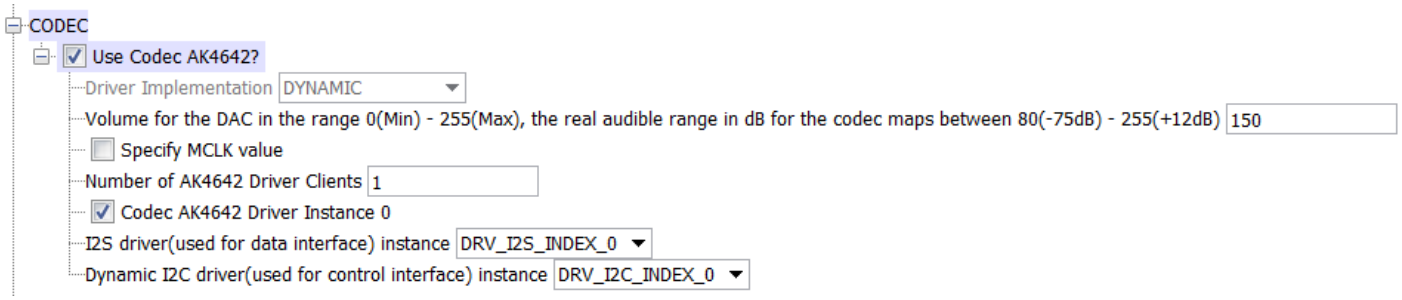


Figure 2: I2S Driver MHC Configuration

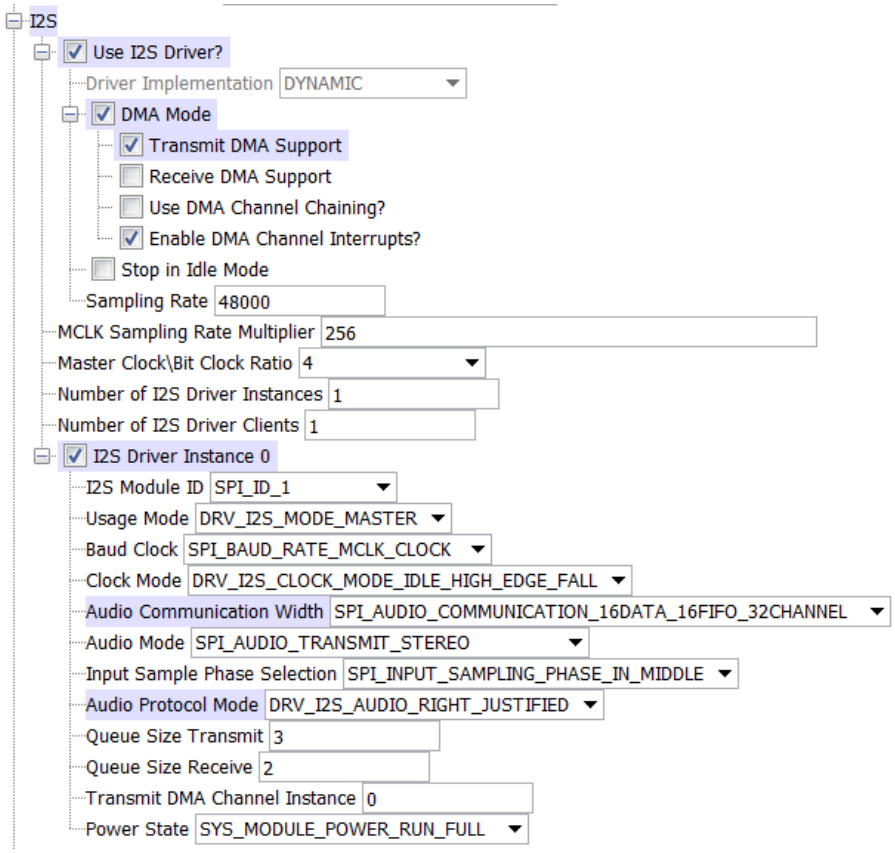
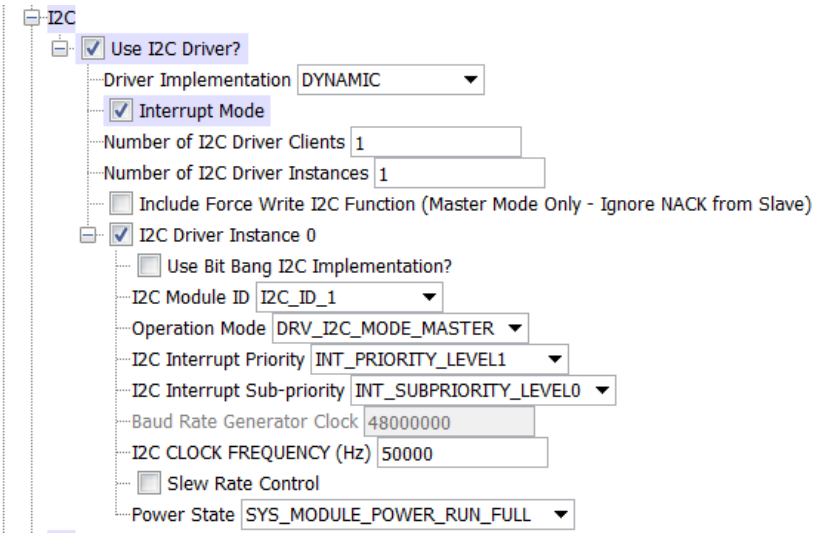


Figure 3: I2C Driver MHC Configuration



Migrating the AK4642 Driver From Earlier Versions of Microchip Harmony

Prior to version 1.08 of MPLAB Harmony, the AK4642 Codec Driver Library used the static I2C driver implementation. Beginning with v1.08 of MPLAB Harmony, applications must use the Dynamic Driver implementation with the MHC configured as shown in Figure 3. In addition, PIC32MZ configurations require the "Include Force Write I2C Function (Master Mode Only - Ignore NACK from Slave)" option to be selected.

Building the Library

This section lists the files that are available in the AK4642 Codec Driver Library.

Description

This section list the files that are available in the `/src` folder of the AK4642 Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/codec/ak4642`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_ak4642.h`	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_ak4642.c`	This file contains implementation of the AK4642 Codec Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
There are no optional files for this driver.	N/A

Module Dependencies

The AK4642 Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_AK4642_Initialize	Initializes hardware and data for the instance of the AK4642 DAC module
	DRV_AK4642_Deinitialize	Deinitializes the specified instance of the AK4642 driver module
	DRV_AK4642_Status	Gets the current status of the AK4642 driver module.
	DRV_AK4642_Tasks	Maintains the driver's control and data interface state machine.

b) Client Setup Functions

	Name	Description
	DRV_AK4642_Open	Opens the specified AK4642 driver instance and returns a handle to it
	DRV_AK4642_Close	Closes an opened-instance of the AK4642 driver

c) Codec Specific Functions

	Name	Description
	DRV_AK4642_MuteOff	This function disables AK4642 output for soft mute.
	DRV_AK4642_MuteOn	This function allows AK4642 output for soft mute on.
	DRV_AK4642_SamplingRateGet	This function gets the sampling rate set on the AK4642. **Implementation:** Dynamic
	DRV_AK4642_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_AK4642_VolumeGet	This function gets the volume for AK4642 CODEC.
	DRV_AK4642_VolumeSet	This function sets the volume for AK4642 CODEC.
	DRV_AK4642_IntExtMicSet	This function sets up the codec for the internal or the external microphone use.
	DRV_AK4642_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK4642_SetAudioCommunicationMode	This function provides a run time audio format configuration
	DRV_AK4642_MicSet	This function select the single-ended AK4642 microphone input for the AK4642 Codec

d) Data Transfer Functions

	Name	Description
	DRV_AK4642_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_AK4642_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4642_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_AK4642_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

e) Other Functions

	Name	Description
	DRV_AK4642_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.
	DRV_AK4642_VersionGet	This function returns the version of AK4642 driver
	DRV_AK4642_VersionStrGet	This function returns the version of AK4642 driver in string format.

f) Data Types and Constants

	Name	Description
	_DRV_AK4642_H	Include files.
	DRV_AK4642_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4642_COUNT	Number of valid AK4642 driver indices
	DRV_AK4642_INDEX_0	AK4642 driver index definitions
	DRV_AK4642_INDEX_1	This is macro DRV_AK4642_INDEX_1.
	DRV_AK4642_INDEX_2	This is macro DRV_AK4642_INDEX_2.
	DRV_AK4642_INDEX_3	This is macro DRV_AK4642_INDEX_3.
	DRV_AK4642_INDEX_4	This is macro DRV_AK4642_INDEX_4.
	DRV_AK4642_INDEX_5	This is macro DRV_AK4642_INDEX_5.
	DRV_AK4642_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4642_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4642_BUFFER_EVENT_HANDLER	Pointer to a AK4642 Driver Buffer Event handler function
	DRV_AK4642_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4642_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4642_COMMAND_EVENT_HANDLER	Pointer to a AK4642 Driver Command Event Handler Function
	DRV_AK4642_INIT	Defines the data required to initialize or reinitialize the AK4642 driver
	DRV_AK4642_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4642_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.
	DRV_AK4642_MIC	This is type DRV_AK4642_MIC.

Description

This section describes the API functions of the AK4642 Codec Driver library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_AK4642_Initialize Function

Initializes hardware and data for the instance of the AK4642 DAC module

File

drv_ak4642.h

C

```
SYS_MODULE_OBJ DRV_AK4642_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the AK4642 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

Remarks

This routine must be called before any other AK4642 routine is called.

This routine should only be called once during system initialization unless DRV_AK4642_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this CODEC driver. DRV_I2C_Initialize must be called if SPI driver is used for handling the control interface of this CODEC driver.

Example

```
DRV_AK4642_INIT                 init;
SYS_MODULE_OBJ                  objectHandle;

init->inUse                             = true;
init->status                            = SYS_STATUS_BUSY;
init->numClients                        = 0;
init->i2sDriverModuleIndex              = ak4642Init->i2sDriverModuleIndex;
init->i2cDriverModuleIndex              = ak4642Init->i2cDriverModuleIndex;
init->samplingRate                      = DRV_AK4642_AUDIO_SAMPLING_RATE;
init->audioDataFormat                   = DRV_AK4642_AUDIO_DATA_FORMAT_MACRO;

init->isInInterruptContext              = false;

init->commandCompleteCallback = (DRV_AK4642_COMMAND_EVENT_HANDLER)0;
init->commandContextData = 0;
init->mclk_multiplier = DRV_AK4642_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_AK4642_Initialize(DRV_AK4642_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized
init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

SYS_MODULE_OBJ DRV_AK4642_Initialize

(

const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT *const init

);

DRV_AK4642_Deinitialize Function

Deinitializes the specified instance of the AK4642 driver module

File

drv_ak4642.h

C

```
void DRV_AK4642_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the AK4642 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_AK4642_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_AK4642_Initialize
SYS_STATUS          status;


DRV_AK4642_Deinitialize(object);

status = DRV_AK4642_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4642_Initialize routine

Function

void DRV_AK4642_Deinitialize(SYS_MODULE_OBJ object)

DRV_AK4642_Status Function

Gets the current status of the AK4642 driver module.

File

drv_ak4642.h

C

```
SYS_STATUS DRV_AK4642_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This routine provides the current status of the AK4642 driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_AK4642_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4642_Initialize
SYS_STATUS          AK4642Status;
```

```
AK4642Status = DRV_AK4642_Status(object);
if (SYS_STATUS_READY == AK4642Status)
{
    // This means the driver can be opened using the
    // DRV_AK4642_Open() function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4642_Initialize routine

Function

SYS_STATUS DRV_AK4642_Status(SYS_MODULE_OBJ object)

DRV_AK4642_Tasks Function

Maintains the driver's control and data interface state machine.

File

drv_ak4642.h

C

```
void DRV_AK4642_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks() function.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4642_Initialize

while (true)
{
    DRV_AK4642_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4642_Initialize)

Function

void DRV_AK4642_Tasks(SYS_MODULE_OBJ object);

b) Client Setup Functions

DRV_AK4642_Open Function

Opens the specified AK4642 driver instance and returns a handle to it

File

drv_ak4642.h

C

```
DRV_HANDLE DRV_AK4642_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_AK4642_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

Description

This routine opens the specified AK4642 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

AK4642 can be opened with DRV_IO_INTENT_WRITE, or DRV_IO_INTENT_READ or DRV_IO_INTENT_WRITEREAD io_intent option. This decides whether the driver is used for headphone output, or microphone input or both modes simultaneously.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_AK4642_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

Function DRV_AK4642_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_AK4642_Open(DRV_AK4642_INDEX_0, DRV_IO_INTENT_WRITEREAD | DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

DRV_HANDLE DRV_AK4642_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

)

DRV_AK4642_Close Function

Closes an opened-instance of the AK4642 driver

File

drv_ak4642.h

C

```
void DRV_AK4642_Close(const DRV_HANDLE handle);
```

Returns

- None

Description

This routine closes an opened-instance of the AK4642 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_AK4642_Open before the caller may use the driver again

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_AK4642_Open

DRV_AK4642_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4642_Close(DRV_Handle handle)

c) Codec Specific Functions

DRV_AK4642_MuteOff Function

This function disables AK4642 output for soft mute.

File

drv_ak4642.h

C

```
void DRV_AK4642_MuteOff(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables AK4642 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

    DRV_AK4642_MuteOff(myAK4642Handle); //AK4642 output soft mute disabled
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4642_MuteOff(DRV_HANDLE handle)

DRV_AK4642_MuteOn Function

This function allows AK4642 output for soft mute on.

File

drv_ak4642.h

C

```
void DRV_AK4642_MuteOn(DRV_HANDLE handle);
```

Returns

None.

Description

This function Enables AK4642 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

DRV_AK4642_MuteOn(myAK4642Handle);  //AK4642 output soft muted
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4642_MuteOn(DRV_HANDLE handle);

DRV_AK4642_SamplingRateGet Function

This function gets the sampling rate set on the AK4642.

Implementation: Dynamic

File

drv_ak4642.h

C

```
uint32_t DRV_AK4642_SamplingRateGet(DRV_HANDLE handle);
```

Description

This function gets the sampling rate set on the DAC AK4642.

Remarks

None.

Example

```
uint32_t baudRate;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

baudRate = DRV_AK4642_SamplingRateGet(myAK4642Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_AK4642_SamplingRateGet(DRV_HANDLE handle)

DRV_AK4642_SamplingRateSet Function

This function sets the sampling rate of the media stream.

File

drv_ak4642.h

C

```
void DRV_AK4642_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

Returns

None.

Description

This function sets the media sampling rate for the client handle.

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

DRV_AK4642_SamplingRateSet(myAK4642Handle, 48000);  //Sets 48000 media sampling rate
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
samplingRate	Sampling frequency in Hz

Function

void DRV_AK4642_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

DRV_AK4642_VolumeGet Function

This function gets the volume for AK4642 CODEC.

File

drv_ak4642.h

C

```
uint8_t DRV_AK4642_VolumeGet(DRV_HANDLE handle, DRV_AK4642_CHANNEL channel);
```

Returns

None.

Description

This functions gets the current volume programmed to the CODEC AK4642.

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

  volume = DRV_AK4642_VolumeGet(myAK4642Handle, DRV_AK4642_CHANNEL_LEFT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
channel	argument indicating Left or Right or Both channel volume to be modified

Function

uint8_t DRV_AK4642_VolumeGet(DRV_HANDLE handle, DRV_AK4642_CHANNEL channel)

DRV_AK4642_VolumeSet Function

This function sets the volume for AK4642 CODEC.

File

drv_ak4642.h

C

```
void DRV_AK4642_VolumeSet(DRV_HANDLE handle, DRV_AK4642_CHANNEL channel, uint8_t volume);
```

Returns

None

Description

This functions sets the volume value from 0-255. The codec has DAC value to volume range mapping as :- 00 H : +12dB FF H : -115dB In order to make the volume value to dB mapping monotonically increasing from 00 to FF, re-mapping is introduced which reverses the volume value to dB mapping as well as normalizes the volume range to a more audible dB range. The current driver implementation assumes that all dB values under -60 dB are inaudible to the human ear. Re-Mapped values 00 H : -60 dB FF H : +12 dB

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

    DRV_AK4642_VolumeSet(myAK4642Handle,DRV_AK4642_CHANNEL_LEFT, 120);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
channel	argument indicating Left or Right or Both channel volume to be modified
volume	volume value specified in the range 0-255 (0x00 to 0xFF)

Function

void DRV_AK4642_VolumeSet(DRV_HANDLE handle, DRV_AK4642_CHANNEL channel, uint8_t volume);

DRV_AK4642_IntExtMicSet Function

This function sets up the codec for the internal or the external microphone use.

File

drv_ak4642.h

C

```
void DRV_AK4642_IntExtMicSet(DRV_HANDLE handle, DRV_AK4642_INT_EXT_MIC micInput);
```

Returns

None

Description

This function sets up the codec for the internal or the external microphone use.

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
micInput	INT_MIC or EXT_MIC

Function

void DRV_AK4642_IntExtMicSet(DRV_HANDLE handle,
DRV_AK4642_INT_EXT_MIC micInput);

DRV_AK4642_MonoStereoMicSet Function

This function sets up the codec for the Mono or Stereo microphone mode.

File

drv_ak4642.h

C

```
void DRV_AK4642_MonoStereoMicSet(DRV_HANDLE handle, DRV_AK4642_MONO_STEREO_MIC mono_stereo_mic);
```

Returns

None

Description

This function sets up the codec for the Mono or Stereo microphone mode.

Remarks

Currently the ak4642 codec does not work in the MONO_LEFT_CHANNEL mode. This issue will be followed up with AKM.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.
DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
mono_stereo_mic	Mono / Stereo mic setup

Function

void DRV_AK4642_MonoStereoMicSet(DRV_HANDLE handle);

DRV_AK4642_SetAudioCommunicationMode Function

This function provides a run time audio format configuration

File

drv_ak4642.h

C

```
void DRV_AK4642_SetAudioCommunicationMode(DRV_HANDLE handle, const DATA_LENGTH dl, const SAMPLE_LENGTH sl);
```

Returns

None

Description

This function sets up audio mode in I2S protocol

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.
DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
dl	Data length for I2S audio interface
sl	Left/Right Sample Length for I2S audio interface

Function

void DRV_AK4642_SetAudioCommunicationMode
(
DRV_HANDLE handle,
const DATA_LENGTH dl,
const SAMPLE_LENGTH sl
)

DRV_AK4642_MicSet Function

This function select the single-ended AK4642 microphone input for the AK4642 Codec

File

drv_ak4642.h

C

```
void DRV_AK4642_MicSet(DRV_HANDLE handle, DRV_AK4642_MIC micInput);
```

Returns

None

Description

This function selects the single-ended AK4642 microphone input for the AK4642 Codec (Where the MEMS mic is MIC1, and the external Microphone input is MIC2 on the AK4642 XC32 Daughter Board)

Remarks

None.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
micInput	MIC1 or MIC2

Function

void DRV_AK4642_MicSet(DRV_HANDLE handle, DRV_AK4642_MIC micInput);

d) Data Transfer Functions

DRV_AK4642_BufferAddWrite Function

Schedule a non-blocking driver write operation.

File

drv_ak4642.h

C

```
void DRV_AK4642_BufferAddWrite(const DRV_HANDLE handle, DRV_AK4642_BUFFER_HANDLE * bufferHandle, void *
buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4642_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4642_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4642_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4642_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4642 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4642 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 device instance and the DRV_AK4642_Status must have returned SYS_STATUS_READY.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_AK4642_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4642_BUFFER_HANDLE bufferHandle;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

// Client registers an event handler with driver

DRV_AK4642_BufferEventHandlerSet(myAK4642Handle,
                APP_AK4642BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4642_BufferAddWrite(myAK4642handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4642_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4642BufferEventHandler(DRV_AK4642_BUFFER_EVENT event,
        DRV_AK4642_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4642_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4642_BUFFER_EVENT_ERROR:
```

```
            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the AK4642 instance as return by the DRV_AK4642_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK4642_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_AK4642_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

DRV_AK4642_BufferAddRead Function

Schedule a non-blocking driver read operation.

File

drv_ak4642.h

C

```
void DRV_AK4642_BufferAddRead(const DRV_HANDLE handle, DRV_AK4642_BUFFER_HANDLE * bufferHandle, void *
buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4642_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4642_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4642_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4642_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4642 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4642 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 device instance and the DRV_AK4642_Status must have returned SYS_STATUS_READY.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_AK4642_Open call.

Parameters

Parameters	Description
handle	Handle of the AK4642 instance as return by the DRV_AK4642_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK4642_BufferAddRead
(
const DRV_HANDLE handle,
DRV_AK4642_BUFFER_HANDLE *bufferHandle,
void *buffer, size_t size
)

DRV_AK4642_BufferAddWriteRead Function

Schedule a non-blocking driver write-read operation.

Implementation: Dynamic

File

drv_ak4642.h

C

```
void DRV_AK4642_BufferAddWriteRead(const DRV_HANDLE handle, DRV_AK4642_BUFFER_HANDLE * bufferHandle, void * transmitBuffer, void * receiveBuffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4642_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4642_BUFFER_EVENT_COMPLETE:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4642_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4642_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4642 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4642 driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every AK4642 write. The transmit and receive size must be same.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 device instance and the DRV_AK4642_Status must have returned SYS_STATUS_READY.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_AK4642_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
```

```
DRV_AK4642_BUFFER_HANDLE bufferHandle;

// myak4642Handle is the handle returned
// by the DRV_AK4642_Open function.

// Client registers an event handler with driver

DRV_AK4642_BufferEventHandlerSet(myak4642Handle,
                APP_AK4642BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4642_BufferAddWriteRead(myak4642handle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_AK4642_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4642BufferEventHandler(DRV_AK4642_BUFFER_EVENT event,
        DRV_AK4642_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4642_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4642_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the AK4642 instance as returned by the DRV_AK4642_Open function
bufferHandle	Pointer to an argument that will contain the return buffer handle
transmitBuffer	The buffer where the transmit data will be stored
receiveBuffer	The buffer where the received data will be stored
size	Buffer size in bytes

Function

void DRV_AK4642_BufferAddWriteRead
(
const DRV_HANDLE handle,
DRV_AK4642_BUFFER_HANDLE *bufferHandle,
void *transmitBuffer,
void *receiveBuffer,
size_t size
)

DRV_AK4642_BufferEventHandlerSet Function

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

File

drv_ak4642.h

C

```
void DRV_AK4642_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_AK4642_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_AK4642_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4642_BUFFER_HANDLE bufferHandle;

// myAK4642Handle is the handle returned
// by the DRV_AK4642_Open function.

// Client registers an event handler with driver

DRV_AK4642_BufferEventHandlerSet(myAK4642Handle,
                APP_AK4642BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4642_BufferAddWrite(myAK4642handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4642_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4642BufferEventHandler(DRV_AK4642_BUFFER_EVENT event,
        DRV_AK4642_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4642_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4642_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;
```

```
            default:
                break;
        }
    }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4642_BufferEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4642_BUFFER_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

e) Other Functions

DRV_AK4642_CommandEventHandlerSet Function

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

File

drv_ak4642.h

C

```
void DRV_AK4642_CommandEventHandlerSet(DRV_HANDLE handle, const DRV_AK4642_COMMAND_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

When a client calls DRV_AK4642_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "AK4642 CODEC Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Preconditions

The DRV_AK4642_Initialize routine must have been called for the specified AK4642 driver instance.

DRV_AK4642_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4642_BUFFER_HANDLE bufferHandle;

// myAK4642Handle is the handle returned
```

```
// by the DRV_AK4642_Open function.

// Client registers an event handler with driver

DRV_AK4642_CommandEventHandlerSet(myAK4642Handle,
                APP_AK4642CommandEventHandler, (uintptr_t)&myAppObj);

DRV_AK4642_DeEmphasisFilterSet(myAK4642Handle, DRV_AK4642_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_AK4642CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4642_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4642_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

DRV_AK4642_VersionGet Function

This function returns the version of AK4642 driver

File

drv_ak4642.h

C

```
uint32_t DRV_AK4642_VersionGet();
```

Returns

returns the version of AK4642 driver.

Description

The version number returned from the DRV_AK4642_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

Remarks

None.

Preconditions

None.

Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

Example 2

```
uint32_t AK4642version;
AK4642version = DRV_AK4642_VersionGet();
```

Function

uint32_t DRV_AK4642_VersionGet(void)

DRV_AK4642_VersionStrGet Function

This function returns the version of AK4642 driver in string format.

File

drv_ak4642.h

C

```
int8_t* DRV_AK4642_VersionStrGet();
```

Returns

returns a string containing the version of AK4642 driver.

Description

The DRV_AK4642_VersionStrGet function returns a string in the format: ".[.][]" Where: is the AK4642 driver's version number. is the AK4642 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces. For example, "0.03a" "1.00"

Remarks

None

Preconditions

None.

Example

```
int8_t *AK4642string;
AK4642string = DRV_AK4642_VersionStrGet();
```

Function

int8_t* DRV_AK4642_VersionStrGet(void)

f) Data Types and Constants

_DRV_AK4642_H Macro

File

drv_ak4642.h

C

```
#define _DRV_AK4642_H
```

Description

Include files.

DRV_AK4642_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_ak4642.h

C

```
#define DRV_AK4642_BUFFER_HANDLE_INVALID ((DRV_AK4642_BUFFER_HANDLE)(-1))
```

Description

AK4642 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_AK4642_BufferAddWrite() and the DRV_AK4642_BufferAddRead() function if the buffer add request was not successful.

Remarks

None.

DRV_AK4642_COUNT Macro

Number of valid AK4642 driver indices

File

drv_ak4642.h

C

```
#define DRV_AK4642_COUNT
```

Description

AK4642 Driver Module Count

This constant identifies the maximum number of AK4642 Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of AK4642 instances on this microcontroller.

Remarks

This value is part-specific.

DRV_AK4642_INDEX_0 Macro

AK4642 driver index definitions

File

drv_ak4642.h

C

```
#define DRV_AK4642_INDEX_0 0
```

Description

Driver AK4642 Module Index

These constants provide AK4642 driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_AK4642_Initialize and DRV_AK4642_Open routines to identify the driver instance in use.

DRV_AK4642_INDEX_1 Macro

File

drv_ak4642.h

C

```
#define DRV_AK4642_INDEX_1 1
```

Description

This is macro DRV_AK4642_INDEX_1.

DRV_AK4642_INDEX_2 Macro

File

drv_ak4642.h

C

```
#define DRV_AK4642_INDEX_2 2
```

Description

This is macro DRV_AK4642_INDEX_2.

DRV_AK4642_INDEX_3 Macro

File

drv_ak4642.h

C

```
#define DRV_AK4642_INDEX_3 3
```

Description

This is macro DRV_AK4642_INDEX_3.

DRV_AK4642_INDEX_4 Macro

File

drv_ak4642.h

C

```
#define DRV_AK4642_INDEX_4 4
```

Description

This is macro DRV_AK4642_INDEX_4.

DRV_AK4642_INDEX_5 Macro

File

drv_ak4642.h

C

```
#define DRV_AK4642_INDEX_5 5
```

Description

This is macro DRV_AK4642_INDEX_5.

DRV_AK4642_AUDIO_DATA_FORMAT Enumeration

Identifies the Serial Audio data interface format.

File

drv_ak4642.h

C

```
typedef enum {
  DRV_AK4642_AUDIO_DATA_FORMAT_NOT_APPLICABLE = 0,
  DRV_AK4642_AUDIO_DATA_FORMAT_16BITMSB_SDTO_16BITLSB_SDTI,
  DRV_AK4642_AUDIO_DATA_FORMAT_16BITMSB_SDTO_16BITMSB_SDTI,
  DRV_AK4642_AUDIO_DATA_FORMAT_I2S
```

```
} DRV_AK4642_AUDIO_DATA_FORMAT;
```

Description

AK4642 Audio data format

This enumeration identifies Serial Audio data interface format.

DRV_AK4642_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_ak4642.h

C

```
typedef enum {
  DRV_AK4642_BUFFER_EVENT_COMPLETE,
  DRV_AK4642_BUFFER_EVENT_ERROR,
  DRV_AK4642_BUFFER_EVENT_ABORT
} DRV_AK4642_BUFFER_EVENT;
```

Members

Members	Description
DRV_AK4642_BUFFER_EVENT_COMPLETE	Data was transferred successfully.
DRV_AK4642_BUFFER_EVENT_ERROR	Error while processing the request
DRV_AK4642_BUFFER_EVENT_ABORT	Data transfer aborted (Applicable in DMA mode)

Description

AK4642 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_AK4642_BufferAddWrite() or the DRV_AK4642_BufferAddRead() function.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_AK4642_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_AK4642_BUFFER_EVENT_HANDLER Type

Pointer to a AK4642 Driver Buffer Event handler function

File

drv_ak4642.h

C

```
typedef void (* DRV_AK4642_BUFFER_EVENT_HANDLER)(DRV_AK4642_BUFFER_EVENT event, DRV_AK4642_BUFFER_HANDLE
bufferHandle, uintptr_t contextHandle);
```

Returns

None.

Description

AK4642 Driver Buffer Event Handler Function

This data type defines the required function signature for the AK4642 driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_AK4642_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_AK4642_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_AK4642_BufferProcessedSizeGet() function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4642_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver(i2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_AK4642_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void APP_MyBufferEventHandler( DRV_AK4642_BUFFER_EVENT event,
                               DRV_AK4642_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_AK4642_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_AK4642_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4642_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

File

drv_ak4642.h

C

```
typedef uintptr_t DRV_AK4642_BUFFER_HANDLE;
```

Description

AK4642 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_AK4642_BufferAddWrite() or DRV_AK4642_BufferAddRead() function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

DRV_AK4642_CHANNEL Enumeration

Identifies Left/Right Audio channel

File

drv_ak4642.h

C

```
typedef enum {
  DRV_AK4642_CHANNEL_LEFT,
  DRV_AK4642_CHANNEL_RIGHT,
  DRV_AK4642_CHANNEL_LEFT_RIGHT,
  DRV_AK4642_NUMBER_OF_CHANNELS
} DRV_AK4642_CHANNEL;
```

Description

AK4642 Audio Channel

This enumeration identifies Left/Right Audio channel

Remarks

None.

DRV_AK4642_COMMAND_EVENT_HANDLER Type

Pointer to a AK4642 Driver Command Event Handler Function

File

drv_ak4642.h

C

```
typedef void (* DRV_AK4642_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

Returns

None.

Description

AK4642 Driver Command Event Handler Function

This data type defines the required function signature for the AK4642 driver command event handling callback function.

A command is a control instruction to the AK4642 CODEC. Example Mute ON/OFF, Zero Detect Enable/Disable etc.

A client must register a pointer to a command event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4642_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_AK4642CommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4642_INIT Structure

Defines the data required to initialize or reinitialize the AK4642 driver

File

drv_ak4642.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX i2sDriverModuleIndex;
  SYS_MODULE_INDEX i2cDriverModuleIndex;
  uint32_t samplingRate;
  uint8_t volume;
  DRV_AK4642_AUDIO_DATA_FORMAT audioDataFormat;
} DRV_AK4642_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX i2sDriverModuleIndex;	Identifies data module(I2S) driver ID for data interface of CODEC
SYS_MODULE_INDEX i2cDriverModuleIndex;	Identifies data module(I2C) driver ID for control interface of CODEC
uint32_t samplingRate;	Sampling rate
uint8_t volume;	Volume
DRV_AK4642_AUDIO_DATA_FORMAT audioDataFormat;	Identifies the Audio data format

Description

AK4642 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the AK4642 CODEC driver.

Remarks

None.

DRV_AK4642_INT_EXT_MIC Enumeration

Identifies the Mic input source.

File

drv_ak4642.h

C

```
typedef enum {
  INT_MIC,
  EXT_MIC
} DRV_AK4642_INT_EXT_MIC;
```

Description

AK4642 Mic Internal / External Input

This enumeration identifies the Mic input source.

DRV_AK4642_MONO_STEREO_MIC Enumeration

Identifies the Mic input as Mono / Stereo.

File

drv_ak4642.h

C

```
typedef enum {
  ALL_ZEROS,
  MONO_RIGHT_CHANNEL,
```

```
  MONO_LEFT_CHANNEL,
  STEREO
} DRV_AK4642_MONO_STEREO_MIC;
```

Description

AK4642 Mic Mono / Stereo Input

This enumeration identifies the Mic input as Mono / Stereo.

DRV_AK4642_MIC Enumeration

File

drv_ak4642.h

C

```
typedef enum {
  MIC1 = 0,
  MIC2,
  DRV_AK4642_NUMBER_MIC
} DRV_AK4642_MIC;
```

Members

Members	Description
MIC1 = 0	INT_MIC
MIC2	EXT_MIC

Description

This is type DRV_AK4642_MIC.

Files

Files

Name	Description
drv_ak4642.h	AK4642 CODEC Driver Interface header file
drv_ak4642_config_template.h	AK4642 Codec Driver Configuration Template.

Description

This section lists the source and header files used by the AK4642 Codec Driver Library.

drv_ak4642.h

AK4642 CODEC Driver Interface header file

Enumerations

	Name	Description
	DRV_AK4642_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4642_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4642_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4642_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4642_MIC	This is type DRV_AK4642_MIC.
	DRV_AK4642_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.

Functions

	Name	Description
	DRV_AK4642_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4642_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_AK4642_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_AK4642_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

	DRV_AK4642_Close	Closes an opened-instance of the AK4642 driver
	DRV_AK4642_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.
	DRV_AK4642_Deinitialize	Deinitializes the specified instance of the AK4642 driver module
	DRV_AK4642_Initialize	Initializes hardware and data for the instance of the AK4642 DAC module
	DRV_AK4642_IntExtMicSet	This function sets up the codec for the internal or the external microphone use.
	DRV_AK4642_MicSet	This function select the single-ended AK4642 microphone input for the AK4642 Codec
	DRV_AK4642_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK4642_MuteOff	This function disables AK4642 output for soft mute.
	DRV_AK4642_MuteOn	This function allows AK4642 output for soft mute on.
	DRV_AK4642_Open	Opens the specified AK4642 driver instance and returns a handle to it
	DRV_AK4642_SamplingRateGet	This function gets the sampling rate set on the AK4642. **Implementation:** Dynamic
	DRV_AK4642_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_AK4642_SetAudioCommunicationMode	This function provides a run time audio format configuration
	DRV_AK4642_Status	Gets the current status of the AK4642 driver module.
	DRV_AK4642_Tasks	Maintains the driver's control and data interface state machine.
	DRV_AK4642_VersionGet	This function returns the version of AK4642 driver
	DRV_AK4642_VersionStrGet	This function returns the version of AK4642 driver in string format.
	DRV_AK4642_VolumeGet	This function gets the volume for AK4642 CODEC.
	DRV_AK4642_VolumeSet	This function sets the volume for AK4642 CODEC.

Macros

	Name	Description
	_DRV_AK4642_H	Include files.
	DRV_AK4642_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4642_COUNT	Number of valid AK4642 driver indices
	DRV_AK4642_INDEX_0	AK4642 driver index definitions
	DRV_AK4642_INDEX_1	This is macro DRV_AK4642_INDEX_1.
	DRV_AK4642_INDEX_2	This is macro DRV_AK4642_INDEX_2.
	DRV_AK4642_INDEX_3	This is macro DRV_AK4642_INDEX_3.
	DRV_AK4642_INDEX_4	This is macro DRV_AK4642_INDEX_4.
	DRV_AK4642_INDEX_5	This is macro DRV_AK4642_INDEX_5.

Structures

	Name	Description
	DRV_AK4642_INIT	Defines the data required to initialize or reinitialize the AK4642 driver

Types

	Name	Description
	DRV_AK4642_BUFFER_EVENT_HANDLER	Pointer to a AK4642 Driver Buffer Event handler function
	DRV_AK4642_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4642_COMMAND_EVENT_HANDLER	Pointer to a AK4642 Driver Command Event Handler Function

Description

AK4642 CODEC Driver Interface

The AK4642 CODEC device driver interface provides a simple interface to manage the AK4642 16/24-Bit CODEC that can be interfaced Microchip Microcontroller. This file provides the interface definition for the AK4642 CODEC device driver.

File Name

drv_ak4642.h

Company

Microchip Technology Inc.

drv_ak4642_config_template.h

AK4642 Codec Driver Configuration Template.

Macros

	Name	Description
	DRV_AK4642_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4642_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4642_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4642_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4642_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4642_MCLK_SOURCE	Indicate the input clock frequency to generate the MCLK to codec.

Description

AK4642 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_ak4642_config_template.h

Company

Microchip Technology Inc.

AK4953 Codec Driver Library

This topic describes the AK4953 Codec Driver Library.

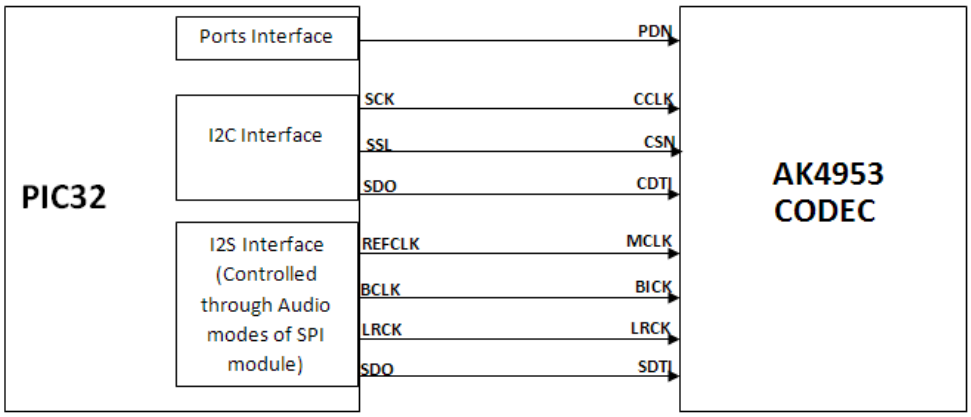
Introduction

This library provides an interface to manage the AK4953 Codec that is serially interfaced to a Microchip microcontroller for providing Audio Solutions.

Description

The AK4953 module is 16/24-bit Audio Codec from Asahi Kasei Microdevices Corporation. The AK4953 can be interfaced to Microchip microcontrollers through I2C and I2S serial interfaces. The I2C interface is used for control command transfer. The I2S interface is used for Audio data output.

A typical interface of AK4953 to a Microchip PIC32 device is provided in the following diagram:



Features

The AK4953 Codec supports the following features:

- Audio Interface Format: MSB first
- ADC: 24-bit MSB justified, 16/24-bit I2S

- DAC: 24-bit MSB justified, 1-6bit LSB justified, 24-bit LSB justified, 16/24-bit I2S
- Sampling Frequency Range: 8 kHz to 192 kHz
- Digital Volume Control: +12dB ~ .115dB, 0.5dB Step
- SoftMute: On and Off
- Master Clock Frequencies: 32 fs/64 fs/128 fs/256 fs

Using the Library

This topic describes the basic architecture of the AK4953 Codec Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_AK4953.h`

The interface to the AK4953 Codec Driver library is defined in the `drv_AK4953.h` header file. Any C language source (`.c`) file that uses the AK4953 Codec Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The AK4953 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced AK4953 DAC module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the AK4953 Codec Driver Library.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Status Functions	Provides status functions.
Other Functions	Provides driver specific miscellaneous functions such as sampling rate setting, control command functions, etc.
Data Types and Constants	These data types and constants are required while interacting and setting up the AK4953 Codec Driver Library.

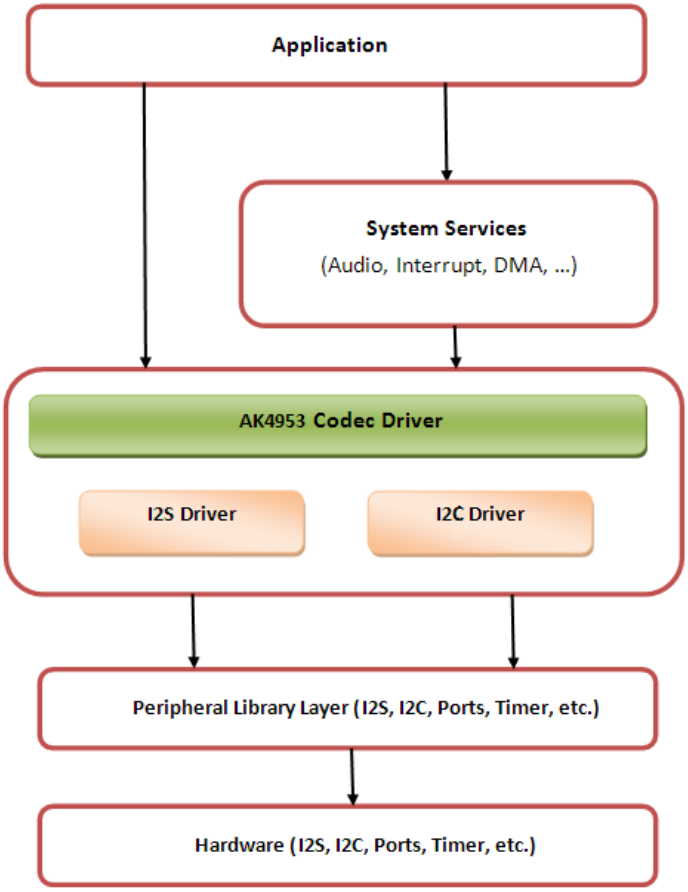
Abstraction Model

This library provides a low-level abstraction of the AK4953 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The abstraction model shown in the following diagram depicts how the AK4953 Codec Driver is positioned in the MPLAB Harmony framework. The AK4953 Codec Driver uses the SPI and I2S drivers for control and audio data transfers to the AK4953 module.

AK4953 Driver Abstraction Model



How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

System Access

This topic describes system initialization, implementations, and includes a system access code example.

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the AK4953 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_AK4953_INIT or by using Initialization Overrides) that are supported by the specific AK4953 device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver.
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver.
- Sampling rate
- Audio data format. The audio data format should match with the audio data format settings done in I2S driver initialization
- Power down pin port initialization
- Queue size for the audio data transmit buffer

The DRV_AK4953_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ AK4953_Deinitialize, DRV_ AK4953_Status and DRV_I2S_Tasks.

Implementations

The AK4953 Codec Driver can has the following implementation:

Description	MPLAB Harmony Components
Dedicated hardware for control (I2C) and data (I2S) interface.	Standard MPLAB Harmony drivers for I2C and I2S interfaces.

Example:

```
DRV_AK4953_INIT drvak4953Codec0InitData =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .i2sDriverModuleIndex = DRV_AK4953_I2S_DRIVER_MODULE_INDEX_IDX0,
    .i2cDriverModuleIndex = DRV_AK4953_I2C_DRIVER_MODULE_INDEX_IDX0,
    .volume = DRV_AK4953_VOLUME,
    .queueSizeTransmit = DRV_AK4953_TRANSMIT_QUEUE_SIZE,
};

// Initialize the I2C driver
   DRV_I2C0_Initialize();

// Initialize the I2S driver. The I2S module index should be same as the one used in initializing
// the I2S driver.
   sysObj.drvI2S0 = DRV_I2S_Initialize(DRV_I2S_INDEX_0, (SYS_MODULE_INIT *)&drvI2S0InitData);

// Initialize the Codec driver
   sysObj.drvak4953Codec0 = DRV_AK4953_Initialize(DRV_AK4953_INDEX_0, (SYS_MODULE_INIT
*)&drvak4953Codec0InitData);

if (SYS_MODULE_OBJ_INVALID == AK4953DevObject)
{
// Handle error
}
```

Task Routine

The DRV_AK4953_Tasks will be called from the System Task Service.

Client Access

For the application to start using an instance of the module, it must call the DRV_AK4953_Open function. The DRV_AK4953_Open provides a driver handle to the AK4953 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_AK4953_Deinitialize, the application must call the DRV_AK4953_Open function again to set up the instance of the driver.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

Client Operations

This topic provides information on client operations and includes a control command and audio buffered data operation flow diagram.

Description

Client operations provide the API interface for control command and audio data transfer to the AK4953 Codec.

The following AK4953 Codec specific control command functions are provided:

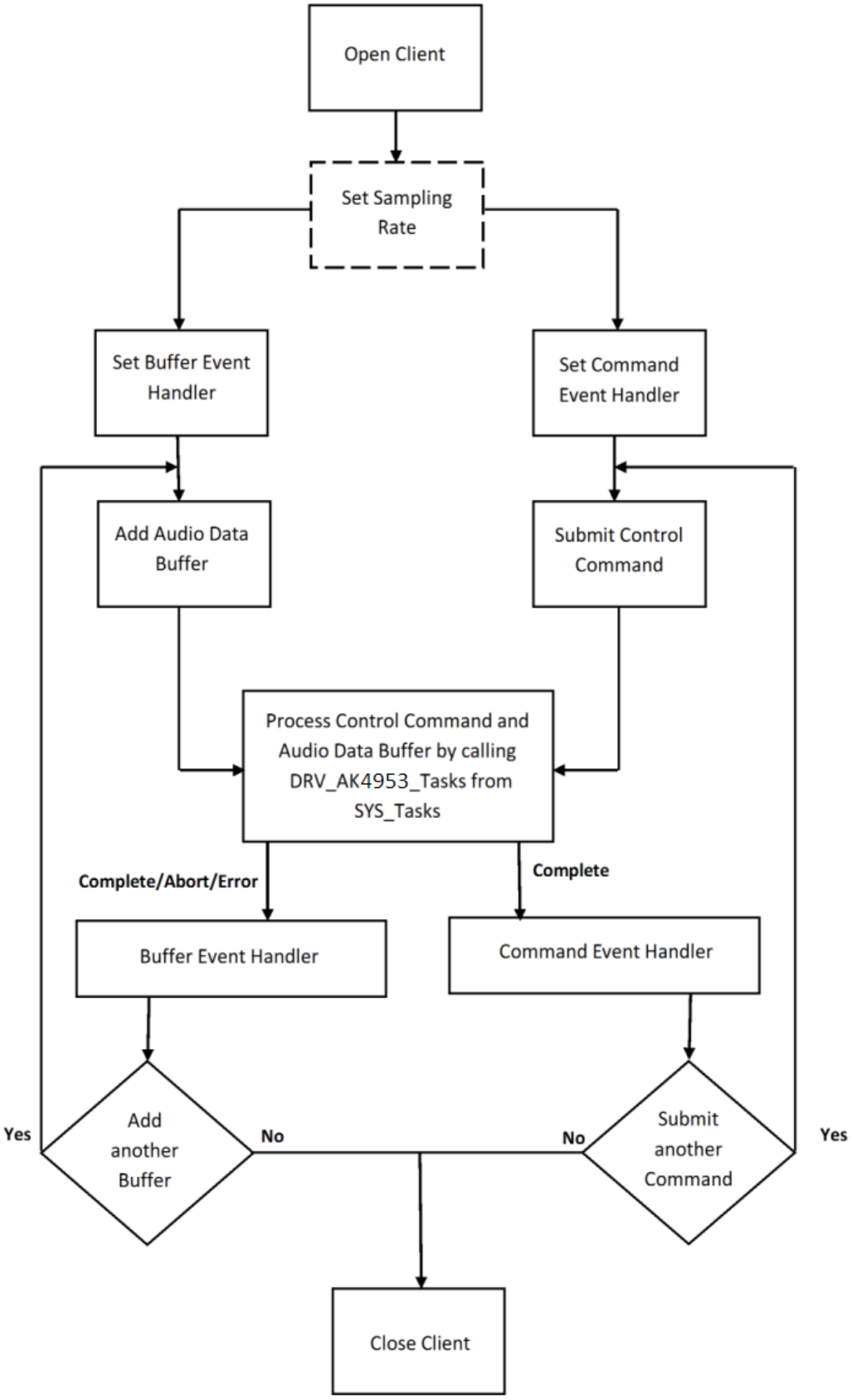
- DRV_AK4953_SamplingRateSet
- DRV_AK4953_SamplingRateGet
- DRV_AK4953_VolumeSet
- DRV_AK4953_VolumeGet
- DRV_AK4953_MuteOn
- DRV_AK4953_MuteOff
- DRV_AK4953_IntExtMicSet
- DRV_AK4953_MonoStereoMicSet

These functions schedule a non-blocking control command transfer operation. These functions submit the control command request to the AK4953 Codec. These functions submit the control command request to I2C Driver transmit queue, the request is processed immediately if it is the first request, or processed when the previous request is complete.

DRV_AK4953_BufferAddWrite, DRV_AK4953_BufferAddRead, and DRV_AK4953_BufferAddWriteRead are buffered data operation functions.

These functions schedule non-blocking audio data transfer operations. These functions add the request to I2S Driver transmit or receive buffer queue depends on the request type, and are executed immediately if it is the first buffer, or executed later when the previous buffer is complete. The driver notifies the client with DRV_AK4953_BUFFER_EVENT_COMPLETE, DRV_AK4953_BUFFER_EVENT_ERROR, or DRV_AK4953_BUFFER_EVENT_ABORT events.

The following diagram illustrates the control commands and audio buffered data operations.



Note: It is not necessary to close and reopen the client between multiple transfers.

Configuring the Library

Macros

	Name	Description
	DRV_AK4953_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4953_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4953_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4953_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4953_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4953_MCLK_SOURCE	Indicate the input clock frequency to generate the MCLK to codec.
	DRV_AK4953_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.

Description

The configuration of the AK4953 Codec Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the AK4953 Codec Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the AK4953 Codec Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_AK4953_BCLK_BIT_CLK_DIVISOR Macro

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_BCLK_BIT_CLK_DIVISOR
```

Description

AK4953 BCLK to LRCK Ratio to Generate Audio Stream

Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

Following BCLK to LRCK ratios are supported 16bit data 16 bit channel :- 32fs, hence divisor would be 8 16bit data 32 bit channel :- 64fs, hence divisor would be 4

Remarks

None.

DRV_AK4953_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_CLIENTS_NUMBER DRV_AK4953_INSTANCES_NUMBER
```

Description

AK4953 Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. Typically only one client could be connected to one hardware instance. This value represents the total number of clients to be supported across all hardware instances. Therefore, if there are five AK4953 hardware interfaces, this number will be 5.

Remarks

None.

DRV_AK4953_INPUT_REFCLOCK Macro

Identifies the input REFCLOCK source to generate the MCLK to codec.

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_INPUT_REFCLOCK
```

Description

AK4953 Input reference clock

Identifies the input REFCLOCK source to generate the MCLK to codec.

Remarks

None.

DRV_AK4953_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_INSTANCES_NUMBER
```

Description

AK4953 driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of AK4953 CODEC modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None.

DRV_AK4953_MCLK_SAMPLE_FREQ_MULTPLIER Macro

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_MCLK_SAMPLE_FREQ_MULTPLIER
```

Description

AK4953 MCLK to LRCK Ratio to Generate Audio Stream

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency I2S sampling frequency

Supported MCLK to Sampling frequency Ratios are as below 256fs, 384fs, 512fs, 768fs or 1152fs

Remarks

None

DRV_AK4953_MCLK_SOURCE Macro

Indicate the input clock frequency to generate the MCLK to codec.

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_MCLK_SOURCE
```

Description

AK4953 Data Interface Master Clock Speed configuration

Indicate the input clock frequency to generate the MCLK to codec.

Remarks

None.

DRV_AK4953_QUEUE_DEPTH_COMBINED Macro

Number of entries of all queues in all instances of the driver.

File

drv_ak4953_config_template.h

C

```
#define DRV_AK4953_QUEUE_DEPTH_COMBINED
```

Description

AK4953 Driver Buffer Queue Entries

This macro defined the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for transmit operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build). The hardware instance transmit buffer queue will queue transmit buffers submitted by the DRV_AK4953_BufferAddWrite function.

A buffer queue will contains buffer queue entries, each related to a BufferAdd request. This configuration macro defines total number of buffer entries that will be available for use between all AK4953 driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking write requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its transmit buffer queue size.

As an example, consider the case of static single client driver application where full duplex non blocking operation is desired without queuing, the minimum transmit queue depth and minimum receive queue depth should be 1. Hence the total number of buffer entries should be 2.

As an example, consider the case of a dynamic driver (say two instances) where instance one will queue up to three write requests and up to two read requests, and instance two will queue up to two write requests and up to six read requests, the value of this macro should be 13 (2 + 3 + 2 + 6).

Configuring the MHC

Provides examples on how to configure the MPLAB Harmony Configurator (MHC) for a specific driver.

Description

The following three figures show examples of MHC configurations for the AK4953 Codec Driver, I2S Driver, and the I2C Driver.

Figure 1: AK4953 Codec Driver MHC Configuration

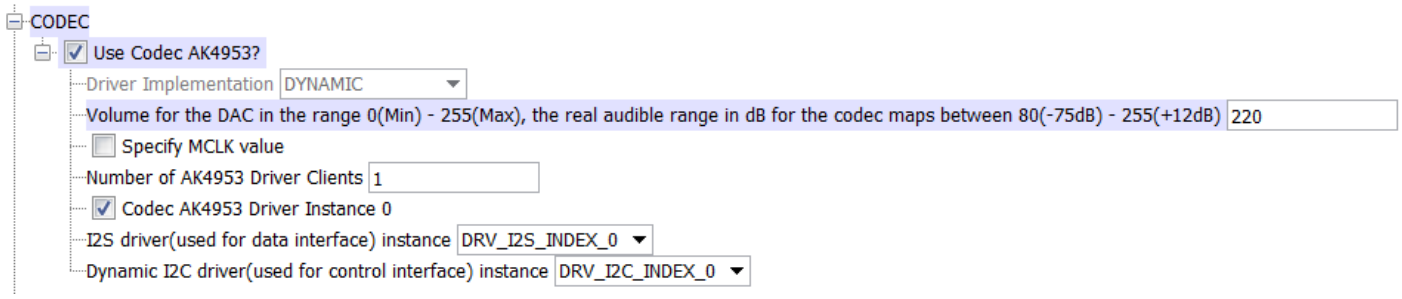


Figure 2: I2S Driver MHC Configuration

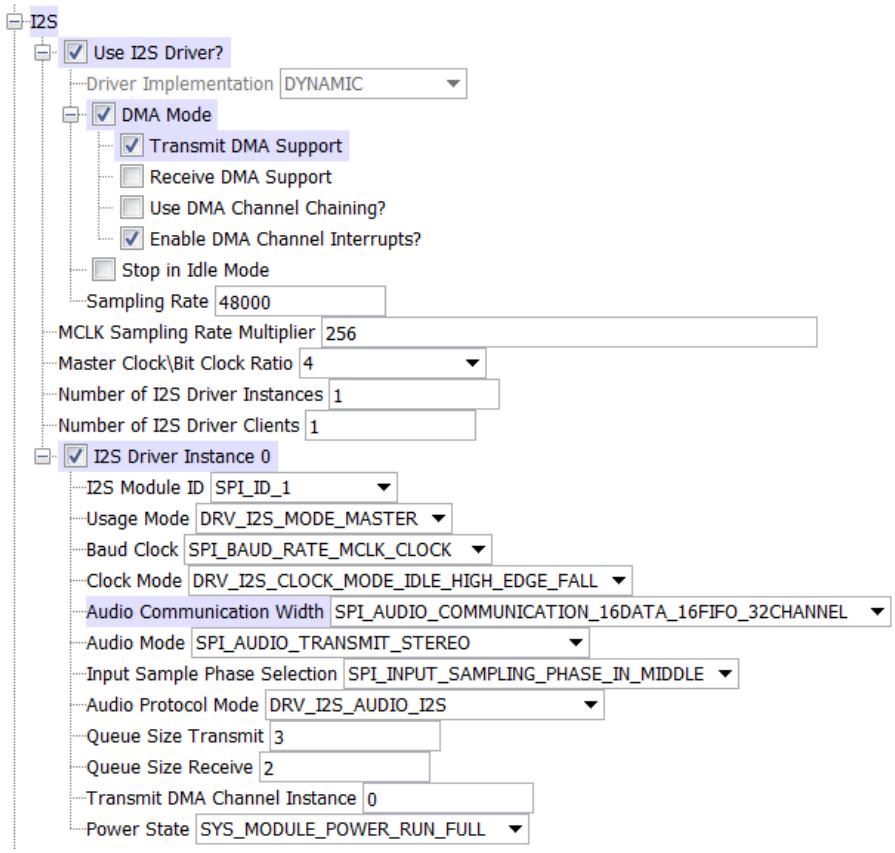
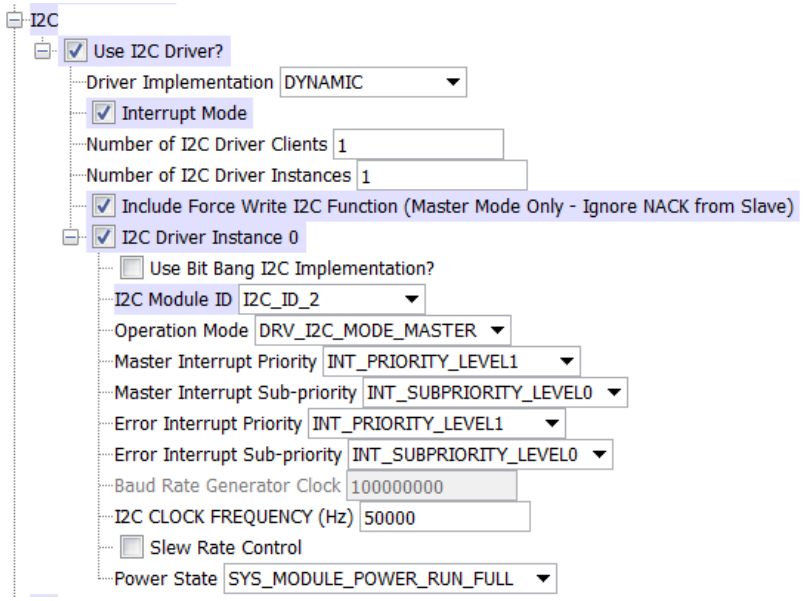


Figure 3: I2C Driver MHC Configuration



Migrating the AK4953 Driver From Earlier Versions of Microchip Harmony

Prior to version 1.08 of MPLAB Harmony, the AK4953 Codec Driver Library used the static I2C driver implementation. Beginning with v1.08 of MPLAB Harmony, applications must use the Dynamic Driver implementation with the MHC configured as shown in Figure 3. In addition, PIC32MZ configurations require the "Include Force Write I2C Function (Master Mode Only - Ignore NACK from Slave)" option to be selected.

Building the Library

This section lists the files that are available in the AK4953 Codec Driver Library.

Description

This section list the files that are available in the `/src` folder of the AK4953 Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/codec/ak4953`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_ak4953.h	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_ak4953.c	This file contains implementation of the AK4953 Codec Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The AK4953 Codec Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_AK4953_Initialize	Initializes hardware and data for the instance of the AK4953 DAC module. **Implementation:** Dynamic
	DRV_AK4953_Deinitialize	Deinitializes the specified instance of the AK4953 driver module. **Implementation:** Dynamic
	DRV_AK4953_Open	Opens the specified AK4953 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_AK4953_Close	Closes an opened-instance of the AK4953 driver. **Implementation:** Dynamic
	DRV_AK4953_Tasks	Maintains the driver's control and data interface state machine. **Implementation:** Dynamic
	DRV_AK4953_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished. **Implementation:** Dynamic
	DRV_AK4953_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_AK4953_SamplingRateSet	This function sets the sampling rate of the media stream. **Implementation:** Dynamic
	DRV_AK4953_SetAudioCommunicationMode	This function provides a run time audio format configuration

b) Status Functions

	Name	Description
	DRV_AK4953_SamplingRateGet	This function gets the sampling rate set on the DAC AK4953. **Implementation:** Dynamic
	DRV_AK4953_Status	Gets the current status of the AK4953 driver module. **Implementation:** Dynamic
	DRV_AK4953_VersionGet	This function returns the version of AK4953 driver. **Implementation:** Dynamic
	DRV_AK4953_VersionStrGet	This function returns the version of AK4953 driver in string format. **Implementation:** Dynamic
	DRV_AK4953_VolumeGet	This function gets the volume for AK4953 CODEC. **Implementation:** Dynamic

c) Other Functions

	Name	Description
	DRV_AK4953_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_AK4953_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_AK4953_MuteOff	This function disables AK4953 output for soft mute. **Implementation:** Dynamic
	DRV_AK4953_MuteOn	This function allows AK4953 output for soft mute on. **Implementation:** Dynamic
	DRV_AK4953_VolumeSet	This function sets the volume for AK4953 CODEC. **Implementation:** Dynamic
	DRV_AK4953_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4953_IntExtMicSet	This function sets up the codec for the X32 DB internal or the external microphone use.
	DRV_AK4953_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK4953_MicSet	This function sets up the codec for the internal or the AK4953 Mic1 or Mic2 input.

d) Data Types and Constants

	Name	Description
	DRV_AK4953_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4953_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4953_BUFFER_EVENT_HANDLER	Pointer to a AK4953 Driver Buffer Event handler function
	DRV_AK4953_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4953_COMMAND_EVENT_HANDLER	Pointer to a AK4953 Driver Command Event Handler Function
	DRV_AK4953_DIGITAL_BLOCK_CONTROL	Identifies Bass-Boost Control function
	DRV_AK4953_INIT	Defines the data required to initialize or reinitialize the AK4953 driver
	_DRV_AK4953_H	Include files.
	DRV_AK4953_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4953_COUNT	Number of valid AK4953 driver indices
	DRV_AK4953_INDEX_0	AK4953 driver index definitions
	DRV_AK4953_INDEX_1	This is macro DRV_AK4953_INDEX_1.
	DRV_AK4953_INDEX_2	This is macro DRV_AK4953_INDEX_2.
	DRV_AK4953_INDEX_3	This is macro DRV_AK4953_INDEX_3.
	DRV_AK4953_INDEX_4	This is macro DRV_AK4953_INDEX_4.
	DRV_AK4953_INDEX_5	This is macro DRV_AK4953_INDEX_5.
	DRV_AK4953_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4953_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4953_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.
	DRV_AK4953_MIC	This is type DRV_AK4953_MIC.

Description

This section describes the API functions of the AK4953 Codec Driver library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_AK4953_Initialize Function

Initializes hardware and data for the instance of the AK4953 DAC module.

Implementation: Dynamic

File

drv_ak4953.h

C

```
SYS_MODULE_OBJ DRV_AK4953_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the AK4953 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

Remarks

This routine must be called before any other AK4953 routine is called.

This routine should only be called once during system initialization unless DRV_AK4953_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this CODEC driver. Also DRV_I2C_Initialize must be called before calling this function to initialize the control interface of this CODEC driver.

Example

```
DRV_AK4953_INIT                 init;
SYS_MODULE_OBJ                  objectHandle;

init->inUse                             = true;
init->status                            = SYS_STATUS_BUSY;
init->numClients                        = 0;
init->i2sDriverModuleIndex              = ak4953Init->i2sDriverModuleIndex;
init->i2cDriverModuleIndex              = ak4953Init->i2cDriverModuleIndex;
init->samplingRate                      = DRV_AK4953_AUDIO_SAMPLING_RATE;
init->audioDataFormat                   = DRV_AK4953_AUDIO_DATA_FORMAT_MACRO;
for(index=0; index < DRV_AK4953_NUMBER_OF_CHANNELS; index++)
{
    init->volume[index] = ak4953Init->volume;
}
init->isInInterruptContext              = false;

init->commandCompleteCallback = (DRV_AK4953_COMMAND_EVENT_HANDLER)0;
init->commandContextData = 0;

init->mclk_multiplier = DRV_AK4953_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_AK4953_Initialize(DRV_AK4953_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized

init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

SYS_MODULE_OBJ DRV_AK4953_Initialize
(
const SYS_MODULE_INDEX drvIndex,
const SYS_MODULE_INIT *const init
);

DRV_AK4953_Deinitialize Function

Deinitializes the specified instance of the AK4953 driver module.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the AK4953 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_AK4953_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_AK4953_Initialize
SYS_STATUS          status;


DRV_AK4953_Deinitialize(object);

status = DRV_AK4953_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4953_Initialize routine

Function

void DRV_AK4953_Deinitialize(SYS_MODULE_OBJ object)

DRV_AK4953_Open Function

Opens the specified AK4953 driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_ak4953.h

C

```
DRV_HANDLE DRV_AK4953_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_AK4953_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

Description

This routine opens the specified AK4953 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

AK4953 can be opened with DRV_IO_INTENT_WRITE, or DRV_IO_INTENT_READ or DRV_IO_INTENT_WRITEREAD io_intent option. This decides whether the driver is used for headphone output, or microphone input or both modes simultaneously.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_AK4953_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

Function DRV_AK4953_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_AK4953_Open(DRV_AK4953_INDEX_0, DRV_IO_INTENT_WRITEREAD | DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

DRV_HANDLE DRV_AK4953_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

)

DRV_AK4953_Close Function

Closes an opened-instance of the AK4953 driver.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This routine closes an opened-instance of the AK4953 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_AK4953_Open before the caller may use the driver again

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_AK4953_Open

DRV_AK4953_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4953_Close(DRV_Handle handle)

DRV_AK4953_Tasks Function

Maintains the driver's control and data interface state machine.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks() function.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4953_Initialize

while (true)
{
    DRV_AK4953_Tasks (object);
```

```
    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4953_Initialize)

Function

void DRV_AK4953_Tasks(SYS_MODULE_OBJ object);

DRV_AK4953_CommandEventHandlerSet Function

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_CommandEventHandlerSet(DRV_HANDLE handle, const DRV_AK4953_COMMAND_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

When a client calls DRV_AK4953_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "AK4953 CODEC Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4953_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

// Client registers an event handler with driver

DRV_AK4953_CommandEventHandlerSet(myAK4953Handle,
                APP_AK4953CommandEventHandler, (uintptr_t)&myAppObj);

DRV_AK4953_DeEmphasisFilterSet(myAK4953Handle, DRV_AK4953_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_AK4953CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.
```

```
    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4953_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4953_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

DRV_AK4953_BufferEventHandlerSet Function

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

File

drv_ak4953.h

C

```
void DRV_AK4953_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_AK4953_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_AK4953_BufferAddRead function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4953_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

// Client registers an event handler with driver

DRV_AK4953_BufferEventHandlerSet(myAK4953Handle,
```

```
                APP_AK4953BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4953_BufferAddRead(myAK4953handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4953_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4953BufferEventHandler(DRV_AK4953_BUFFER_EVENT event,
        DRV_AK4953_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4953_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4953_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4953_BufferEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4953_BUFFER_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

DRV_AK4953_SamplingRateSet Function

This function sets the sampling rate of the media stream.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

Returns

None.

Description

This function sets the media sampling rate for the client handle.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

DRV_AK4953_SamplingRateSet(myAK4953Handle, 48000);  //Sets 48000 media sampling rate
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4953_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

DRV_AK4953_SetAudioCommunicationMode Function

This function provides a run time audio format configuration

File

drv_ak4953.h

C

```
void DRV_AK4953_SetAudioCommunicationMode(DRV_HANDLE handle, const DATA_LENGTH dl, const SAMPLE_LENGTH sl);
```

Returns

None

Description

This function sets up audio mode in I2S protocol

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
dl	Data length for I2S audio interface
sl	Left/Right Sample Length for I2S audio interface

Function

void DRV_AK4953_SetAudioCommunicationMode

(

DRV_HANDLE handle,

const DATA_LENGTH dl,

const SAMPLE_LENGTH sl
)

b) Status Functions

DRV_AK4953_SamplingRateGet Function

This function gets the sampling rate set on the DAC AK4953.

Implementation: Dynamic

File

drv_ak4953.h

C

```
uint32_t DRV_AK4953_SamplingRateGet(DRV_HANDLE handle);
```

Returns

None.

Description

This function gets the sampling rate set on the DAC AK4953.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
uint32_t baudRate;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

baudRate = DRV_AK4953_SamplingRateGet(myAK4953Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_AK4953_SamplingRateGet(DRV_HANDLE handle)

DRV_AK4953_Status Function

Gets the current status of the AK4953 driver module.

Implementation: Dynamic

File

drv_ak4953.h

C

```
SYS_STATUS DRV_AK4953_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This routine provides the current status of the AK4953 driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_AK4953_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4953_Initialize
SYS_STATUS          AK4953Status;

AK4953Status = DRV_AK4953_Status(object);
if (SYS_STATUS_READY == AK4953Status)
{
    // This means the driver can be opened using the
    // DRV_AK4953_Open() function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4953_Initialize routine

Function

SYS_STATUS DRV_AK4953_Status(SYS_MODULE_OBJ object)

DRV_AK4953_VersionGet Function

This function returns the version of AK4953 driver.

Implementation: Dynamic

File

drv_ak4953.h

C

```
uint32_t DRV_AK4953_VersionGet();
```

Returns

returns the version of AK4953 driver.

Description

The version number returned from the DRV_AK4953_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

Remarks

None.

Preconditions

None.

Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

Example 2

```
    uint32_t AK4953version;
    AK4953version = DRV_AK4953_VersionGet();
```

Function

uint32_t DRV_AK4953_VersionGet(void)

DRV_AK4953_VersionStrGet Function

This function returns the version of AK4953 driver in string format.

Implementation: Dynamic

File

drv_ak4953.h

C

```
int8_t* DRV_AK4953_VersionStrGet();
```

Returns

returns a string containing the version of AK4953 driver.

Description

The DRV_AK4953_VersionStrGet function returns a string in the format: ".[.][]" Where: is the AK4953 driver's version number. is the AK4953 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces.

Remarks

None.

Preconditions

None.

Example 1

"0.03a" "1.00"

Example 2

```
int8_t *AK4953string;
AK4953string = DRV_AK4953_VersionStrGet();
```

Function

int8_t* DRV_AK4953_VersionStrGet(void)

DRV_AK4953_VolumeGet Function

This function gets the volume for AK4953 CODEC.

Implementation: Dynamic

File

drv_ak4953.h

C

```
uint8_t DRV_AK4953_VolumeGet(DRV_HANDLE handle, DRV_AK4953_CHANNEL chan);
```

Returns

None.

Description

This functions gets the current volume programmed to the CODEC AK4953.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

  volume = DRV_AK4953_VolumeGet(myAK4953Handle,DRV_AK4953_CHANNEL_LEFT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to be set

Function

uint8_t DRV_AK4953_VolumeGet(DRV_HANDLE handle, DRV_AK4953_CHANNEL chan)

c) Other Functions

DRV_AK4953_BufferAddWrite Function

Schedule a non-blocking driver write operation.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_BufferAddWrite(const DRV_HANDLE handle, DRV_AK4953_BUFFER_HANDLE * bufferHandle, void *
buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4953_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4953_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4953_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4953_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4953 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4953 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 device instance and the DRV_AK4953_Status must have returned SYS_STATUS_READY.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_AK4953_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4953_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

// Client registers an event handler with driver

DRV_AK4953_BufferEventHandlerSet(myAK4953Handle,
                APP_AK4953BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4953_BufferAddWrite(myAK4953handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4953_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4953BufferEventHandler(DRV_AK4953_BUFFER_EVENT event,
        DRV_AK4953_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4953_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4953_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the AK4953 instance as return by the DRV_AK4953_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK4953_BufferAddWrite
(
const DRV_HANDLE handle,
DRV_AK4953_BUFFER_HANDLE *bufferHandle,
void *buffer, size_t size
)

DRV_AK4953_BufferAddWriteRead Function

Schedule a non-blocking driver write-read operation.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_BufferAddWriteRead(const DRV_HANDLE handle, DRV_AK4953_BUFFER_HANDLE * bufferHandle, void *
transmitBuffer, void * receiveBuffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4953_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4953_BUFFER_EVENT_COMPLETE:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4953_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4953_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4953 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4953 driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every AK4953 write. The transmit and receive size must be same.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 device instance and the DRV_AK4953_Status must have returned SYS_STATUS_READY.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_AK4953_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_AK4953_BUFFER_HANDLE bufferHandle;

// myak4953Handle is the handle returned
// by the DRV_AK4953_Open function.

// Client registers an event handler with driver

DRV_AK4953_BufferEventHandlerSet(myak4953Handle,
                APP_AK4953BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4953_BufferAddWriteRead(myak4953handle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_AK4953_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4953BufferEventHandler(DRV_AK4953_BUFFER_EVENT event,
        DRV_AK4953_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
```

```
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4953_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4953_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the AK4953 instance as returned by the DRV_AK4953_Open function
bufferHandle	Pointer to an argument that will contain the return buffer handle
transmitBuffer	The buffer where the transmit data will be stored
receiveBuffer	The buffer where the received data will be stored
size	Buffer size in bytes

Function

void DRV_AK4953_BufferAddWriteRead
(
const DRV_HANDLE handle,
DRV_AK4953_BUFFER_HANDLE *bufferHandle,
void *transmitBuffer,
void *receiveBuffer,
size_t size
)

DRV_AK4953_MuteOff Function

This function disables AK4953 output for soft mute.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_MuteOff(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables AK4953 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

    DRV_AK4953_MuteOff(myAK4953Handle); //AK4953 output soft mute disabled
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4953_MuteOff(DRV_HANDLE handle)

DRV_AK4953_MuteOn Function

This function allows AK4953 output for soft mute on.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_MuteOn(DRV_HANDLE handle);
```

Returns

None.

Description

This function Enables AK4953 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

DRV_AK4953_MuteOn(myAK4953Handle);  //AK4953 output soft muted
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4953_MuteOn(DRV_HANDLE handle);

DRV_AK4953_VolumeSet Function

This function sets the volume for AK4953 CODEC.

Implementation: Dynamic

File

drv_ak4953.h

C

```
void DRV_AK4953_VolumeSet(DRV_HANDLE handle, DRV_AK4953_CHANNEL channel, uint8_t volume);
```

Returns

None.

Description

This functions sets the volume value from 0-255. The codec has DAC value to volume range mapping as :- 00 H : +12dB FF H : -115dB In order to make the volume value to dB mapping monotonically increasing from 00 to FF, re-mapping is introduced which reverses the volume value to dB mapping as well as normalizes the volume range to a more audible dB range. The current driver implementation assumes that all dB values under -60 dB are inaudible to the human ear. Re-Mapped values 00 H : -60 dB FF H : +12 dB

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4953Handle is the handle returned
// by the DRV_AK4953_Open function.

    DRV_AK4953_VolumeSet(myAK4953Handle, DRV_AK4953_CHANNEL_LEFT, 120);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to be set
volume	volume value specified in the range 0-255 (0x00 to 0xFF)

Function

void DRV_AK4953_VolumeSet(DRV_HANDLE handle, DRV_AK4953_CHANNEL channel, uint8_t volume);

DRV_AK4953_BufferAddRead Function

Schedule a non-blocking driver read operation.

File

drv_ak4953.h

C

```
void DRV_AK4953_BufferAddRead(const DRV_HANDLE handle, DRV_AK4953_BUFFER_HANDLE * bufferHandle, void *
buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4953_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4953_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4953_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4953_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4953 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4953 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 device instance and the DRV_AK4953_Status must have returned SYS_STATUS_READY.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_AK4953_Open call.

Parameters

Parameters	Description
handle	Handle of the AK4953 instance as return by the DRV_AK4953_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

```
void DRV_AK4953_BufferAddRead
(
const       DRV_HANDLE handle,
      DRV_AK4953_BUFFER_HANDLE *bufferHandle,
void *buffer, size_t size
)
```

DRV_AK4953_IntExtMicSet Function

This function sets up the codec for the X32 DB internal or the external microphone use.

File

drv_ak4953.h

C

```
void DRV_AK4953_IntExtMicSet(DRV_HANDLE handle, DRV_AK4953_INT_EXT_MIC micInput);
```

Returns

None

Description

This function sets up the codec for the internal or the external microphone use.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
micInput	Internal vs External mic input

Function

void DRV_AK4953_IntExtMicSet

DRV_AK4953_MonoStereoMicSet Function

This function sets up the codec for the Mono or Stereo microphone mode.

File

drv_ak4953.h

C

```
void DRV_AK4953_MonoStereoMicSet(DRV_HANDLE handle, DRV_AK4953_MONO_STEREO_MIC mono_stereo_mic);
```

Returns

None

Description

This function sets up the codec for the Mono or Stereo microphone mode.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.
DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4953_MonoStereoMicSet(DRV_HANDLE handle);

DRV_AK4953_MicSet Function

This function sets up the codec for the internal or the AK4953 Mic1 or Mic2 input.

File

drv_ak4953.h

C

```
void DRV_AK4953_MicSet(DRV_HANDLE handle, DRV_AK4953_MIC micInput);
```

Returns

None

Description

This function sets up the codec.

Remarks

None.

Preconditions

The DRV_AK4953_Initialize routine must have been called for the specified AK4953 driver instance.

DRV_AK4953_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
micInput	Internal vs External mic input

Function

void DRV_AK4953_IntMic12Set

d) Data Types and Constants

DRV_AK4953_AUDIO_DATA_FORMAT Enumeration

Identifies the Serial Audio data interface format.

File

drv_ak4953.h

C

```
typedef enum {
  DRV_AK4953_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_24BIT_LSB_SDTI = 0,
  DRV_AK4953_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_16BIT_LSB_SDTI,
  DRV_AK4953_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_24BIT_MSB_SDTI,
  DRV_AK4953_AUDIO_DATA_FORMAT_I2S
} DRV_AK4953_AUDIO_DATA_FORMAT;
```

Description

AK4953 Audio data format

This enumeration identifies Serial Audio data interface format.

DRV_AK4953_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_ak4953.h

C

```
typedef enum {
  DRV_AK4953_BUFFER_EVENT_COMPLETE,
  DRV_AK4953_BUFFER_EVENT_ERROR,
  DRV_AK4953_BUFFER_EVENT_ABORT
} DRV_AK4953_BUFFER_EVENT;
```

Members

Members	Description
DRV_AK4953_BUFFER_EVENT_COMPLETE	Data was transferred successfully.
DRV_AK4953_BUFFER_EVENT_ERROR	Error while processing the request
DRV_AK4953_BUFFER_EVENT_ABORT	Data transfer aborted (Applicable in DMA mode)

Description

AK4953 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_AK4953_BufferAddWrite() function.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_AK4953_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_AK4953_BUFFER_EVENT_HANDLER Type

Pointer to a AK4953 Driver Buffer Event handler function

File

drv_ak4953.h

C

```
typedef void (* DRV_AK4953_BUFFER_EVENT_HANDLER)(DRV_AK4953_BUFFER_EVENT event, DRV_AK4953_BUFFER_HANDLE
bufferHandle, uintptr_t contextHandle);
```

Returns

None.

Description

AK4953 Driver Buffer Event Handler Function

This data type defines the required function signature for the AK4953 driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_AK4953_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_AK4953_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_AK4953_BufferProcessedSizeGet() function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4953_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver (I2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_AK4953_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void APP_MyBufferEventHandler( DRV_AK4953_BUFFER_EVENT event,
                               DRV_AK4953_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_AK4953_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_AK4953_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4953_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

File

drv_ak4953.h

C

```
typedef uintptr_t DRV_AK4953_BUFFER_HANDLE;
```

Description

AK4953 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_AK4953_BufferAddWrite() function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

DRV_AK4953_COMMAND_EVENT_HANDLER Type

Pointer to a AK4953 Driver Command Event Handler Function

File

drv_ak4953.h

C

```
typedef void (* DRV_AK4953_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

Returns

None.

Description

AK4953 Driver Command Event Handler Function

This data type defines the required function signature for the AK4953 driver command event handling callback function.

A command is a control instruction to the AK4953 CODEC. Example Mute ON/OFF, Zero Detect Enable/Disable etc.

A client must register a pointer to a command event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4953_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_AK4953CommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4953_DIGITAL_BLOCK_CONTROL Enumeration

Identifies Bass-Boost Control function

File

drv_ak4953.h

C

```
typedef enum {
  DRV_AK4953_RECORDING_MODE,
  DRV_AK4953_PLAYBACK_MODE,
  DRV_AK4953_RECORDING_PLAYBACK_2_MODE,
  DRV_AK4953_LOOPBACK_MODE
} DRV_AK4953_DIGITAL_BLOCK_CONTROL;
```

Members

Members	Description
DRV_AK4953_RECORDING_MODE	This is the default setting
DRV_AK4953_PLAYBACK_MODE	Min control
DRV_AK4953_RECORDING_PLAYBACK_2_MODE	Medium control
DRV_AK4953_LOOPBACK_MODE	Maximum control

Description

AK4953 Bass-Boost Control

This enumeration identifies the settings for Bass-Boost Control function.

Remarks

None.

DRV_AK4953_INIT Structure

Defines the data required to initialize or reinitialize the AK4953 driver

File

drv_ak4953.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX i2sDriverModuleIndex;
  SYS_MODULE_INDEX i2cDriverModuleIndex;
  uint32_t samplingRate;
  uint8_t volume;
  DRV_AK4953_AUDIO_DATA_FORMAT audioDataFormat;
} DRV_AK4953_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX i2sDriverModuleIndex;	Identifies data module(I2S) driver ID for data interface of CODEC
SYS_MODULE_INDEX i2cDriverModuleIndex;	Identifies data module(I2C) driver ID for control interface of CODEC
uint32_t samplingRate;	Sampling rate
uint8_t volume;	Volume
DRV_AK4953_AUDIO_DATA_FORMAT audioDataFormat;	Identifies the Audio data format

Description

AK4953 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the AK4953 CODEC driver.

Remarks

None.

_DRV_AK4953_H Macro

File

drv_ak4953.h

C

```
#define _DRV_AK4953_H
```

Description

Include files.

DRV_AK4953_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_ak4953.h

C

```
#define DRV_AK4953_BUFFER_HANDLE_INVALID ((DRV_AK4953_BUFFER_HANDLE)(-1))
```

Description

AK4953 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_AK4953_BufferAddWrite() function if the buffer add request was not successful.

Remarks

None

DRV_AK4953_COUNT Macro

Number of valid AK4953 driver indices

File

drv_ak4953.h

C

```
#define DRV_AK4953_COUNT
```

Description

AK4953 Driver Module Count

This constant identifies the maximum number of AK4953 Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of AK4953 instances on this microcontroller.

Remarks

This value is part-specific.

DRV_AK4953_INDEX_0 Macro

AK4953 driver index definitions

File

drv_ak4953.h

C

```
#define DRV_AK4953_INDEX_0 0
```

Description

Driver AK4953 Module Index

These constants provide AK4953 driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_AK4953_Initialize and DRV_AK4953_Open routines to identify the driver instance in use.

DRV_AK4953_INDEX_1 Macro

File

drv_ak4953.h

C

```
#define DRV_AK4953_INDEX_1 1
```

Description

This is macro DRV_AK4953_INDEX_1.

DRV_AK4953_INDEX_2 Macro

File

drv_ak4953.h

C

```
#define DRV_AK4953_INDEX_2 2
```

Description

This is macro DRV_AK4953_INDEX_2.

DRV_AK4953_INDEX_3 Macro

File

drv_ak4953.h

C

```
#define DRV_AK4953_INDEX_3 3
```

Description

This is macro DRV_AK4953_INDEX_3.

DRV_AK4953_INDEX_4 Macro

File

drv_ak4953.h

C

```
#define DRV_AK4953_INDEX_4 4
```

Description

This is macro DRV_AK4953_INDEX_4.

DRV_AK4953_INDEX_5 Macro

File

drv_ak4953.h

C

```
#define DRV_AK4953_INDEX_5 5
```

Description

This is macro DRV_AK4953_INDEX_5.

DRV_AK4953_CHANNEL Enumeration

Identifies Left/Right Audio channel

File

drv_ak4953.h

C

```
typedef enum {
  DRV_AK4953_CHANNEL_LEFT,
  DRV_AK4953_CHANNEL_RIGHT,
  DRV_AK4953_CHANNEL_LEFT_RIGHT,
  DRV_AK4953_NUMBER_OF_CHANNELS
} DRV_AK4953_CHANNEL;
```

Description

AK4953 Audio Channel

This enumeration identifies Left/Right Audio channel

Remarks

None.

DRV_AK4953_INT_EXT_MIC Enumeration

Identifies the Mic input source.

File

drv_ak4953.h

C

```
typedef enum {
  INT_MIC,
  EXT_MIC
} DRV_AK4953_INT_EXT_MIC;
```

Description

AK4953 Mic Internal / External Input

This enumeration identifies the Mic input source.

DRV_AK4953_MONO_STEREO_MIC Enumeration

Identifies the Mic input as Mono / Stereo.

File

drv_ak4953.h

C

```
typedef enum {
  ALL_ZEROS,
  MONO_RIGHT_CHANNEL,
  MONO_LEFT_CHANNEL,
```

```
  STEREO
} DRV_AK4953_MONO_STEREO_MIC;
```

Description

AK4953 Mic Mono / Stereo Input

This enumeration identifies the Mic input as Mono / Stereo.

DRV_AK4953_MIC Enumeration

File

drv_ak4953.h

C

```
typedef enum {
  MIC1 = 0,
  MIC2,
  MIC3,
  DRV_AK4953_NUMBER_OF_MIC
} DRV_AK4953_MIC;
```

Members

Members	Description
MIC1 = 0	INT_MIC
MIC2	EXT_MIC
MIC3	LINE-IN

Description

This is type DRV_AK4953_MIC.

Files

Files

Name	Description
drv_ak4953.h	AK4953 CODEC Driver Interface header file
drv_ak4953_config_template.h	AK4953 Codec Driver Configuration Template.

Description

This section lists the source and header files used by the AK4953Codec Driver Library.

drv_ak4953.h

AK4953 CODEC Driver Interface header file

Enumerations

	Name	Description
	DRV_AK4953_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4953_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4953_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4953_DIGITAL_BLOCK_CONTROL	Identifies Bass-Boost Control function
	DRV_AK4953_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4953_MIC	This is type DRV_AK4953_MIC.
	DRV_AK4953_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.

Functions

	Name	Description
	DRV_AK4953_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4953_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic

	DRV_AK4953_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_AK4953_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_AK4953_Close	Closes an opened-instance of the AK4953 driver. **Implementation:** Dynamic
	DRV_AK4953_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished. **Implementation:** Dynamic
	DRV_AK4953_Deinitialize	Deinitializes the specified instance of the AK4953 driver module. **Implementation:** Dynamic
	DRV_AK4953_Initialize	Initializes hardware and data for the instance of the AK4953 DAC module. **Implementation:** Dynamic
	DRV_AK4953_IntExtMicSet	This function sets up the codec for the X32 DB internal or the external microphone use.
	DRV_AK4953_MicSet	This function sets up the codec for the internal or the AK4953 Mic1 or Mic2 input.
	DRV_AK4953_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK4953_MuteOff	This function disables AK4953 output for soft mute. **Implementation:** Dynamic
	DRV_AK4953_MuteOn	This function allows AK4953 output for soft mute on. **Implementation:** Dynamic
	DRV_AK4953_Open	Opens the specified AK4953 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_AK4953_SamplingRateGet	This function gets the sampling rate set on the DAC AK4953. **Implementation:** Dynamic
	DRV_AK4953_SamplingRateSet	This function sets the sampling rate of the media stream. **Implementation:** Dynamic
	DRV_AK4953_SetAudioCommunicationMode	This function provides a run time audio format configuration
	DRV_AK4953_Status	Gets the current status of the AK4953 driver module. **Implementation:** Dynamic
	DRV_AK4953_Tasks	Maintains the driver's control and data interface state machine. **Implementation:** Dynamic
	DRV_AK4953_VersionGet	This function returns the version of AK4953 driver. **Implementation:** Dynamic
	DRV_AK4953_VersionStrGet	This function returns the version of AK4953 driver in string format. **Implementation:** Dynamic
	DRV_AK4953_VolumeGet	This function gets the volume for AK4953 CODEC. **Implementation:** Dynamic
	DRV_AK4953_VolumeSet	This function sets the volume for AK4953 CODEC. **Implementation:** Dynamic

Macros

	Name	Description
	_DRV_AK4953_H	Include files.
	DRV_AK4953_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4953_COUNT	Number of valid AK4953 driver indices
	DRV_AK4953_INDEX_0	AK4953 driver index definitions
	DRV_AK4953_INDEX_1	This is macro DRV_AK4953_INDEX_1.
	DRV_AK4953_INDEX_2	This is macro DRV_AK4953_INDEX_2.
	DRV_AK4953_INDEX_3	This is macro DRV_AK4953_INDEX_3.
	DRV_AK4953_INDEX_4	This is macro DRV_AK4953_INDEX_4.
	DRV_AK4953_INDEX_5	This is macro DRV_AK4953_INDEX_5.

Structures

	Name	Description
	DRV_AK4953_INIT	Defines the data required to initialize or reinitialize the AK4953 driver

Types

	Name	Description
	DRV_AK4953_BUFFER_EVENT_HANDLER	Pointer to a AK4953 Driver Buffer Event handler function
	DRV_AK4953_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4953_COMMAND_EVENT_HANDLER	Pointer to a AK4953 Driver Command Event Handler Function

Description

AK4953 CODEC Driver Interface

The AK4953 CODEC device driver interface provides a simple interface to manage the AK4953 106dB 192kHz 24-Bit DAC that can be interfaced Microchip Microcontroller. This file provides the interface definition for the AK4953 CODEC device driver.

File Name

drv_AK4953.h

Company

Microchip Technology Inc.

drv_ak4953_config_template.h

AK4953 Codec Driver Configuration Template.

Macros

	Name	Description
	DRV_AK4953_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4953_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4953_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4953_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4953_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4953_MCLK_SOURCE	Indicate the input clock frequency to generate the MCLK to codec.
	DRV_AK4953_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.

Description

AK4953 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_ak4953_config_template.h

Company

Microchip Technology Inc.

AK4954 Codec Driver Library

This topic describes the AK4954 Codec Driver Library.

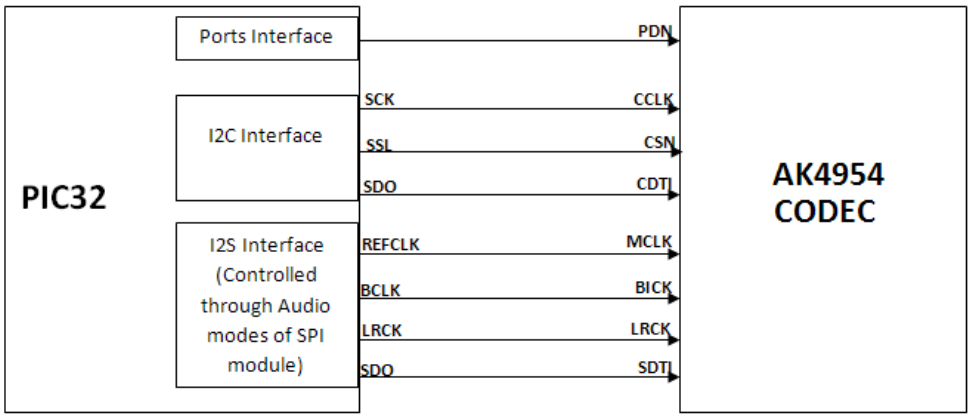
Introduction

This library provides an interface to manage the AK4954 Codec that is serially interfaced to a Microchip microcontroller for providing Audio Solutions.

Description

The AK4954 module is 16/24-bit Audio Codec from Asahi Kasei Microdevices Corporation. The AK4954 can be interfaced to Microchip microcontrollers through I2C and I2S serial interfaces. The I2C interface is used for control command transfer. The I2S interface is used for Audio data output.

A typical interface of AK4954 to a Microchip PIC32 device is provided in the following diagram:



Features

The AK4954 Codec supports the following features:

- Audio Interface Format: MSB first
- ADC: 24-bit MSB justified, 16/24-bit I2S
- DAC: 24-bit MSB justified, 1-6bit LSB justified, 24-bit LSB justified, 16/24-bit I2S
- Sampling Frequency Range: 8 kHz to 192 kHz
- Digital Volume Control: +12dB ~ .115dB, 0.5dB Step
- SoftMute: On and Off
- Master Clock Frequencies: 32 fs/64 fs/128 fs/256 fs

Using the Library

This topic describes the basic architecture of the AK4954 Codec Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_ak4954.h`

The interface to the AK4954 Codec Driver library is defined in the `drv_ak4954.h` header file. Any C language source (`.c`) file that uses the AK4954 Codec Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

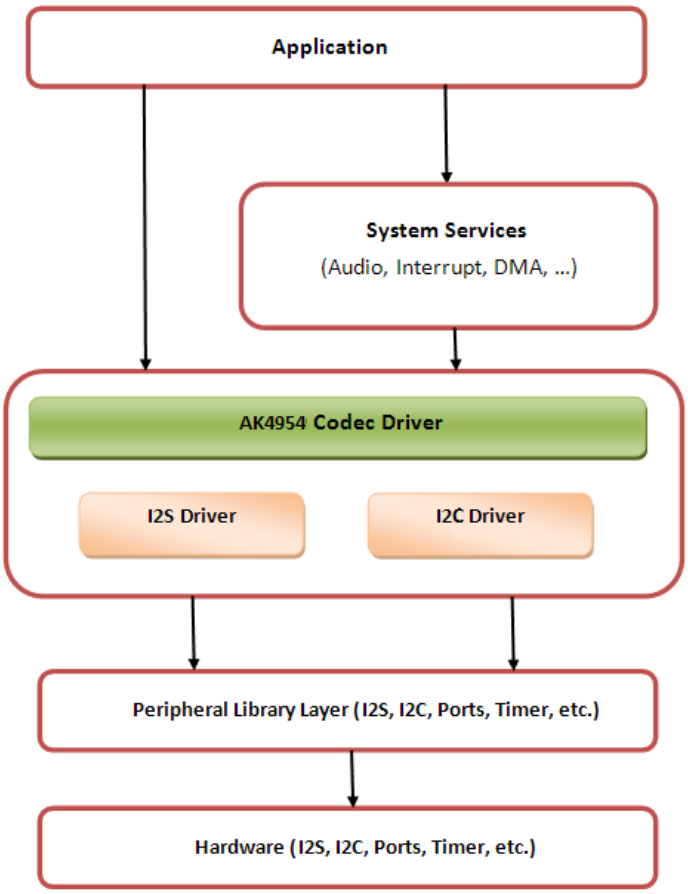
Abstraction Model

This library provides a low-level abstraction of the AK4954 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The abstraction model shown in the following diagram depicts how the AK4954 Codec Driver is positioned in the MPLAB Harmony framework. The AK4954 Codec Driver uses the SPI and I2S drivers for control and audio data transfers to the AK4954 module.

AK4954 Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The AK4954 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced AK4954 DAC module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the AK4954 Codec Driver Library.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Status Functions	Provides status functions.
Other Functions	Provides driver specific miscellaneous functions such as sampling rate setting, control command functions, etc.
Data Types and Constants	These data types and constants are required while interacting and setting up the AK4954 Codec Driver Library.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

System Access

This topic describes system initialization, implementations, and includes a system access code example.

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the AK4954 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_AK4954_INIT or by using Initialization Overrides) that are supported by the specific AK4954 device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver.
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver.
- Sampling rate
- Audio data format. The audio data format should match with the audio data format settings done in I2S driver initialization
- Power down pin port initialization
- Queue size for the audio data transmit buffer

The DRV_AK4954_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ AK4954_Deinitialize, DRV_ AK4954_Status and DRV_I2S_Tasks.

Implementations

The AK4954 Codec Driver can has the following implementation:

Description	MPLAB Harmony Components
Dedicated hardware for control (I2C) and data (I2S) interface.	Standard MPLAB Harmony drivers for I2C and I2S interfaces.

Example:

```
DRV_AK4954_INIT drvak4954Codec0InitData =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .i2sDriverModuleIndex = DRV_AK4954_I2S_DRIVER_MODULE_INDEX_IDX0,
    .i2cDriverModuleIndex = DRV_AK4954_I2C_DRIVER_MODULE_INDEX_IDX0,
    .volume = DRV_AK4954_VOLUME,
    .queueSizeTransmit = DRV_AK4954_TRANSMIT_QUEUE_SIZE,
};

// Initialize the I2C driver
   DRV_I2C0_Initialize();

// Initialize the I2S driver. The I2S module index should be same as the one used in initializing
// the I2S driver.
   sysObj.drvI2S0 = DRV_I2S_Initialize(DRV_I2S_INDEX_0, (SYS_MODULE_INIT *)&drvI2S0InitData);

// Initialize the Codec driver
   sysObj.drvak4954Codec0 = DRV_AK4954_Initialize(DRV_AK4954_INDEX_0, (SYS_MODULE_INIT
*)&drvak4954Codec0InitData);

if (SYS_MODULE_OBJ_INVALID == AK4954DevObject)
{
// Handle error
}
```

Task Routine

The DRV_AK4954_Tasks will be called from the System Task Service.

Client Access

For the application to start using an instance of the module, it must call the DRV_AK4954_Open function. The DRV_AK4954_Open provides a driver handle to the AK4954 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_AK4954_Deinitialize, the application must call the DRV_AK4954_Open function again to set up the instance of the driver.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

Client Operations

This topic provides information on client operations and includes a control command and audio buffered data operation flow diagram.

Description

Client operations provide the API interface for control command and audio data transfer to the AK4954 Codec.

The following AK4954 Codec specific control command functions are provided:

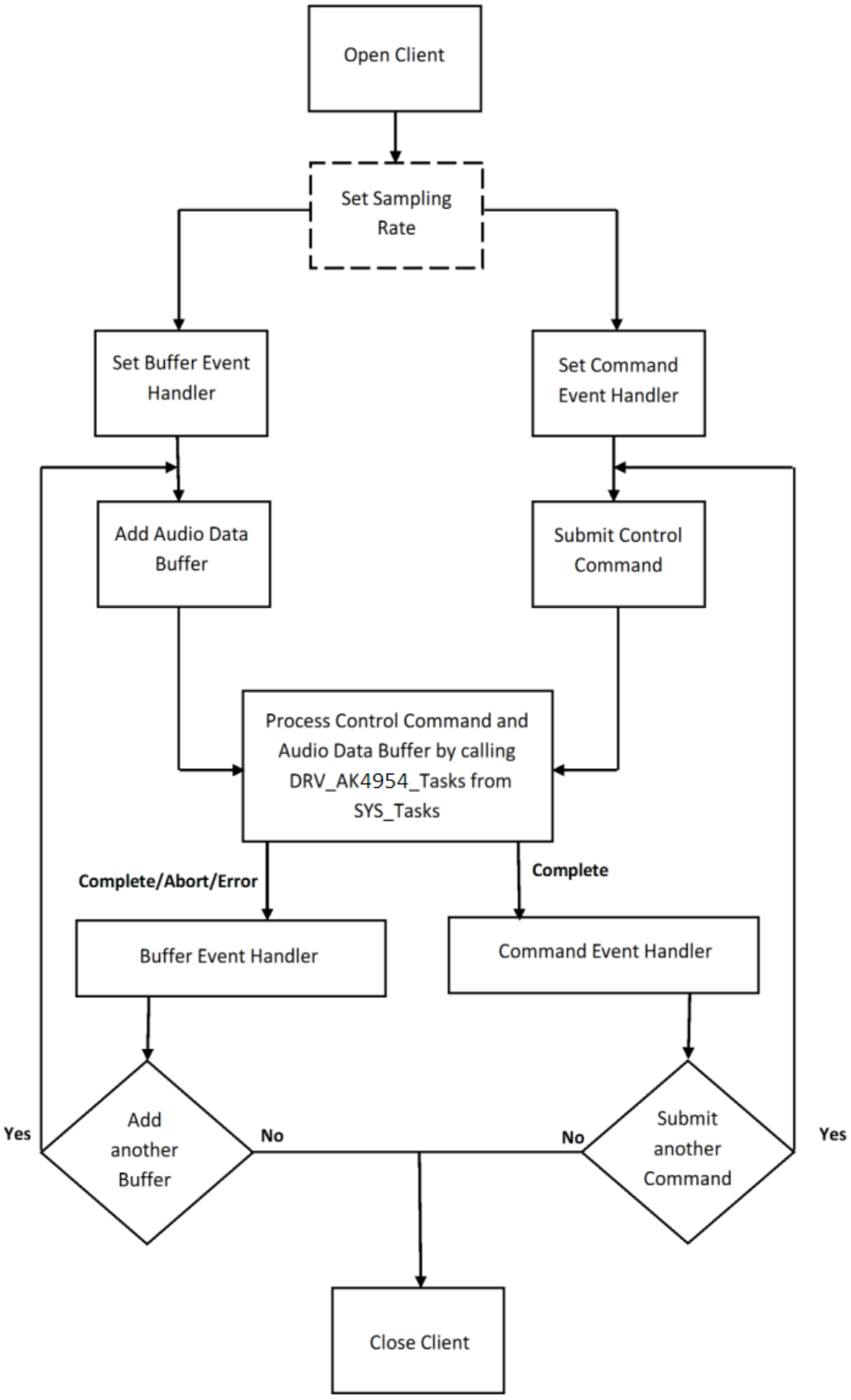
- DRV_AK4954_SamplingRateSet
- DRV_AK4954_SamplingRateGet
- DRV_AK4954_VolumeSet
- DRV_AK4954_VolumeGet
- DRV_AK4954_MuteOn
- DRV_AK4954_MuteOff
- DRV_AK4954_IntExtMicSet
- DRV_AK4954_MonoStereoMicSet

These functions schedule a non-blocking control command transfer operation. These functions submit the control command request to the AK4954 Codec. These functions submit the control command request to I2C Driver transmit queue, the request is processed immediately if it is the first request, or processed when the previous request is complete.

DRV_AK4954_BufferAddWrite, DRV_AK4954_BufferAddRead, and DRV_AK4954_BufferAddWriteRead are buffered data operation functions.

These functions schedule non-blocking audio data transfer operations. These functions add the request to I2S Driver transmit or receive buffer queue depends on the request type, and are executed immediately if it is the first buffer, or executed later when the previous buffer is complete. The driver notifies the client with DRV_AK4954_BUFFER_EVENT_COMPLETE, DRV_AK4954_BUFFER_EVENT_ERROR, or DRV_AK4954_BUFFER_EVENT_ABORT events.

The following diagram illustrates the control commands and audio buffered data operations.



Note: It is not necessary to close and reopen the client between multiple transfers.

Configuring the Library

Macros

	Name	Description
	DRV_AK4954_BCLK_BIT_CLK_DIVISOR	Indicates whether the initilization of the AK4954 codec should be delayed.
	DRV_AK4954_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4954_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4954_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4954_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4954_MCLK_SOURCE	Indicate the input clock frequency to generate the MCLK to codec.
	DRV_AK4954_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.

Description

The configuration of the AK4954 Codec Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the AK4954 Codec Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the AK4954 Codec Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_AK4954_BCLK_BIT_CLK_DIVISOR Macro

Indicates whether the initilization of the AK4954 codec should be delayed.

File

drv_ak4954_config_template.h

C

```
#define DRV_AK4954_BCLK_BIT_CLK_DIVISOR
```

Description

AK4954 Delay Initialization

If the AK4954 Codec shares its RESET pin with another peripheral, such as a Bluetooth module, then this define should be true, in order to indicate the AK4954 Codec should starts its initialization only after the other peripheral has completed theirs. It is set in the MHC menu with the checkbox: "Delay driver initialization (due to shared RESET pin)"

Remarks

This needs to be set, for example, in the case where the AK4954 and the BM64 share a common PDN (power down) or RESET pin on the PIC32 Bluetooth Audio Development Kit (BTADK).

DRV_AK4954_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_ak4954_config_template.h

C

```
#define DRV_AK4954_CLIENTS_NUMBER DRV_AK4954_INSTANCES_NUMBER
```

Description

AK4954 Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. Typically only one client could be connected to one hardware instance. This value represents the total number of clients to be supported across all hardware instances. Therefore, if there are five AK4954 hardware interfaces, this number will be 5.

Remarks

None.

DRV_AK4954_INPUT_REFCLOCK Macro

Identifies the input REFCLOCK source to generate the MCLK to codec.

File

drv_ak4954_config_template.h

C

```
#define DRV_AK4954_INPUT_REFCLOCK
```

Description

AK4954 Input reference clock

Identifies the input REFCLOCK source to generate the MCLK to codec.

Remarks

None.

DRV_AK4954_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_ak4954_config_template.h

C

```
#define DRV_AK4954_INSTANCES_NUMBER
```

Description

AK4954 driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of AK4954 CODEC modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None.

DRV_AK4954_MCLK_SAMPLE_FREQ_MULTPLIER Macro

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency

File

drv_ak4954_config_template.h

C

```
#define DRV_AK4954_MCLK_SAMPLE_FREQ_MULTPLIER
```

Description

AK4954 MCLK to LRCK Ratio to Generate Audio Stream

Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency I2S sampling frequency

Supported MCLK to Sampling frequency Ratios are as below 256fs, 384fs, 512fs, 768fs or 1152fs

Remarks

None

DRV_AK4954_MCLK_SOURCE Macro

Indicate the input clock frequency to generate the MCLK to codec.

File

drv_ak4954_config_template.h

C

```
#define DRV_AK4954_MCLK_SOURCE
```

Description

AK4954 Data Interface Master Clock Speed configuration

Indicate the input clock frequency to generate the MCLK to codec.

Remarks

None.

DRV_AK4954_QUEUE_DEPTH_COMBINED Macro

Number of entries of all queues in all instances of the driver.

File

drv_ak4954_config_template.h

C

```
#define DRV_AK4954_QUEUE_DEPTH_COMBINED
```

Description

AK4954 Driver Buffer Queue Entries

This macro defined the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for transmit operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build). The hardware instance transmit buffer queue will queue transmit buffers submitted by the DRV_AK4954_BufferAddWrite function.

A buffer queue will contains buffer queue entries, each related to a BufferAdd request. This configuration macro defines total number of buffer entries that will be available for use between all AK4954 driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking write requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its transmit buffer queue size.

As an example, consider the case of static single client driver application where full duplex non blocking operation is desired without queuing, the minimum transmit queue depth and minimum receive queue depth should be 1. Hence the total number of buffer entries should be 2.

As an example, consider the case of a dynamic driver (say two instances) where instance one will queue up to three write requests and up to two read requests, and instance two will queue up to two write requests and up to six read requests, the value of this macro should be 13 (2 + 3 + 2 + 6).

Configuring the MHC

Provides examples on how to configure the MPLAB Harmony Configurator (MHC) for a specific driver.

Description

The following three figures show examples of MHC configurations for the AK4954 Codec Driver, I2S Driver, and the I2C Driver.

Figure 1: AK4954 Codec Driver MHC Configuration

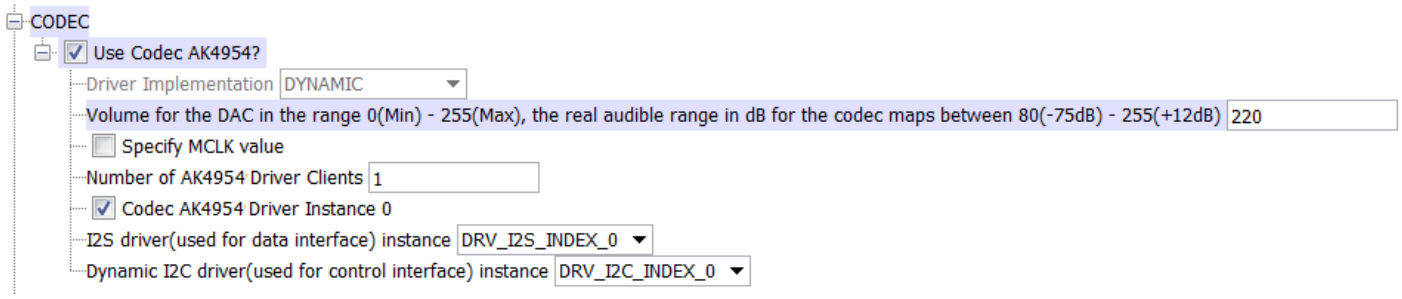


Figure 2: I2S Driver MHC Configuration

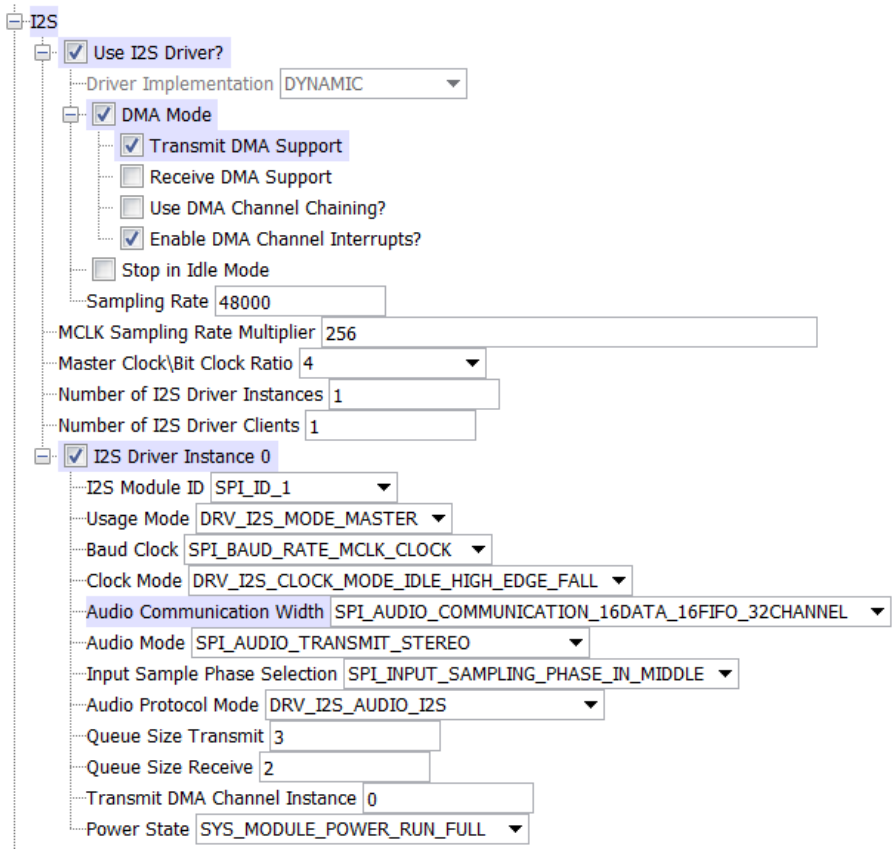
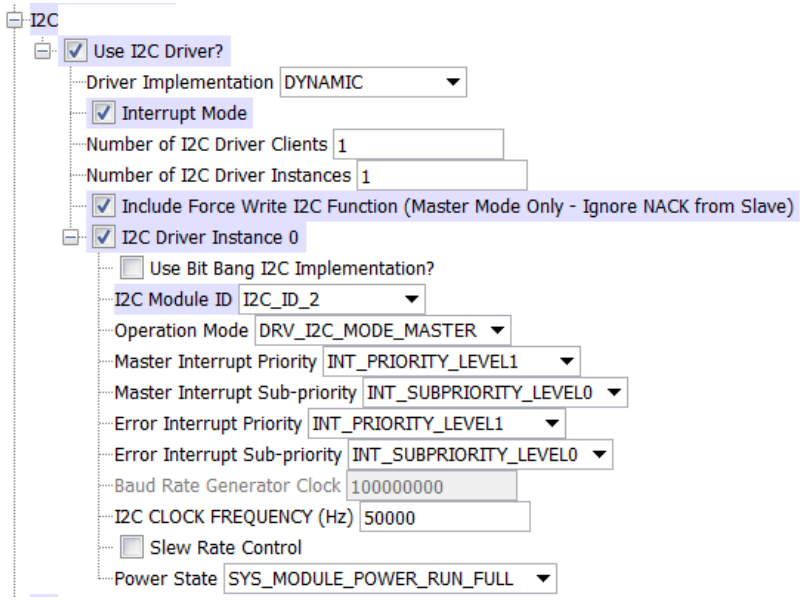


Figure 3: I2C Driver MHC Configuration



Migrating the AK4954 Driver From Earlier Versions of Microchip Harmony

Prior to version 1.08 of MPLAB Harmony, the AK4954 Codec Driver Library used the static I2C driver implementation. Beginning with v1.08 of MPLAB Harmony, applications must use the Dynamic Driver implementation with the MHC configured as shown in Figure 3. In addition, PIC32MZ configurations require the "Include Force Write I2C Function (Master Mode Only - Ignore NACK from Slave)" option to be selected.

Building the Library

This section lists the files that are available in the AK4954 Codec Driver Library.

Description

This section list the files that are available in the `/src` folder of the AK4954 Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/codec/ak4954`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_ak4954.h	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_ak4954.c	This file contains implementation of the AK4954 Codec Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The AK4954 Codec Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_AK4954_Initialize	Initializes hardware and data for the instance of the AK4954 DAC module. **Implementation:** Dynamic
	DRV_AK4954_Deinitialize	Deinitializes the specified instance of the AK4954 driver module. **Implementation:** Dynamic
	DRV_AK4954_Open	Opens the specified AK4954 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_AK4954_Close	Closes an opened-instance of the AK4954 driver. **Implementation:** Dynamic
	DRV_AK4954_Tasks	Maintains the driver's control and data interface state machine. **Implementation:** Dynamic
	DRV_AK4954_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished. **Implementation:** Dynamic
	DRV_AK4954_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_AK4954_SamplingRateSet	This function sets the sampling rate of the media stream. **Implementation:** Dynamic
	DRV_AK4954_SetAudioCommunicationMode	This function provides a run time audio format configuration

b) Status Functions

	Name	Description
	DRV_AK4954_SamplingRateGet	This function gets the sampling rate set on the DAC AK4954. **Implementation:** Dynamic
	DRV_AK4954_Status	Gets the current status of the AK4954 driver module. **Implementation:** Dynamic
	DRV_AK4954_VersionGet	This function returns the version of AK4954 driver. **Implementation:** Dynamic
	DRV_AK4954_VersionStrGet	This function returns the version of AK4954 driver in string format. **Implementation:** Dynamic
	DRV_AK4954_VolumeGet	This function gets the volume for AK4954 CODEC. **Implementation:** Dynamic

c) Other Functions

	Name	Description
	DRV_AK4954_VolumeSet	This function sets the volume for AK4954 CODEC. **Implementation:** Dynamic
	DRV_AK4954_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4954_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_AK4954_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_AK4954_IntExtMicSet	This function sets up the codec for the X32 DB internal or the external microphone use.
	DRV_AK4954_MicSet	This function sets up the codec for the internal or the AK4954 Mic1 or Mic2 input.
	DRV_AK4954_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK4954_MuteOff	This function disables AK4954 output for soft mute. **Implementation:** Dynamic
	DRV_AK4954_MuteOn	This function allows AK4954 output for soft mute on. **Implementation:** Dynamic

d) Data Types and Constants

	Name	Description
	DRV_AK4954_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4954_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4954_BUFFER_EVENT_HANDLER	Pointer to a AK4954 Driver Buffer Event handler function
	DRV_AK4954_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4954_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4954_COMMAND_EVENT_HANDLER	Pointer to a AK4954 Driver Command Event Handler Function
	DRV_AK4954_DIGITAL_BLOCK_CONTROL	Identifies Bass-Boost Control function
	DRV_AK4954_INIT	Defines the data required to initialize or reinitialize the AK4954 driver
	DRV_AK4954_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4954_MIC	This is type DRV_AK4954_MIC.
	DRV_AK4954_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.
	DRV_AK4954_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4954_COUNT	Number of valid AK4954 driver indices
	DRV_AK4954_INDEX_0	AK4954 driver index definitions
	DRV_AK4954_INDEX_1	This is macro DRV_AK4954_INDEX_1.
	DRV_AK4954_INDEX_2	This is macro DRV_AK4954_INDEX_2.
	DRV_AK4954_INDEX_3	This is macro DRV_AK4954_INDEX_3.
	DRV_AK4954_INDEX_4	This is macro DRV_AK4954_INDEX_4.
	DRV_AK4954_INDEX_5	This is macro DRV_AK4954_INDEX_5.

Description

This section describes the API functions of the AK4954 Codec Driver library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_AK4954_Initialize Function

Initializes hardware and data for the instance of the AK4954 DAC module.

Implementation: Dynamic

File

drv_ak4954.h

C

```
SYS_MODULE_OBJ DRV_AK4954_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the AK4954 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

Remarks

This routine must be called before any other AK4954 routine is called.

This routine should only be called once during system initialization unless DRV_AK4954_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this CODEC driver. Also DRV_I2C_Initialize must be called before calling this function to initialize the control interface of this CODEC driver.

Example

```
DRV_AK4954_INIT                 init;
SYS_MODULE_OBJ                  objectHandle;

init->inUse                             = true;
init->status                            = SYS_STATUS_BUSY;
init->numClients                        = 0;
init->i2sDriverModuleIndex              = ak4954Init->i2sDriverModuleIndex;
init->i2cDriverModuleIndex              = ak4954Init->i2cDriverModuleIndex;
init->samplingRate                      = DRV_AK4954_AUDIO_SAMPLING_RATE;
init->audioDataFormat                   = DRV_AK4954_AUDIO_DATA_FORMAT_MACRO;
for(index=0; index < DRV_AK4954_NUMBER_OF_CHANNELS; index++)
{
    init->volume[index] = ak4954Init->volume;
}
init->isInInterruptContext              = false;

init->commandCompleteCallback = (DRV_AK4954_COMMAND_EVENT_HANDLER)0;
init->commandContextData = 0;

init->mclk_multiplier = DRV_AK4954_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_AK4954_Initialize(DRV_AK4954_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized

init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

SYS_MODULE_OBJ DRV_AK4954_Initialize
(
const SYS_MODULE_INDEX drvIndex,
const SYS_MODULE_INIT *const init
);

DRV_AK4954_Deinitialize Function

Deinitializes the specified instance of the AK4954 driver module.

Implementation: Dynamic

File

drv_ak4954.h

C

```
void DRV_AK4954_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the AK4954 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_AK4954_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_AK4954_Initialize
SYS_STATUS          status;


DRV_AK4954_Deinitialize(object);

status = DRV_AK4954_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4954_Initialize routine

Function

void DRV_AK4954_Deinitialize(SYS_MODULE_OBJ object)

DRV_AK4954_Open Function

Opens the specified AK4954 driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_ak4954.h

C

```
DRV_HANDLE DRV_AK4954_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_AK4954_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

Description

This routine opens the specified AK4954 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

AK4954 can be opened with DRV_IO_INTENT_WRITE, or DRV_IO_INTENT_READ or DRV_IO_INTENT_WRITEREAD io_intent option. This decides whether the driver is used for headphone output, or microphone input or both modes simultaneously.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_AK4954_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

Function DRV_AK4954_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_AK4954_Open(DRV_AK4954_INDEX_0, DRV_IO_INTENT_WRITEREAD | DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

DRV_HANDLE DRV_AK4954_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

)

DRV_AK4954_Close Function

Closes an opened-instance of the AK4954 driver.

Implementation: Dynamic

File

drv_ak4954.h

C

```
void DRV_AK4954_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This routine closes an opened-instance of the AK4954 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_AK4954_Open before the caller may use the driver again

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_AK4954_Open

DRV_AK4954_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4954_Close(DRV_Handle handle)

DRV_AK4954_Tasks Function

Maintains the driver's control and data interface state machine.

Implementation: Dynamic

File

drv_ak4954.h

C

```
void DRV_AK4954_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks() function.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4954_Initialize

while (true)
{
    DRV_AK4954_Tasks (object);
```

```
    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK4954_Initialize)

Function

void DRV_AK4954_Tasks(SYS_MODULE_OBJ object);

DRV_AK4954_CommandEventHandlerSet Function

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

Implementation: Dynamic

File

drv_ak4954.h

C

```
void DRV_AK4954_CommandEventHandlerSet(DRV_HANDLE handle, const DRV_AK4954_COMMAND_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

When a client calls DRV_AK4954_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "AK4954 CODEC Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4954_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

// Client registers an event handler with driver

DRV_AK4954_CommandEventHandlerSet(myAK4954Handle,
                APP_AK4954CommandEventHandler, (uintptr_t)&myAppObj);

DRV_AK4954_DeEmphasisFilterSet(myAK4954Handle, DRV_AK4954_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_AK4954CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.
```

```
    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4954_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4954_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

DRV_AK4954_BufferEventHandlerSet Function

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

File

drv_ak4954.h

C

```
void DRV_AK4954_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_AK4954_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_AK4954_BufferAddRead function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4954_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

// Client registers an event handler with driver

DRV_AK4954_BufferEventHandlerSet(myAK4954Handle,
```

```
                APP_AK4954BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4954_BufferAddRead(myAK4954handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4954_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4954BufferEventHandler(DRV_AK4954_BUFFER_EVENT event,
        DRV_AK4954_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4954_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4954_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK4954_BufferEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK4954_BUFFER_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

DRV_AK4954_SamplingRateSet Function

This function sets the sampling rate of the media stream.

Implementation: Dynamic

File

drv_ak4954.h

C

```
void DRV_AK4954_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

Returns

None.

Description

This function sets the media sampling rate for the client handle.

Remarks

None.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Example

```
// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

DRV_AK4954_SamplingRateSet(myAK4954Handle, 48000);  //Sets 48000 media sampling rate
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4954_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

DRV_AK4954_SetAudioCommunicationMode Function

This function provides a run time audio format configuration

File

drv_ak4954.h

C

```
void DRV_AK4954_SetAudioCommunicationMode(DRV_HANDLE handle, const DATA_LENGTH dl, const SAMPLE_LENGTH sl);
```

Returns

None

Description

This function sets up audio mode in I2S protocol

Remarks

None.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
dl	Data length for I2S audio interface
sl	Left/Right Sample Length for I2S audio interface

Function

void DRV_AK4954_SetAudioCommunicationMode

(

DRV_HANDLE handle,

const DATA_LENGTH dl,

const SAMPLE_LENGTH sl
)

b) Status Functions

DRV_AK4954_SamplingRateGet Function

This function gets the sampling rate set on the DAC AK4954.

Implementation: Dynamic

File

drv_ak4954.h

C

```
uint32_t DRV_AK4954_SamplingRateGet(DRV_HANDLE handle);
```

Returns

None.

Description

This function gets the sampling rate set on the DAC AK4954.

Remarks

None.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Example

```
uint32_t baudRate;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

baudRate = DRV_AK4954_SamplingRateGet(myAK4954Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_AK4954_SamplingRateGet(DRV_HANDLE handle)

DRV_AK4954_Status Function

Gets the current status of the AK4954 driver module.

Implementation: Dynamic

File

drv_ak4954.h

C

```
SYS_STATUS DRV_AK4954_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This routine provides the current status of the AK4954 driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_AK4954_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK4954_Initialize
SYS_STATUS          AK4954Status;

AK4954Status = DRV_AK4954_Status(object);
if (SYS_STATUS_READY == AK4954Status)
{
    // This means the driver can be opened using the
    // DRV_AK4954_Open() function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4954_Initialize routine

Function

SYS_STATUS DRV_AK4954_Status(SYS_MODULE_OBJ object)

DRV_AK4954_VersionGet Function

This function returns the version of AK4954 driver.

Implementation: Dynamic

File

drv_ak4954.h

C

```
uint32_t DRV_AK4954_VersionGet();
```

Returns

returns the version of AK4954 driver.

Description

The version number returned from the DRV_AK4954_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

Remarks

None.

Preconditions

None.

Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

Example 2

```
    uint32_t AK4954version;
    AK4954version = DRV_AK4954_VersionGet();
```

Function

uint32_t DRV_AK4954_VersionGet(void)

DRV_AK4954_VersionStrGet Function

This function returns the version of AK4954 driver in string format.

Implementation: Dynamic

File

drv_ak4954.h

C

```
int8_t* DRV_AK4954_VersionStrGet();
```

Returns

returns a string containing the version of AK4954 driver.

Description

The DRV_AK4954_VersionStrGet function returns a string in the format: ".[.][]" Where: is the AK4954 driver's version number. is the AK4954 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces.

Remarks

None.

Preconditions

None.

Example 1

"0.03a" "1.00"

Example 2

```
int8_t *AK4954string;
AK4954string = DRV_AK4954_VersionStrGet();
```

Function

int8_t* DRV_AK4954_VersionStrGet(void)

DRV_AK4954_VolumeGet Function

This function gets the volume for AK4954 CODEC.

Implementation: Dynamic

File

drv_ak4954.h

C

```
uint8_t DRV_AK4954_VolumeGet(DRV_HANDLE handle, DRV_AK4954_CHANNEL chan);
```

Returns

None.

Description

This functions gets the current volume programmed to the CODEC AK4954.

Remarks

None.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

  volume = DRV_AK4954_VolumeGet(myAK4954Handle,DRV_AK4954_CHANNEL_LEFT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to be set

Function

uint8_t DRV_AK4954_VolumeGet(DRV_HANDLE handle, DRV_AK4954_CHANNEL chan)

c) Other Functions

DRV_AK4954_VolumeSet Function

This function sets the volume for AK4954 CODEC.

Implementation: Dynamic

File

drv_ak4954.h

C

```
void DRV_AK4954_VolumeSet(DRV_HANDLE handle, DRV_AK4954_CHANNEL channel, uint8_t volume);
```

Returns

None.

Description

This functions sets the volume value from 0-255. The codec has DAC value to volume range mapping as :- 00 H : +12dB FF H : -115dB In order to make the volume value to dB mapping monotonically increasing from 00 to FF, re-mapping is introduced which reverses the volume value to dB mapping as well as normalizes the volume range to a more audible dB range. The current driver implementation assumes that all dB values under -60 dB are inaudible to the human ear. Re-Mapped values 00 H : -60 dB FF H : +12 dB

Remarks

None.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

    DRV_AK4954_VolumeSet(myAK4954Handle, DRV_AK4954_CHANNEL_LEFT, 120);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
chan	Audio channel volume to be set
volume	volume value specified in the range 0-255 (0x00 to 0xFF)

Function

void DRV_AK4954_VolumeSet(DRV_HANDLE handle, DRV_AK4954_CHANNEL channel, uint8_t volume);

DRV_AK4954_BufferAddRead Function

Schedule a non-blocking driver read operation.

File

drv_ak4954.h

C

```
void DRV_AK4954_BufferAddRead(const DRV_HANDLE handle, DRV_AK4954_BUFFER_HANDLE * bufferHandle, void *
buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4954_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4954_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4954_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4954_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4954 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4954 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 device instance and the DRV_AK4954_Status must have returned SYS_STATUS_READY.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_AK4954_Open call.

Parameters

Parameters	Description
handle	Handle of the AK4954 instance as return by the DRV_AK4954_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK4954_BufferAddRead

(

const DRV_HANDLE handle,

DRV_AK4954_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

DRV_AK4954_BufferAddWrite Function

Schedule a non-blocking driver write operation.

Implementation: Dynamic

File

drv_ak4954.h

C

```
void DRV_AK4954_BufferAddWrite(const DRV_HANDLE handle, DRV_AK4954_BUFFER_HANDLE * bufferHandle, void *
buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4954_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4954_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4954_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4954_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4954 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4954 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 device instance and the DRV_AK4954_Status must have returned SYS_STATUS_READY.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_AK4954_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK4954_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

// Client registers an event handler with driver

DRV_AK4954_BufferEventHandlerSet(myAK4954Handle,
                APP_AK4954BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4954_BufferAddWrite(myAK4954handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK4954_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
```

```
// the buffer is processed.

void APP_AK4954BufferEventHandler(DRV_AK4954_BUFFER_EVENT event,
        DRV_AK4954_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4954_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4954_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the AK4954 instance as return by the DRV_AK4954_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK4954_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_AK4954_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

DRV_AK4954_BufferAddWriteRead Function

Schedule a non-blocking driver write-read operation.

Implementation: Dynamic

File

drv_ak4954.h

C

```
void DRV_AK4954_BufferAddWriteRead(const DRV_HANDLE handle, DRV_AK4954_BUFFER_HANDLE * bufferHandle, void *
transmitBuffer, void * receiveBuffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK4954_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK4954_BUFFER_EVENT_COMPLETE:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only

- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK4954_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK4954_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK4954 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK4954 driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every AK4954 write. The transmit and receive size must be same.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 device instance and the DRV_AK4954_Status must have returned SYS_STATUS_READY.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_AK4954_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_AK4954_BUFFER_HANDLE bufferHandle;

// myak4954Handle is the handle returned
// by the DRV_AK4954_Open function.

// Client registers an event handler with driver

DRV_AK4954_BufferEventHandlerSet(myak4954Handle,
                APP_AK4954BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK4954_BufferAddWriteRead(myak4954handle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_AK4954_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK4954BufferEventHandler(DRV_AK4954_BUFFER_EVENT event,
        DRV_AK4954_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK4954_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK4954_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the AK4954 instance as returned by the DRV_AK4954_Open function
bufferHandle	Pointer to an argument that will contain the return buffer handle
transmitBuffer	The buffer where the transmit data will be stored
receiveBuffer	The buffer where the received data will be stored
size	Buffer size in bytes

Function

```
void DRV_AK4954_BufferAddWriteRead
(
const        DRV_HANDLE handle,
        DRV_AK4954_BUFFER_HANDLE *bufferHandle,
void *transmitBuffer,
void *receiveBuffer,
size_t size
)
```

DRV_AK4954_IntExtMicSet Function

This function sets up the codec for the X32 DB internal or the external microphone use.

File

drv_ak4954.h

C

```
void DRV_AK4954_IntExtMicSet(DRV_HANDLE handle, DRV_AK4954_INT_EXT_MIC micInput);
```

Returns

None

Description

This function sets up the codec for the internal or the external microphone use.

Remarks

None.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.
DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
micInput	Internal vs External mic input

Function

void DRV_AK4954_IntExtMicSet

DRV_AK4954_MicSet Function

This function sets up the codec for the internal or the AK4954 Mic1 or Mic2 input.

File

drv_ak4954.h

C

```
void DRV_AK4954_MicSet(DRV_HANDLE handle, DRV_AK4954_MIC micInput);
```

Returns

None

Description

This function sets up the codec.

Remarks

None.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.
DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
micInput	Internal vs External mic input

Function

void DRV_AK4954_IntMic12Set

DRV_AK4954_MonoStereoMicSet Function

This function sets up the codec for the Mono or Stereo microphone mode.

File

drv_ak4954.h

C

```
void DRV_AK4954_MonoStereoMicSet(DRV_HANDLE handle, DRV_AK4954_MONO_STEREO_MIC mono_stereo_mic);
```

Returns

None

Description

This function sets up the codec for the Mono or Stereo microphone mode.

Remarks

None.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.
DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4954_MonoStereoMicSet(DRV_HANDLE handle);

DRV_AK4954_MuteOff Function

This function disables AK4954 output for soft mute.

Implementation: Dynamic

File

drv_ak4954.h

C

```
void DRV_AK4954_MuteOff(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables AK4954 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

    DRV_AK4954_MuteOff(myAK4954Handle); //AK4954 output soft mute disabled
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4954_MuteOff(DRV_HANDLE handle)

DRV_AK4954_MuteOn Function

This function allows AK4954 output for soft mute on.

Implementation: Dynamic

File

drv_ak4954.h

C

```
void DRV_AK4954_MuteOn(DRV_HANDLE handle);
```

Returns

None.

Description

This function Enables AK4954 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK4954_Initialize routine must have been called for the specified AK4954 driver instance.

DRV_AK4954_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
```

```
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK4954Handle is the handle returned
// by the DRV_AK4954_Open function.

DRV_AK4954_MuteOn(myAK4954Handle);  //AK4954 output soft muted
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_AK4954_MuteOn(DRV_HANDLE handle);

d) Data Types and Constants

DRV_AK4954_AUDIO_DATA_FORMAT Enumeration

Identifies the Serial Audio data interface format.

File

drv_ak4954.h

C

```
typedef enum {
  DRV_AK4954_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_24BIT_LSB_SDTI = 0,
  DRV_AK4954_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_16BIT_LSB_SDTI,
  DRV_AK4954_AUDIO_DATA_FORMAT_24BIT_MSB_SDTO_24BIT_MSB_SDTI,
  DRV_AK4954_AUDIO_DATA_FORMAT_I2S_16BIT_24BIT,
  DRV_AK4954_AUDIO_DATA_FORMAT_32BIT_MSB_SDTO_32BIT_MSB_SDTI = 6,
  DRV_AK4954_AUDIO_DATA_FORMAT_I2S_32BIT
} DRV_AK4954_AUDIO_DATA_FORMAT;
```

Description

AK4954 Audio data format

This enumeration identifies Serial Audio data interface format.

DRV_AK4954_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_ak4954.h

C

```
typedef enum {
  DRV_AK4954_BUFFER_EVENT_COMPLETE,
  DRV_AK4954_BUFFER_EVENT_ERROR,
  DRV_AK4954_BUFFER_EVENT_ABORT
} DRV_AK4954_BUFFER_EVENT;
```

Members

Members	Description
DRV_AK4954_BUFFER_EVENT_COMPLETE	Data was transferred successfully.
DRV_AK4954_BUFFER_EVENT_ERROR	Error while processing the request
DRV_AK4954_BUFFER_EVENT_ABORT	Data transfer aborted (Applicable in DMA mode)

Description

AK4954 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the

DRV_AK4954_BufferAddWrite() function.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_AK4954_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_AK4954_BUFFER_EVENT_HANDLER Type

Pointer to a AK4954 Driver Buffer Event handler function

File

drv_ak4954.h

C

```
typedef void (* DRV_AK4954_BUFFER_EVENT_HANDLER)(DRV_AK4954_BUFFER_EVENT event, DRV_AK4954_BUFFER_HANDLE
bufferHandle, uintptr_t contextHandle);
```

Returns

None.

Description

AK4954 Driver Buffer Event Handler Function

This data type defines the required function signature for the AK4954 driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_AK4954_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_AK4954_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_AK4954_BufferProcessedSizeGet() function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4954_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver (I2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_AK4954_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void APP_MyBufferEventHandler( DRV_AK4954_BUFFER_EVENT event,
                               DRV_AK4954_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_AK4954_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_AK4954_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4954_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

File

drv_ak4954.h

C

```
typedef uintptr_t DRV_AK4954_BUFFER_HANDLE;
```

Description

AK4954 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_AK4954_BufferAddWrite() function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

DRV_AK4954_CHANNEL Enumeration

Identifies Left/Right Audio channel

File

drv_ak4954.h

C

```
typedef enum {
  DRV_AK4954_CHANNEL_LEFT,
  DRV_AK4954_CHANNEL_RIGHT,
  DRV_AK4954_CHANNEL_LEFT_RIGHT,
  DRV_AK4954_NUMBER_OF_CHANNELS
} DRV_AK4954_CHANNEL;
```

Description

AK4954 Audio Channel

This enumeration identifies Left/Right Audio channel

Remarks

None.

DRV_AK4954_COMMAND_EVENT_HANDLER Type

Pointer to a AK4954 Driver Command Event Handler Function

File

drv_ak4954.h

C

```
typedef void (* DRV_AK4954_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

Returns

None.

Description

AK4954 Driver Command Event Handler Function

This data type defines the required function signature for the AK4954 driver command event handling callback function.

A command is a control instruction to the AK4954 CODEC. Example Mute ON/OFF, Zero Detect Enable/Disable etc.

A client must register a pointer to a command event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK4954_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_AK4954CommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the event handling function.

DRV_AK4954_DIGITAL_BLOCK_CONTROL Enumeration

Identifies Bass-Boost Control function

File

drv_ak4954.h

C

```
typedef enum {
  DRV_AK4954_RECORDING_MODE,
  DRV_AK4954_PLAYBACK_MODE,
  DRV_AK4954_RECORDING_PLAYBACK_2_MODE,
  DRV_AK4954_LOOPBACK_MODE
} DRV_AK4954_DIGITAL_BLOCK_CONTROL;
```

Members

Members	Description
DRV_AK4954_RECORDING_MODE	This is the default setting
DRV_AK4954_PLAYBACK_MODE	Min control
DRV_AK4954_RECORDING_PLAYBACK_2_MODE	Medium control
DRV_AK4954_LOOPBACK_MODE	Maximum control

Description

AK4954 Bass-Boost Control

This enumeration identifies the settings for Bass-Boost Control function.

Remarks

None.

DRV_AK4954_INIT Structure

Defines the data required to initialize or reinitialize the AK4954 driver

File

drv_ak4954.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX i2sDriverModuleIndex;
  SYS_MODULE_INDEX i2cDriverModuleIndex;
  uint32_t samplingRate;
  uint8_t volume;
  DRV_AK4954_AUDIO_DATA_FORMAT audioDataFormat;
  bool delayDriverInitialization;
} DRV_AK4954_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX i2sDriverModuleIndex;	Identifies data module(I2S) driver ID for data interface of CODEC
SYS_MODULE_INDEX i2cDriverModuleIndex;	Identifies data module(I2C) driver ID for control interface of CODEC
uint32_t samplingRate;	Sampling rate
uint8_t volume;	Volume
DRV_AK4954_AUDIO_DATA_FORMAT audioDataFormat;	Identifies the Audio data format
bool delayDriverInitialization;	true if driver initialization should be delayed due to shared RESET pin

Description

AK4954 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the AK4954 CODEC driver.

Remarks

None.

DRV_AK4954_INT_EXT_MIC Enumeration

Identifies the Mic input source.

File

drv_ak4954.h

C

```
typedef enum {
  INT_MIC,
  EXT_MIC
} DRV_AK4954_INT_EXT_MIC;
```

Description

AK4954 Mic Internal / External Input

This enumeration identifies the Mic input source.

DRV_AK4954_MIC Enumeration

File

drv_ak4954.h

C

```
typedef enum {
  MIC1 = 0,
  MIC2,
```

```
  MIC3,
  DRV_AK4954_NUMBER_OF_MIC
} DRV_AK4954_MIC;
```

Members

Members	Description
MIC1 = 0	INT_MIC
MIC2	EXT_MIC
MIC3	LINE-IN

Description

This is type DRV_AK4954_MIC.

DRV_AK4954_MONO_STEREO_MIC Enumeration

Identifies the Mic input as Mono / Stereo.

File

drv_ak4954.h

C

```
typedef enum {
  ALL_ZEROS,
  MONO_RIGHT_CHANNEL,
  MONO_LEFT_CHANNEL,
  STEREO
} DRV_AK4954_MONO_STEREO_MIC;
```

Description

AK4954 Mic Mono / Stereo Input

This enumeration identifies the Mic input as Mono / Stereo.

DRV_AK4954_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_ak4954.h

C

```
#define DRV_AK4954_BUFFER_HANDLE_INVALID ((DRV_AK4954_BUFFER_HANDLE)(-1))
```

Description

AK4954 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_AK4954_BufferAddWrite() function if the buffer add request was not successful.

Remarks

None

DRV_AK4954_COUNT Macro

Number of valid AK4954 driver indices

File

drv_ak4954.h

C

```
#define DRV_AK4954_COUNT
```

Description

AK4954 Driver Module Count

This constant identifies the maximum number of AK4954 Driver instances that should be defined by the application. Defining more instances than

this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of AK4954 instances on this microcontroller.

Remarks

This value is part-specific.

DRV_AK4954_INDEX_0 Macro

AK4954 driver index definitions

File

drv_ak4954.h

C

```
#define DRV_AK4954_INDEX_0 0
```

Description

Driver AK4954 Module Index

These constants provide AK4954 driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_AK4954_Initialize and DRV_AK4954_Open routines to identify the driver instance in use.

DRV_AK4954_INDEX_1 Macro

File

drv_ak4954.h

C

```
#define DRV_AK4954_INDEX_1 1
```

Description

This is macro DRV_AK4954_INDEX_1.

DRV_AK4954_INDEX_2 Macro

File

drv_ak4954.h

C

```
#define DRV_AK4954_INDEX_2 2
```

Description

This is macro DRV_AK4954_INDEX_2.

DRV_AK4954_INDEX_3 Macro

File

drv_ak4954.h

C

```
#define DRV_AK4954_INDEX_3 3
```

Description

This is macro DRV_AK4954_INDEX_3.

DRV_AK4954_INDEX_4 Macro

File

drv_ak4954.h

C

```
#define DRV_AK4954_INDEX_4 4
```

Description

This is macro DRV_AK4954_INDEX_4.

DRV_AK4954_INDEX_5 Macro

File

drv_ak4954.h

C

```
#define DRV_AK4954_INDEX_5 5
```

Description

This is macro DRV_AK4954_INDEX_5.

Files

Files

Name	Description
drv_ak4954.h	AK4954 CODEC Driver Interface header file
drv_ak4954_config_template.h	AK4954 Codec Driver Configuration Template.

Description

This section lists the source and header files used by the AK4954Codec Driver Library.

drv_ak4954.h

AK4954 CODEC Driver Interface header file

Enumerations

	Name	Description
	DRV_AK4954_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK4954_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK4954_CHANNEL	Identifies Left/Right Audio channel
	DRV_AK4954_DIGITAL_BLOCK_CONTROL	Identifies Bass-Boost Control function
	DRV_AK4954_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK4954_MIC	This is type DRV_AK4954_MIC.
	DRV_AK4954_MONO_STEREO_MIC	Identifies the Mic input as Mono / Stereo.

Functions

	Name	Description
	DRV_AK4954_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK4954_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_AK4954_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_AK4954_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_AK4954_Close	Closes an opened-instance of the AK4954 driver. **Implementation:** Dynamic

	DRV_AK4954_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished. **Implementation:** Dynamic
	DRV_AK4954_Deinitialize	Deinitializes the specified instance of the AK4954 driver module. **Implementation:** Dynamic
	DRV_AK4954_Initialize	Initializes hardware and data for the instance of the AK4954 DAC module. **Implementation:** Dynamic
	DRV_AK4954_IntExtMicSet	This function sets up the codec for the X32 DB internal or the external microphone use.
	DRV_AK4954_MicSet	This function sets up the codec for the internal or the AK4954 Mic1 or Mic2 input.
	DRV_AK4954_MonoStereoMicSet	This function sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK4954_MuteOff	This function disables AK4954 output for soft mute. **Implementation:** Dynamic
	DRV_AK4954_MuteOn	This function allows AK4954 output for soft mute on. **Implementation:** Dynamic
	DRV_AK4954_Open	Opens the specified AK4954 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_AK4954_SamplingRateGet	This function gets the sampling rate set on the DAC AK4954. **Implementation:** Dynamic
	DRV_AK4954_SamplingRateSet	This function sets the sampling rate of the media stream. **Implementation:** Dynamic
	DRV_AK4954_SetAudioCommunicationMode	This function provides a run time audio format configuration
	DRV_AK4954_Status	Gets the current status of the AK4954 driver module. **Implementation:** Dynamic
	DRV_AK4954_Tasks	Maintains the driver's control and data interface state machine. **Implementation:** Dynamic
	DRV_AK4954_VersionGet	This function returns the version of AK4954 driver. **Implementation:** Dynamic
	DRV_AK4954_VersionStrGet	This function returns the version of AK4954 driver in string format. **Implementation:** Dynamic
	DRV_AK4954_VolumeGet	This function gets the volume for AK4954 CODEC. **Implementation:** Dynamic
	DRV_AK4954_VolumeSet	This function sets the volume for AK4954 CODEC. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_AK4954_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK4954_COUNT	Number of valid AK4954 driver indices
	DRV_AK4954_INDEX_0	AK4954 driver index definitions
	DRV_AK4954_INDEX_1	This is macro DRV_AK4954_INDEX_1.
	DRV_AK4954_INDEX_2	This is macro DRV_AK4954_INDEX_2.
	DRV_AK4954_INDEX_3	This is macro DRV_AK4954_INDEX_3.
	DRV_AK4954_INDEX_4	This is macro DRV_AK4954_INDEX_4.
	DRV_AK4954_INDEX_5	This is macro DRV_AK4954_INDEX_5.

Structures

	Name	Description
	DRV_AK4954_INIT	Defines the data required to initialize or reinitialize the AK4954 driver

Types

	Name	Description
	DRV_AK4954_BUFFER_EVENT_HANDLER	Pointer to a AK4954 Driver Buffer Event handler function
	DRV_AK4954_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK4954_COMMAND_EVENT_HANDLER	Pointer to a AK4954 Driver Command Event Handler Function

Description

AK4954 CODEC Driver Interface

The AK4954 CODEC device driver interface provides a simple interface to manage the AK4954 106dB 192kHz 24-Bit DAC that can be interfaced Microchip Microcontroller. This file provides the interface definition for the AK4954 CODEC device driver.

File Name

drv_AK4954.h

Company

Microchip Technology Inc.

drv_ak4954_config_template.h

AK4954 Codec Driver Configuration Template.

Macros

	Name	Description
	DRV_AK4954_BCLK_BIT_CLK_DIVISOR	Indicates whether the initilization of the AK4954 codec should be delayed.
	DRV_AK4954_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK4954_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to codec.
	DRV_AK4954_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK4954_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK Ratio to Generate Audio Stream for specified sampling frequency
	DRV_AK4954_MCLK_SOURCE	Indicate the input clock frequency to generate the MCLK to codec.
	DRV_AK4954_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.

Description

AK4954 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_ak4954_config_template.h

Company

Microchip Technology Inc.

AK7755 Codec Driver Library

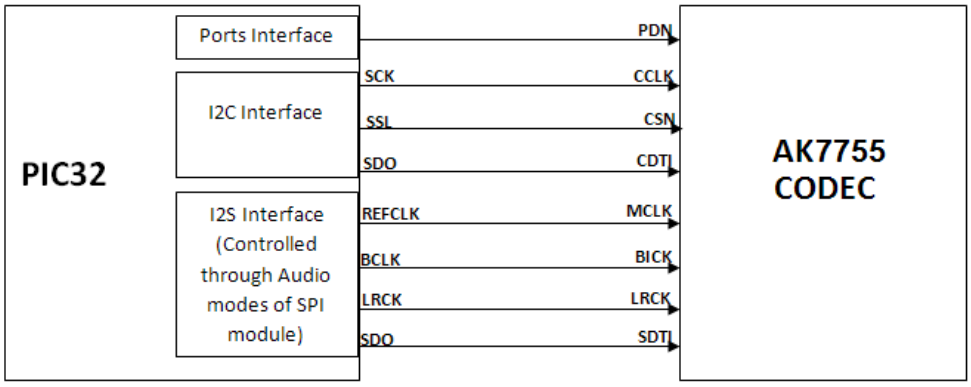
This topic describes the AK7755 Codec Driver Library.

Introduction

This library provides an interface to manage the AK7755 Codec that is serially interfaced to a Microchip microcontroller for providing Audio Solutions.

Description

The AK7755 module is 16/20/24-bit Audio Codec from Asahi Kasei Microdevices Corporation. The AK7755 can be interfaced to Microchip microcontrollers through I2C and I2S serial interfaces. The I2C interface is used for control command transfer. The I2S interface is used for Audio data output. A typical interface of the AK7755 Codec to a Microchip PIC32 device is provided in the following diagram:



Features

The AK7755 Codec supports the following features:

- Two Digital Interfaces (I/F1, I/F2):
  - 4-channel/6-channel Digital Signal Input Port: MSB justified 24-bit, LSB justified 24/20/16-bit, I2S
  - Short/Long Frame
  - 24-bit linear, 8-bit A-law, 8-bit µ-law
  - TDM 256 fs (8-channel) MSB Justified and I2S Formats
- SoftMute: On and Off
- Stereo 24-bit ADC:
  - Sampling Frequency: fs = 8 kHz ~96 kHz
  - ADC Characteristics S/(N+D): 91 dB, DR, S/N: 102 dB
  - Two-Channel Analog Input Selector (Differential, Single-ended Input)
  - Channel Independent Microphone Analog Gain Amplifier (0 ~18 dB (2 dB Step), 18 ~36 dB (3 dB Step))
  - Analog DRC (Dynamic Range Control)
  - Channel Independent Digital Volume (24 ~-103 dB, 0.5 dB Step Mute)
  - Digital HPF for DC Offset Cancelling
- Mono 24-bit ADC:
  - Sampling Frequency: 8 kHz ~ 96 kHz
  - ADC Characteristics S/(N+D): 90 dB; DR, S/N: 100 dB
  - Line Amplifier: 21 dB ~ -21 dB, 3 dB Step
  - Digital Volume (24 dB ~ -103 dB, 0.5 dB step, Mute)
  - Digital HPF for DC Offset Cancelling
- Stereo 24-bit DAC:
  - Sampling Frequency: fs = 8 kHz ~ 96 kHz
  - Digital Volume (12 dB ~ -115 dB, 0.5 step, Mute)
  - Digital De-emphasis Filter (tc = 50/15 µs, fs = 32 kHz, 44.1 kHz, 48 kHz)
- Master Clock: 2560 fs (internally generated by PLL from 32, 48, 64, 128, 256 and 384 fs clock)

Using the Library

This topic describes the basic architecture of the AK7755 Codec Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_AK7755.h`

The interface to the AK7755 Codec Driver library is defined in the `drv_AK7755.h` header file. Any C language source (`.c`) file that uses the AK7755 Codec Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

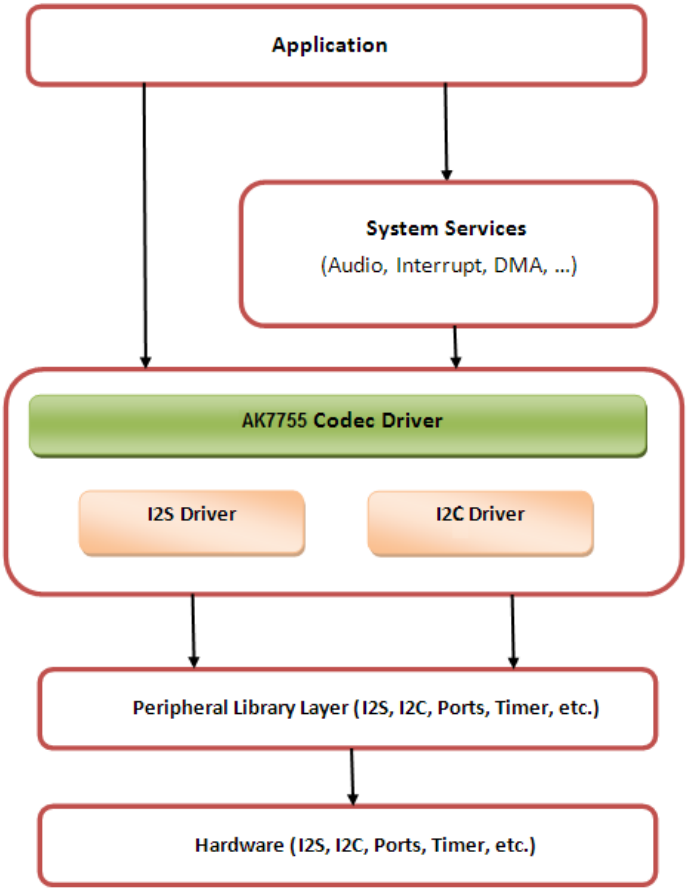
Abstraction Model

This library provides a low-level abstraction of the AK7755 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The abstraction model shown in the following diagram depicts how the AK7755 Codec Driver is positioned in the MPLAB Harmony framework. The AK7755 Codec Driver uses the SPI and I2S drivers for control and audio data transfers to the AK7755 module.

AK7755 Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The AK7755 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced AK7755 DAC module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the AK7755 Codec Driver Library.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Status Functions	Provides status functions.
Other Functions	Provides driver specific miscellaneous functions such as sampling rate setting, control command functions, etc.
Data Types and Constants	These data types and constants are required while interacting and setting up the AK7755 Codec Driver Library.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

System Access

This topic describes system initialization, implementations, and includes a system access code example.

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the AK7755 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_AK7755_INIT or by using Initialization Overrides) that are supported by the specific AK7755 device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver.
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver.
- Sampling rate
- Audio data format. The audio data format should match with the audio data format settings done in I2S driver initialization
- Power down pin port initialization
- Queue size for the audio data transmit buffer

The DRV_AK7755_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ AK7755_Deinitialize, DRV_ AK7755_Status and DRV_I2S_Tasks.

Implementations

The AK7755 Codec Driver can has the following implementation:

Description	MPLAB Harmony Components
Dedicated hardware for control (I2C) and data (I2S) interface.	Standard MPLAB Harmony drivers for I2C and I2S interfaces.

Example:

```
DRV_AK7755_INIT drvak7755Codec0InitData =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .i2sDriverModuleIndex = DRV_AK7755_I2S_DRIVER_MODULE_INDEX_IDX0,
    .i2cDriverModuleIndex = DRV_AK7755_I2C_DRIVER_MODULE_INDEX_IDX0,
    .volume = DRV_AK7755_VOLUME,
    .queueSizeTransmit = DRV_AK7755_TRANSMIT_QUEUE_SIZE,
};

// Initialize the I2C driver
   DRV_I2C0_Initialize();

// Initialize the I2S driver. The I2S module index should be same as the one used in initializing
// the I2S driver.
   sysObj.drvI2S0 = DRV_I2S_Initialize(DRV_I2S_INDEX_0, (SYS_MODULE_INIT *)&drvI2S0InitData);

// Initialize the Codec driver
   sysObj.drvak7755Codec0 = DRV_AK7755_Initialize(DRV_AK7755_INDEX_0, (SYS_MODULE_INIT
*)&drvak7755Codec0InitData);

if (SYS_MODULE_OBJ_INVALID == AK7755DevObject)
{
// Handle error
}
```

Task Routine

The DRV_AK7755_Tasks will be called from the System Task Service.

Client Access

For the application to start using an instance of the module, it must call the DRV_AK7755_Open function. The DRV_AK7755_Open provides a driver handle to the AK7755 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_AK7755_Deinitialize, the application must call the DRV_AK7755_Open function again to set up the instance of the driver.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

Client Operations

This topic provides information on client operations and includes a control command and audio buffered data operation flow diagram.

Description

Client operations provide the API interface for control command and audio data transfer to the AK7755 Codec.

The following AK7755 Codec specific control command functions are provided:

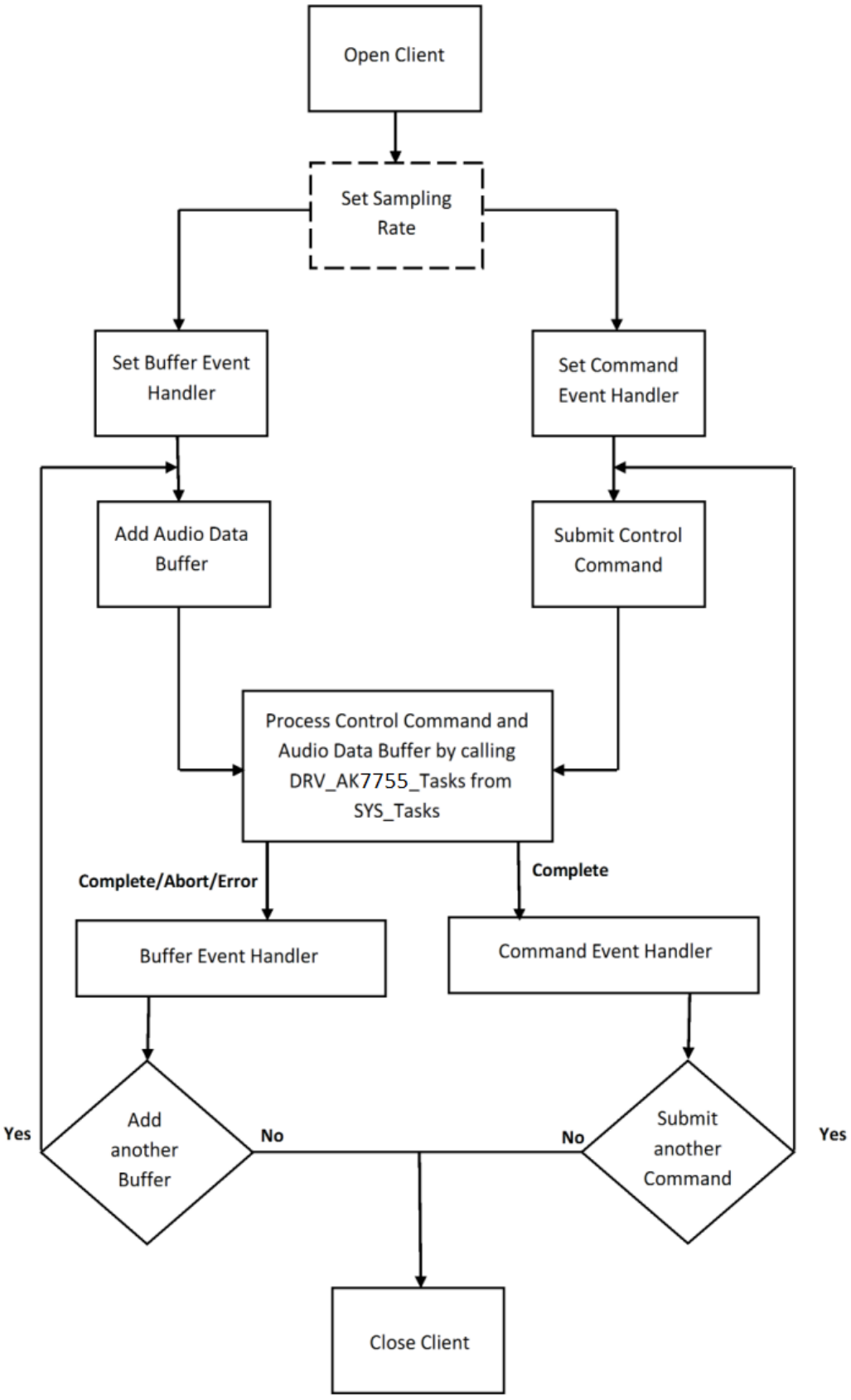
- DRV_AK7755_SamplingRateSet
- DRV_AK7755_SamplingRateGet
- DRV_AK7755_VolumeSet
- DRV_AK7755_VolumeGet
- DRV_AK7755_MuteOn
- DRV_AK7755_MuteOff
- DRV_AK7755_IntExtMicSet
- DRV_AK7755_MonoStereoMicSet

These functions schedule a non-blocking control command transfer operation. These functions submit the control command request to the AK7755 Codec. These functions submit the control command request to I2C Driver transmit queue, the request is processed immediately if it is the first request, or processed when the previous request is complete.

DRV_AK7755_BufferAddWrite, DRV_AK7755_BufferAddRead, and DRV_AK7755_BufferAddWriteRead are buffered data operation functions.

These functions schedule non-blocking audio data transfer operations. These functions add the request to I2S Driver transmit or receive buffer queue depends on the request type, and are executed immediately if it is the first buffer, or executed later when the previous buffer is complete. The driver notifies the client with DRV_AK7755_BUFFER_EVENT_COMPLETE, DRV_AK7755_BUFFER_EVENT_ERROR, or DRV_AK7755_BUFFER_EVENT_ABORT events.

The following diagram illustrates the control commands and audio buffered data operations.



Note: It is not necessary to close and reopen the client between multiple transfers.

Configuring the Library

Macros

	Name	Description
	DRV_AK7755_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK ratio to generate the audio stream for the specified sampling frequency.
	DRV_AK7755_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK7755_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to the codec.
	DRV_AK7755_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK7755_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK ratio to generate the audio stream for the specified sampling frequency.
	DRV_AK7755_MCLK_SOURCE	Indicates the input clock frequency to generate the MCLK to the codec.

Description

The configuration of the AK7755 Codec Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the AK7755 Codec Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the AK7755 Codec Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_AK7755_BCLK_BIT_CLK_DIVISOR Macro

Sets up the BCLK to LRCK ratio to generate the audio stream for the specified sampling frequency.

File

drv_ak7755_config_template.h

C

```
#define DRV_AK7755_BCLK_BIT_CLK_DIVISOR
```

Description

AK7755 BCLK to LRCK Ratio to Generate Audio Stream

This macro sets up the BCLK to LRCK ratio to generate the audio stream for the specified sampling frequency.

The following BCLK to LRCK ratios are supported:

- 16-bit data 16-bit channel: 32 fs; therefore, the divisor would be 8
- 16-bit data 32-bit channel: 64 fs; therefore, the divisor would be 4

Remarks

None.

DRV_AK7755_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_ak7755_config_template.h

C

```
#define DRV_AK7755_CLIENTS_NUMBER DRV_AK7755_INSTANCES_NUMBER
```

Description

AK7755 Client Count Configuration

This macro sets up the maximum number of clients that can be connected to any hardware instance. Typically only one client could be connected to one hardware instance. This value represents the total number of clients to be supported across all hardware instances. Therefore, if there are five AK7755 hardware interfaces, this number will be 5.

Remarks

None.

DRV_AK7755_INPUT_REFCLOCK Macro

Identifies the input REFCLOCK source to generate the MCLK to the codec.

File

drv_ak7755_config_template.h

C

```
#define DRV_AK7755_INPUT_REFCLOCK
```

Description

AK7755 Input reference clock

This macro identifies the input REFCLOCK source to generate the MCLK to the codec.

Remarks

None.

DRV_AK7755_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_ak7755_config_template.h

C

```
#define DRV_AK7755_INSTANCES_NUMBER
```

Description

AK7755 driver objects configuration

This macro sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of AK7755 Codec modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, the driver will be built statically.

Remarks

None.

DRV_AK7755_MCLK_SAMPLE_FREQ_MULTPLIER Macro

Sets up the MCLK to LRCK ratio to generate the audio stream for the specified sampling frequency.

File

drv_ak7755_config_template.h

C

```
#define DRV_AK7755_MCLK_SAMPLE_FREQ_MULTPLIER
```

Description

AK7755 MCLK to LRCK Ratio to Generate Audio Stream

This macro sets up the MCLK to LRCK ratio to generate the audio stream for the specified I2S sampling frequency.

The supported MCLK to sampling frequency ratios are as follows:

- 256 fs
- 384 fs
- 512 fs
- 768 fs
- 1152 fs

Remarks

None.

DRV_AK7755_MCLK_SOURCE Macro

Indicates the input clock frequency to generate the MCLK to the codec.

File

drv_ak7755_config_template.h

C

```
#define DRV_AK7755_MCLK_SOURCE
```

Description

AK7755 Data Interface Master Clock Speed configuration

This macro indicates the input clock frequency to generate the MCLK to the codec.

Remarks

None.

Configuring the MHC

Description

The following three figures show examples of MHC configurations for the AK7755 Codec Driver, I2S Driver, and the I2C Driver.

Figure 1: AK7755 Codec Driver MHC Configuration

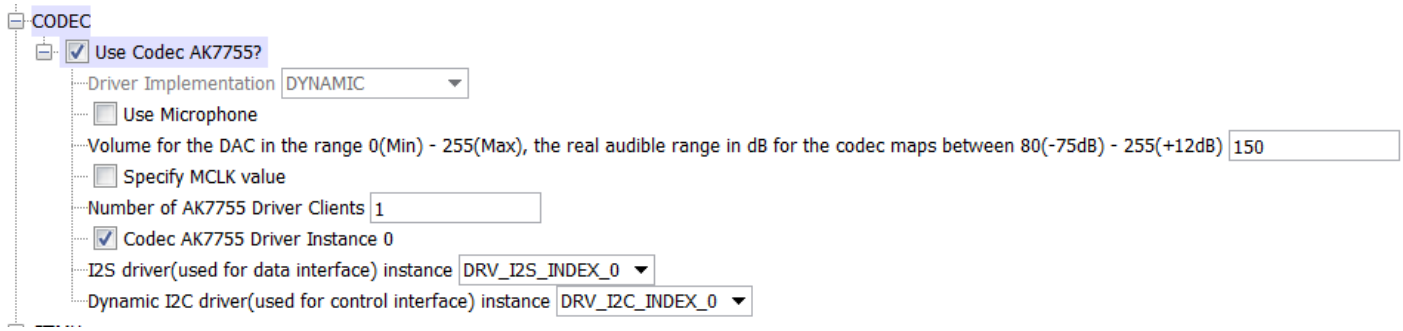


Figure 2: I2S Driver MHC Configuration

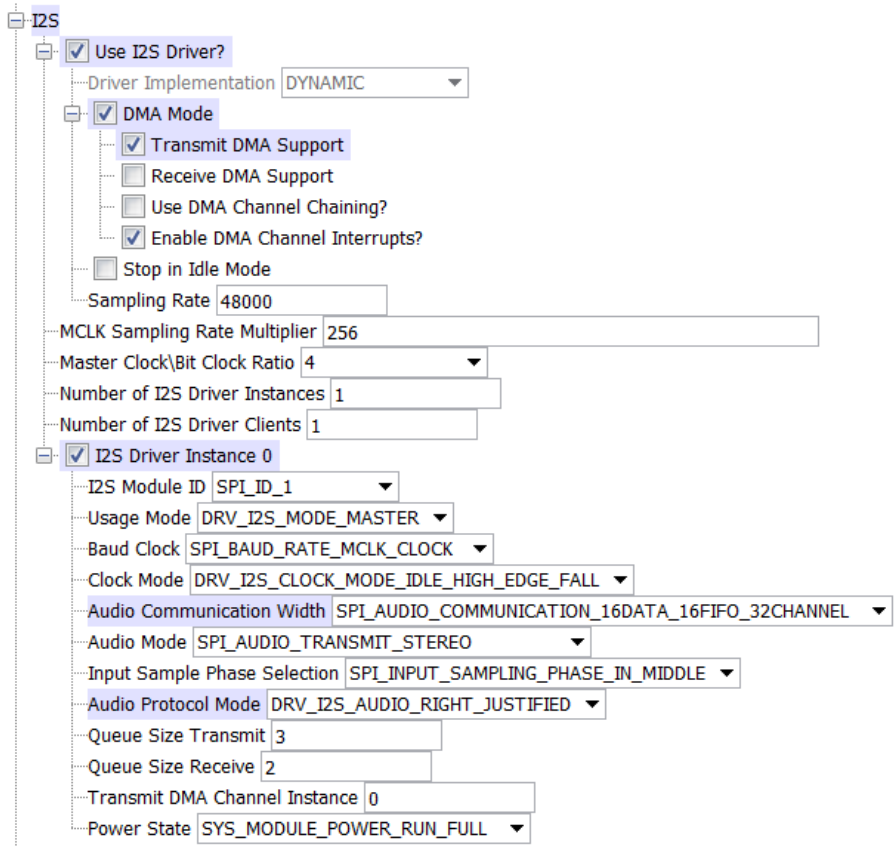
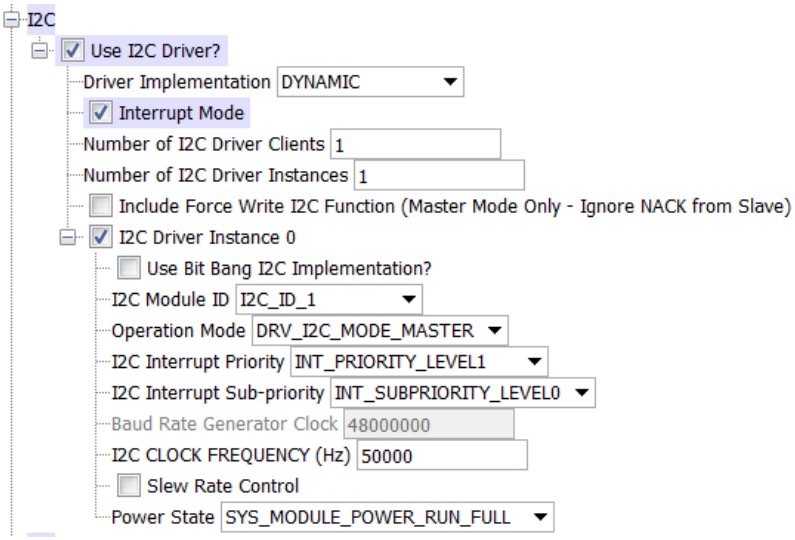


Figure 3: I2C Driver MHC Configuration



Migrating the AK7755 Driver From Earlier Versions of Microchip Harmony

Prior to version 1.08 of MPLAB Harmony, the AK7755 Codec Driver Library used the static I2C driver implementation. Beginning with v1.08 of MPLAB Harmony, applications must use the Dynamic Driver implementation with the MHC configured as shown in Figure 3. In addition, PIC32MZ configurations require the "Include Force Write I2C Function (Master Mode Only - Ignore NACK from Slave)" option to be selected.

Building the Library

This section lists the files that are available in the AK7755 Codec Driver Library.

Description

This section list the files that are available in the `/src` folder of the AK7755 Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/codec/ak7755`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_ak7755.h`	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_ak7755.c`	This file contains implementation of the AK7755 Codec Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The AK7755 Codec Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_AK7755_Close	Closes an opened-instance of the AK7755 Codec Driver.
	DRV_AK7755_Deinitialize	Deinitializes the specified instance of the AK7755 Codec Driver module.
	DRV_AK7755_Initialize	Initializes hardware and data for the instance of the AK7755 DAC module
	DRV_AK7755_Open	Opens the specified AK7755 Codec Driver instance and returns a handle to it
	DRV_AK7755_Tasks	Maintains the driver's control and data interface state machine.
	DRV_AK7755_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_AK7755_CommandEventHandlerSet	Allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.
	DRV_AK7755_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_AK7755_SetAudioCommunicationMode	This function provides a run time audio format configuration

b) Status Functions

	Name	Description
	DRV_AK7755_SamplingRateGet	This function gets the sampling rate set on the AK7755. **Implementation:** Dynamic
	DRV_AK7755_Status	Gets the current status of the AK7755 Codec Driver module.
	DRV_AK7755_VersionGet	Returns the version of the AK7755 Codec Driver.
	DRV_AK7755_VersionStrGet	This function returns the version of AK7755 Codec Driver in string format.
	DRV_AK7755_VolumeGet	Gets the volume for the AK7755 Codec Driver.

c) Other Functions

	Name	Description
	DRV_AK7755_VolumeSet	This function sets the volume for AK7755 CODEC.
	DRV_AK7755_BufferAddRead	Schedule a non-blocking driver read operation.

	DRV_AK7755_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_AK7755_BufferAddWriteRead	This is function DRV_AK7755_BufferAddWriteRead.
	DRV_AK7755_IntExtMicSet	Sets up the codec for the internal or the external microphone use.
	DRV_AK7755_MonoStereoMicSet	Sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK7755_MuteOff	Disables AK7755 output for soft mute.
	DRV_AK7755_MuteOn	Allows AK7755 output for soft mute on.

d) Data Types and Constants

	Name	Description
	_DRV_AK7755_H	Include files.
	DRV_AK7755_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK7755_COUNT	Number of valid AK7755 Codec Driver indices
	DRV_AK7755_INDEX_0	AK7755 driver index definitions
	DRV_AK7755_INDEX_1	This is macro DRV_AK7755_INDEX_1.
	DRV_AK7755_INDEX_2	This is macro DRV_AK7755_INDEX_2.
	DRV_AK7755_INDEX_3	This is macro DRV_AK7755_INDEX_3.
	DRV_AK7755_INDEX_4	This is macro DRV_AK7755_INDEX_4.
	DRV_AK7755_INDEX_5	This is macro DRV_AK7755_INDEX_5.
	DRV_AK7755_BICK_FS_FORMAT	This is type DRV_AK7755_BICK_FS_FORMAT.
	DRV_AK7755_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK7755_BUFFER_EVENT_HANDLER	Pointer to a AK7755 Driver Buffer Event handler function.
	DRV_AK7755_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK7755_CHANNEL	Identifies left/right audio channel.
	DRV_AK7755_COMMAND_EVENT_HANDLER	Pointer to a AK7755 Codec Driver command event handler function.
	DRV_AK7755_DAC_INPUT_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK7755_DSP_DIN1_INPUT_FORMAT	This is type DRV_AK7755_DSP_DIN1_INPUT_FORMAT.
	DRV_AK7755_DSP_DOUT1_OUTPUT_FORMAT	This is type DRV_AK7755_DSP_DOUT1_OUTPUT_FORMAT.
	DRV_AK7755_DSP_DOUT4_OUTPUT_FORMAT	This is type DRV_AK7755_DSP_DOUT4_OUTPUT_FORMAT.
	DRV_AK7755_DSP_PROGRAM	This is type DRV_AK7755_DSP_PROGRAM.
	DRV_AK7755_INIT	Defines the data required to initialize or reinitialize the AK7755 Codec Driver.
	DRV_AK7755_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK7755_LRCK_IF_FORMAT	This is type DRV_AK7755_LRCK_IF_FORMAT.
	DRV_AK7755_MONO_STEREO_MIC	Identifies the Mic input as Mono/Stereo.
	DRV_I2C_INDEX	This is macro DRV_I2C_INDEX.
	DATA_LENGTH	in bits
	SAMPLE_LENGTH	in bits

Description

This section describes the API functions of the AK7755 Codec Driver library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_AK7755_Close Function

Closes an opened-instance of the AK7755 Codec Driver.

File

drv_ak7755.h

C

```
void DRV_AK7755_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened-instance of the AK7755 Codec Driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this function, the handle passed in "handle" must not be used with any of the remaining driver functions. A new handle must be obtained by calling DRV_AK7755_Open before the caller may use the driver again.

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this function is called.

Preconditions

The DRV_AK7755_Initialize function must have been called for the specified AK7755 Codec Driver instance.

DRV_AK7755_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_AK7755_Open

DRV_AK7755_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

void DRV_AK7755_Close(DRV_Handle handle)

DRV_AK7755_Deinitialize Function

Deinitializes the specified instance of the AK7755 Codec Driver module.

File

drv_ak7755.h

C

```
void DRV_AK7755_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the specified instance of the AK7755 Codec Driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This function will NEVER block waiting for hardware.

Preconditions

The DRV_AK7755_Initialize function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_AK7755_Initialize
SYS_STATUS          status;

DRV_AK7755_Deinitialize(object-->);

status = DRV_AK7755_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4642_Initialize routine

Function

void DRV_AK7755_Deinitialize(SYS_MODULE_OBJ object)

DRV_AK7755_Initialize Function

Initializes hardware and data for the instance of the AK7755 DAC module

File

drv_ak7755.h

C

```
SYS_MODULE_OBJ DRV_AK7755_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes the AK7755 Codec Driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

Remarks

This function must be called before any other AK7755 function is called.

This function should only be called once during system initialization unless DRV_AK7755_Deinitialize is called to deinitialize the driver instance. This function will NEVER block for hardware access.

Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this codec driver. DRV_SPI_Initialize must be called if SPI driver is used for handling the control interface of this codec driver.

Example

```
DRV_AK7755_INIT                 init;
SYS_MODULE_OBJ                 objectHandle;

init->inUse                           = true;
init->status                          = SYS_STATUS_BUSY;
init->numClients                      = 0;
init->i2sDriverModuleIndex            = ak7755Init->i2sDriverModuleIndex;
init->i2cDriverModuleIndex            = ak7755Init->i2cDriverModuleIndex;
init->samplingRate                    = DRV_AK7755_AUDIO_SAMPLING_RATE;
init->audioDataFormat                 = DRV_AK7755_AUDIO_DATA_FORMAT_MACRO;

init->isInInterruptContext            = false;

init->commandCompleteCallback = (DRV_AK7755_COMMAND_EVENT_HANDLER)0;
init->commandContextData = 0;
init->mclk_multiplier = DRV_AK7755_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_AK7755_Initialize(DRV_AK7755_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized

init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

SYS_MODULE_OBJ DRV_AK7755_Initialize

(

const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT *const init

);

DRV_AK7755_Open Function

Opens the specified AK7755 Codec Driver instance and returns a handle to it

File

drv_ak7755.h

C

```
DRV_HANDLE DRV_AK7755_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the function returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_AK7755_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

Description

This function opens the specified AK7755 Codec Driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

Only DRV_IO_INTENT_WRITE is a valid ioIntent option as AK7755 is DAC only.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_AK7755_Close function is called. This function will NEVER block waiting for hardware.If the requested intent flags are not supported, the function will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

The DRV_AK7755_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_AK7755_Open(DRV_AK7755_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

DRV_HANDLE DRV_AK7755_Open
(
const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT ioIntent
)

DRV_AK7755_Tasks Function

Maintains the driver's control and data interface state machine.

File

drv_ak7755.h

C

```
void DRV_AK7755_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks function (SYS_Tasks).

Preconditions

The DRV_AK7755_Initialize function must have been called for the specified AK7755 Codec Driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK7755_Initialize

while (true)
{
    DRV_AK7755_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_AK7755_Initialize)

Function

void DRV_AK7755_Tasks(SYS_MODULE_OBJ object);

DRV_AK7755_BufferEventHandlerSet Function

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

File

drv_ak7755.h

C

```
void DRV_AK7755_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_AK7755_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_AK7755_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Preconditions

The DRV_AK7755_Initialize function must have been called for the specified AK7755 Codec Driver instance.

DRV_AK7755_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK7755_BUFFER_HANDLE bufferHandle;

// myAK7755Handle is the handle returned
// by the DRV_AK7755_Open function.

// Client registers an event handler with driver

DRV_AK7755_BufferEventHandlerSet(myAK7755Handle,
                APP_AK7755BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK7755_BufferAddWrite(myAK7755handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK7755_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_AK7755BufferEventHandler(DRV_AK7755_BUFFER_EVENT event,
        DRV_AK7755_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK7755_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK7755_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function.

context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK7755_BufferEventHandlerSet
(
DRV_HANDLE handle,
const DRV_AK7755_BUFFER_EVENT_HANDLER eventHandler,
const uintptr_t contextHandle
)

DRV_AK7755_CommandEventHandlerSet Function

Allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

File

drv_ak7755.h

C

```
void DRV_AK7755_CommandEventHandlerSet(DRV_HANDLE handle, const DRV_AK7755_COMMAND_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

When a client calls DRV_AK7755_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "AK7755 CODEC Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Preconditions

The DRV_AK7755_Initialize function must have been called for the specified AK7755 Codec Driver instance.

DRV_AK7755_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK7755_BUFFER_HANDLE bufferHandle;

// myAK7755Handle is the handle returned
// by the DRV_AK7755_Open function.

// Client registers an event handler with driver

DRV_AK7755_CommandEventHandlerSet(myAK7755Handle,
                APP_AK7755CommandEventHandler, (uintptr_t)&myAppObj);

DRV_AK7755_DeEmphasisFilterSet(myAK7755Handle, DRV_AK7755_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_AK7755CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.
```

```
    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_AK7755_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_AK7755_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

DRV_AK7755_SamplingRateSet Function

This function sets the sampling rate of the media stream.

File

drv_ak7755.h

C

```
void DRV_AK7755_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

Returns

None.

Description

This function sets the media sampling rate for the client handle.

Remarks

None.

Preconditions

The DRV_AK7755_Initialize function must have been called for the specified AK7755 Codec Driver instance.

DRV_AK7755_Open must have been called to obtain a valid opened device handle.

Example

```
// myAK7755Handle is the handle returned
// by the DRV_AK7755_Open function.

DRV_AK7755_SamplingRateSet(myAK7755Handle, 48000);    //Sets 48000 media sampling rate
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
samplingRate	Sampling frequency in Hz

Function

void DRV_AK7755_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

DRV_AK7755_SetAudioCommunicationMode Function

This function provides a run time audio format configuration

File

drv_ak7755.h

C

```
void DRV_AK7755_SetAudioCommunicationMode(DRV_HANDLE handle, const DATA_LENGTH dl, const SAMPLE_LENGTH sl);
```

Returns

None

Description

This function sets up audio mode in I2S protocol

Remarks

None.

Preconditions

The DRV_AK7755_Initialize routine must have been called for the specified AK7755 driver instance.

DRV_AK7755_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
dl	Data length for I2S audio interface
sl	Left/Right Sample Length for I2S audio interface

Function

void DRV_AK7755_SetAudioCommunicationMode
(
DRV_HANDLE handle,
const DATA_LENGTH dl,
const SAMPLE_LENGTH sl
)

b) Status Functions

DRV_AK7755_SamplingRateGet Function

This function gets the sampling rate set on the AK7755.

Implementation: Dynamic

File

drv_ak7755.h

C

```
uint32_t DRV_AK7755_SamplingRateGet(DRV_HANDLE handle);
```

Description

This function gets the sampling rate set on the DAC AK7755.

Remarks

None.

Example

```
uint32_t baudRate;
```

```
// myAK7755Handle is the handle returned
// by the DRV_AK7755_Open function.

baudRate = DRV_AK7755_SamplingRateGet(myAK7755Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

uint32_t DRV_AK7755_SamplingRateGet(DRV_HANDLE handle)

DRV_AK7755_Status Function

Gets the current status of the AK7755 Codec Driver module.

File

drv_ak7755.h

C

```
SYS_STATUS DRV_AK7755_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized
- SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed
- SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed
- SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This function provides the current status of the AK7755 Codec Driver module.

Remarks

A driver can be opened only when its status is SYS_STATUS_READY.

Preconditions

The DRV_AK7755_Initialize function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_AK7755_Initialize
SYS_STATUS          AK7755Status;

AK7755Status = DRV_AK7755_Status(object);
if (SYS_STATUS_READY == AK7755Status)
{
    // This means the driver can be opened using the
    // DRV_AK7755_Open function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_AK4642_Initialize routine

Function

SYS_STATUS DRV_AK7755_Status(SYS_MODULE_OBJ object)

DRV_AK7755_VersionGet Function

Returns the version of the AK7755 Codec Driver.

File

drv_ak7755.h

C

```
uint32_t DRV_AK7755_VersionGet();
```

Returns

Returns the version of the AK7755 Codec Driver.

Description

The version number returned from the DRV_AK7755_VersionGet function is an unsigned integer in the following decimal format:

- * 10000 + * 100 +

Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

Remarks

None.

Preconditions

None.

Example 1

- For version "0.03a", return: 0 * 10000 + 3 * 100 + 0
- For version "1.00", return: 1 * 100000 + 0 * 100 + 0

Example 2

```
uint32_t AK7755version;
AK7755version = DRV_AK7755_VersionGet();
```

Function

uint32_t DRV_AK7755_VersionGet(void)

DRV_AK7755_VersionStrGet Function

This function returns the version of AK7755 Codec Driver in string format.

File

drv_ak7755.h

C

```
int8_t* DRV_AK7755_VersionStrGet();
```

Returns

returns a string containing the version of the AK7755 Codec Driver.

Description

The DRV_AK7755_VersionStrGet function returns a string in the format: ".[.][]" Where:

- is the AK7755 Codec Driver's version number.
- is the AK7755 Codec Driver's version number.
- is an optional "patch" or "dot" release number (which is not

included in the string if it equals "00").

- is an optional release type ("a" for alpha, "b" for beta ?

not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces. For example, "0.03a" "1.00"

Remarks

None

Preconditions

None.

Example

```
int8_t *AK7755string;
AK7755string = DRV_AK7755_VersionStrGet();
```

Function

int8_t* DRV_AK7755_VersionStrGet(void)

DRV_AK7755_VolumeGet Function

Gets the volume for the AK7755 Codec Driver.

File

drv_ak7755.h

C

```
uint8_t DRV_AK7755_VolumeGet(DRV_HANDLE handle, DRV_AK7755_CHANNEL channel);
```

Returns

None.

Description

This functions gets the current volume programmed to the AK7755 Codec Driver.

Remarks

None.

Preconditions

The DRV_AK7755_Initialize function must have been called for the specified AK7755 Codec Driver instance.

DRV_AK7755_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myAK7755Handle is the handle returned
// by the DRV_AK7755_Open function.

  volume = DRV_AK7755_VolumeGet(myAK7755Handle, DRV_AK7755_CHANNEL_LEFT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
channel	argument indicating Left or Right or Both channel volume to be modified

Function

uint8_t DRV_AK7755_VolumeGet(DRV_HANDLE handle, DRV_AK7755_CHANNEL channel)

c) Other Functions

DRV_AK7755_VolumeSet Function

This function sets the volume for AK7755 CODEC.

File

drv_ak7755.h

C

```
void DRV_AK7755_VolumeSet(DRV_HANDLE handle, DRV_AK7755_CHANNEL channel, uint8_t volume);
```

Returns

None

Description

This functions sets the volume value from 0-255, which can attenuate from -115 dB to +12 dB. All decibels below approximately -50 dB are inaudible.

Remarks

None.

Preconditions

The DRV_AK7755_Initialize function must have been called for the specified AK7755 Codec Driver instance.

DRV_AK7755_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK7755Handle is the handle returned
// by the DRV_AK7755_Open function.

    DRV_AK7755_VolumeSet(myAK7755Handle,DRV_AK7755_CHANNEL_LEFT, 120);    //Step 120 volume
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's Open function
channel	argument indicating Left or Right or Both channel volume to be modified
volume	Updated volume specified in the range 0-255

Function

void DRV_AK7755_VolumeSet(DRV_HANDLE handle, DRV_AK7755_CHANNEL channel, uint8_t volume);

DRV_AK7755_BufferAddRead Function

Schedule a non-blocking driver read operation.

File

drv_ak7755.h

C

```
void DRV_AK7755_BufferAddRead(const DRV_HANDLE handle, DRV_AK7755_BUFFER_HANDLE * bufferHandle, void *
buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK7755_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK7755_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK7755_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK7755_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK7755 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK7755 Codec Driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK7755_Initialize function must have been called for the specified AK7755 device instance and the DRV_AK7755_Status must have returned SYS_STATUS_READY.

DRV_AK7755_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_AK7755_Open call.

Parameters

Parameters	Description
handle	Handle of the AK7755 instance as return by the DRV_AK7755_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK7755_BufferAddRead

(

const DRV_HANDLE handle,

DRV_AK7755_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

DRV_AK7755_BufferAddWrite Function

Schedule a non-blocking driver write operation.

File

drv_ak7755.h

C

```
void DRV_AK7755_BufferAddWrite(const DRV_HANDLE handle, DRV_AK7755_BUFFER_HANDLE * bufferHandle, void *
buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_AK7755_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_AK7755_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_AK7755_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_AK7755_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the AK7755 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another AK7755 Codec Driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_AK7755_Initialize function must have been called for the specified AK7755 device instance and the DRV_AK7755_Status must have

returned SYS_STATUS_READY.

DRV_AK7755_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_AK7755_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_AK7755_BUFFER_HANDLE bufferHandle;

// myAK7755Handle is the handle returned
// by the DRV_AK7755_Open function.

// Client registers an event handler with driver

DRV_AK7755_BufferEventHandlerSet(myAK7755Handle,
                APP_AK7755BufferEventHandler, (uintptr_t)&myAppObj);

DRV_AK7755_BufferAddWrite(myAK7755handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_AK7755_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_AK7755BufferEventHandler(DRV_AK7755_BUFFER_EVENT event,
        DRV_AK7755_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_AK7755_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_AK7755_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the AK7755 instance as return by the DRV_AK7755_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_AK7755_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_AK7755_BUFFER_HANDLE *bufferHandle,

void *buffer, size_t size

)

DRV_AK7755_BufferAddWriteRead Function

File

drv_ak7755.h

C

```
void DRV_AK7755_BufferAddWriteRead(const DRV_HANDLE handle, DRV_AK7755_BUFFER_HANDLE * bufferHandle, void *
transmitBuffer, void * receiveBuffer, size_t size);
```

Description

This is function DRV_AK7755_BufferAddWriteRead.

DRV_AK7755_IntExtMicSet Function

Sets up the codec for the internal or the external microphone use.

File

drv_ak7755.h

C

```
void DRV_AK7755_IntExtMicSet(DRV_HANDLE handle, DRV_AK7755_INT_EXT_MIC micInput);
```

Returns

None.

Description

This function sets up the codec for the internal or the external microphone use.

Remarks

None.

Preconditions

The DRV_AK7755_Initialize function must have been called for the specified AK7755 Codec Driver instance.

DRV_AK7755_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
micInput	Internal vs. External microphone input

Function

void DRV_AK7755_IntExtMicSet(DRV_HANDLE handle);

DRV_AK7755_MonoStereoMicSet Function

Sets up the codec for the Mono or Stereo microphone mode.

File

drv_ak7755.h

C

```
void DRV_AK7755_MonoStereoMicSet(DRV_HANDLE handle, DRV_AK7755_MONO_STEREO_MIC mono_stereo_mic);
```

Returns

None.

Description

This function sets up the codec for the Mono or Stereo microphone mode.

Remarks

None.

Preconditions

The DRV_AK7755_Initialize function must have been called for the specified AK7755 Codec Driver instance.

DRV_AK7755_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
mono_stereo_mic	Mono/Stereo microphone setup

Function

void DRV_AK7755_MonoStereoMicSet(DRV_HANDLE handle);

DRV_AK7755_MuteOff Function

Disables AK7755 output for soft mute.

File

drv_ak7755.h

C

```
void DRV_AK7755_MuteOff(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables AK7755 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK7755_Initialize function must have been called for the specified AK7755 Codec Driver instance.

DRV_AK7755_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK7755Handle is the handle returned
// by the DRV_AK7755_Open function.

    DRV_AK7755_MuteOff(myAK7755Handle);    //AK7755 output soft mute disabled
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

void DRV_AK7755_MuteOff(DRV_HANDLE handle)

DRV_AK7755_MuteOn Function

Allows AK7755 output for soft mute on.

File

drv_ak7755.h

C

```
void DRV_AK7755_MuteOn(DRV_HANDLE handle);
```

Returns

None.

Description

This function enables AK7755 output for soft mute.

Remarks

None.

Preconditions

The DRV_AK7755_Initialize function must have been called for the specified AK7755 Codec Driver instance.

DRV_AK7755_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myAK7755Handle is the handle returned
// by the DRV_AK7755_Open function.

DRV_AK7755_MuteOn(myAK7755Handle);    //AK7755 output soft muted
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

void DRV_AK7755_MuteOn(DRV_HANDLE handle);

d) Data Types and Constants

_DRV_AK7755_H Macro

File

drv_ak7755.h

C

```
#define _DRV_AK7755_H
```

Description

Include files.

DRV_AK7755_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_ak7755.h

C

```
#define DRV_AK7755_BUFFER_HANDLE_INVALID ((DRV_AK7755_BUFFER_HANDLE)(-1))
```

Description

AK7755 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_AK7755_BufferAddWrite and the DRV_AK7755_BufferAddRead function if the buffer add request was not successful.

Remarks

None.

DRV_AK7755_COUNT Macro

Number of valid AK7755 Codec Driver indices

File

drv_ak7755.h

C

```
#define DRV_AK7755_COUNT
```

Description

AK7755 Driver Module Count

This constant identifies the maximum number of AK7755 Codec Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of AK7755 instances on this microcontroller.

Remarks

This value is device-specific.

DRV_AK7755_INDEX_0 Macro

AK7755 driver index definitions

File

drv_ak7755.h

C

```
#define DRV_AK7755_INDEX_0 0
```

Description

Driver AK7755 Module Index

These constants provide AK7755 Codec Driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_AK7755_Initialize and DRV_AK7755_Open functions to identify the driver instance in use.

DRV_AK7755_INDEX_1 Macro

File

drv_ak7755.h

C

```
#define DRV_AK7755_INDEX_1 1
```

Description

This is macro DRV_AK7755_INDEX_1.

DRV_AK7755_INDEX_2 Macro

File

drv_ak7755.h

C

```
#define DRV_AK7755_INDEX_2 2
```

Description

This is macro DRV_AK7755_INDEX_2.

DRV_AK7755_INDEX_3 Macro

File

drv_ak7755.h

C

```
#define DRV_AK7755_INDEX_3 3
```

Description

This is macro DRV_AK7755_INDEX_3.

DRV_AK7755_INDEX_4 Macro

File

drv_ak7755.h

C

```
#define DRV_AK7755_INDEX_4 4
```

Description

This is macro DRV_AK7755_INDEX_4.

DRV_AK7755_INDEX_5 Macro

File

drv_ak7755.h

C

```
#define DRV_AK7755_INDEX_5 5
```

Description

This is macro DRV_AK7755_INDEX_5.

DRV_AK7755_BICK_FS_FORMAT Enumeration

File

drv_ak7755.h

C

```
typedef enum {
  DRV_AK7755_BICK_64FS,
  DRV_AK7755_BICK_48FS,
  DRV_AK7755_BICK_32FS,
  DRV_AK7755_BICK_256FS
} DRV_AK7755_BICK_FS_FORMAT;
```

Description

This is type DRV_AK7755_BICK_FS_FORMAT.

DRV_AK7755_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_ak7755.h

C

```
typedef enum {
  DRV_AK7755_BUFFER_EVENT_COMPLETE,
  DRV_AK7755_BUFFER_EVENT_ERROR,
  DRV_AK7755_BUFFER_EVENT_ABORT
} DRV_AK7755_BUFFER_EVENT;
```

Members

Members	Description
DRV_AK7755_BUFFER_EVENT_COMPLETE	Data was transferred successfully.
DRV_AK7755_BUFFER_EVENT_ERROR	Error while processing the request
DRV_AK7755_BUFFER_EVENT_ABORT	Data transfer aborted (Applicable in DMA mode)

Description

AK7755 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_AK7755_BufferAddWrite or the DRV_AK7755_BufferAddRead function.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_AK7755_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_AK7755_BUFFER_EVENT_HANDLER Type

Pointer to a AK7755 Driver Buffer Event handler function.

File

drv_ak7755.h

C

```
typedef void (* DRV_AK7755_BUFFER_EVENT_HANDLER)(DRV_AK7755_BUFFER_EVENT event, DRV_AK7755_BUFFER_HANDLE
bufferHandle, uintptr_t contextHandle);
```

Returns

None.

Description

AK7755 Driver Buffer Event Handler Function

This data type defines the required function signature for the AK7755 Codec Driver buffer event handling callback function. A client must register a pointer to a buffer event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_AK7755_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_AK7755_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_AK7755_BufferProcessedSizeGet function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK7755_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver (i.e., I2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_AK7755_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void APP_MyBufferEventHandler( DRV_AK7755_BUFFER_EVENT event,
                               DRV_AK7755_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_AK7755_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_AK7755_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_AK7755_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

File

drv_ak7755.h

C

```
typedef uintptr_t DRV_AK7755_BUFFER_HANDLE;
```

Description

AK7755 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_AK7755_BufferAddWrite or DRV_AK7755_BufferAddRead function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_AK7755_CHANNEL Enumeration

Identifies left/right audio channel.

File

drv_ak7755.h

C

```
typedef enum {
  DRV_AK7755_CHANNEL_LEFT,
  DRV_AK7755_CHANNEL_RIGHT,
  DRV_AK7755_CHANNEL_LEFT_RIGHT,
  DRV_AK7755_NUMBER_OF_CHANNELS
} DRV_AK7755_CHANNEL;
```

Description

AK7755 Audio Channel

This enumeration identifies the left/right audio channel.

Remarks

None.

DRV_AK7755_COMMAND_EVENT_HANDLER Type

Pointer to a AK7755 Codec Driver command event handler function.

File

drv_ak7755.h

C

```
typedef void (* DRV_AK7755_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

Returns

None.

Description

AK7755 Driver Command Event Handler Function

This data type defines the required function signature for the AK7755 Codec Driver command event handling callback function.

A command is a control instruction to the AK7755 Codec. For example, Mute ON/OFF, Zero Detect Enable/Disable, etc.

A client must register a pointer to a command event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_AK7755_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_AK7755CommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the event handling function.

DRV_AK7755_DAC_INPUT_FORMAT Enumeration

Identifies the Serial Audio data interface format.

File

drv_ak7755.h

C

```
typedef enum {
  DRV_AK7755_DAC_INPUT_24BITMSB,
  DRV_AK7755_DAC_INPUT_24BITLSB,
```

```
  DRV_AK7755_DAC_INPUT_20BITLSB,
  DRV_AK7755_DAC_INPUT_16BITLSB
} DRV_AK7755_DAC_INPUT_FORMAT;
```

Members

Members	Description
DRV_AK7755_DAC_INPUT_20BITLSB	not supported

Description

AK7755 Audio Data Format

This enumeration identifies the Serial Audio data interface format.

DRV_AK7755_DSP_DIN1_INPUT_FORMAT Enumeration

File

drv_ak7755.h

C

```
typedef enum {
  DRV_AK7755_DSP_DIN1_INPUT_24BITMSB,
  DRV_AK7755_DSP_DIN1_INPUT_24BITLSB,
  DRV_AK7755_DSP_DIN1_INPUT_20BITLSB,
  DRV_AK7755_DSP_DIN1_INPUT_16BITLSB
} DRV_AK7755_DSP_DIN1_INPUT_FORMAT;
```

Description

This is type DRV_AK7755_DSP_DIN1_INPUT_FORMAT.

DRV_AK7755_DSP_DOUT1_OUTPUT_FORMAT Enumeration

File

drv_ak7755.h

C

```
typedef enum {
  DRV_AK7755_DSP_DOUT1_OUTPUT_24BITMSB,
  DRV_AK7755_DSP_DOUT1_OUTPUT_24BITLSB,
  DRV_AK7755_DSP_DOUT1_OUTPUT_20BITLSB,
  DRV_AK7755_DSP_DOUT1_OUTPUT_16BITLSB
} DRV_AK7755_DSP_DOUT1_OUTPUT_FORMAT;
```

Description

This is type DRV_AK7755_DSP_DOUT1_OUTPUT_FORMAT.

DRV_AK7755_DSP_DOUT4_OUTPUT_FORMAT Enumeration

File

drv_ak7755.h

C

```
typedef enum {
  DRV_AK7755_DSP_DOUT4_OUTPUT_24BITMSB,
  DRV_AK7755_DSP_DOUT4_OUTPUT_24BITLSB,
  DRV_AK7755_DSP_DOUT4_OUTPUT_20BITLSB,
  DRV_AK7755_DSP_DOUT4_OUTPUT_16BITLSB
} DRV_AK7755_DSP_DOUT4_OUTPUT_FORMAT;
```

Description

This is type DRV_AK7755_DSP_DOUT4_OUTPUT_FORMAT.

DRV_AK7755_DSP_PROGRAM Enumeration

File

drv_ak7755.h

C

```
typedef enum {
  DRV_AK7755_DSP_ECHO_CANCELLATION,
  DRV_AK7755_DSP_REGULAR
} DRV_AK7755_DSP_PROGRAM;
```

Description

This is type DRV_AK7755_DSP_PROGRAM.

DRV_AK7755_INIT Structure

Defines the data required to initialize or reinitialize the AK7755 Codec Driver.

File

drv_ak7755.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX i2sDriverModuleIndex;
  SYS_MODULE_INDEX i2cDriverModuleIndex;
  uint32_t samplingRate;
  uint8_t volume;
} DRV_AK7755_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX i2sDriverModuleIndex;	Identifies data module (I2S) driver ID for data interface of CODEC
SYS_MODULE_INDEX i2cDriverModuleIndex;	Identifies data module (I2C) driver ID for control interface of CODEC
uint32_t samplingRate;	Sampling rate
uint8_t volume;	Volume

Description

AK7755 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the AK7755 Codec Driver.

Remarks

None.

DRV_AK7755_INT_EXT_MIC Enumeration

Identifies the Mic input source.

File

drv_ak7755.h

C

```
typedef enum {
  INT_MIC,
  EXT_MIC
} DRV_AK7755_INT_EXT_MIC;
```

Description

AK7755 Mic Internal / External Input

This enumeration identifies the Mic input source.

Remarks

None.

DRV_AK7755_LRCK_IF_FORMAT Enumeration

File

drv_ak7755.h

C

```
typedef enum {
  DRV_AK7755_LRCK_IF_STANDARD,
  DRV_AK7755_LRCK_IF_I2S_COMPATIBLE,
  DRV_AK7755_LRCK_IF_PCM_SHORT_FRAME,
  DRV_AK7755_LRCK_IF_PCM_LONG_FRAME
} DRV_AK7755_LRCK_IF_FORMAT;
```

Description

This is type DRV_AK7755_LRCK_IF_FORMAT.

DRV_AK7755_MONO_STEREO_MIC Enumeration

Identifies the Mic input as Mono/Stereo.

File

drv_ak7755.h

C

```
typedef enum {
  ALL_ZEROS,
  MONO_RIGHT_CHANNEL,
  MONO_LEFT_CHANNEL,
  STEREO
} DRV_AK7755_MONO_STEREO_MIC;
```

Description

AK7755 Mic Mono/Stereo Input

This enumeration identifies the Mic input as Mono/Stereo.

Remarks

None.

DRV_I2C_INDEX Macro

File

drv_wm8904.h

C

```
#define DRV_I2C_INDEX DRV_WM8904_I2C_INSTANCES_NUMBER
```

Description

This is macro DRV_I2C_INDEX.

DATA_LENGTH Enumeration

File

drv_wm8904.h

C

```
typedef enum {
  DATA_LENGTH_16,
  DATA_LENGTH_24,
  DATA_LENGTH_32
```

```
} DATA_LENGTH;
```

Description

in bits

SAMPLE_LENGTH Enumeration

File

drv_ak7755.h

C

```
typedef enum {
  SAMPLE_LENGTH_16,
  SAMPLE_LENGTH_32
} SAMPLE_LENGTH;
```

Description

in bits

Files

Files

Name	Description
drv_ak7755.h	AK7755 CODEC Driver Interface header file
drv_ak7755_config_template.h	AK7755 Codec Driver configuration template.

Description

This section lists the source and header files used by the AK7755Codec Driver Library.

drv_ak7755.h

AK7755 CODEC Driver Interface header file

Enumerations

	Name	Description
	DRV_AK7755_BICK_FS_FORMAT	This is type DRV_AK7755_BICK_FS_FORMAT.
	DRV_AK7755_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_AK7755_CHANNEL	Identifies left/right audio channel.
	DRV_AK7755_DAC_INPUT_FORMAT	Identifies the Serial Audio data interface format.
	DRV_AK7755_DSP_DIN1_INPUT_FORMAT	This is type DRV_AK7755_DSP_DIN1_INPUT_FORMAT.
	DRV_AK7755_DSP_DOUT1_OUTPUT_FORMAT	This is type DRV_AK7755_DSP_DOUT1_OUTPUT_FORMAT.
	DRV_AK7755_DSP_DOUT4_OUTPUT_FORMAT	This is type DRV_AK7755_DSP_DOUT4_OUTPUT_FORMAT.
	DRV_AK7755_DSP_PROGRAM	This is type DRV_AK7755_DSP_PROGRAM.
	DRV_AK7755_INT_EXT_MIC	Identifies the Mic input source.
	DRV_AK7755_LRCK_IF_FORMAT	This is type DRV_AK7755_LRCK_IF_FORMAT.
	DRV_AK7755_MONO_STEREO_MIC	Identifies the Mic input as Mono/Stereo.
	SAMPLE_LENGTH	in bits

Functions

	Name	Description
	DRV_AK7755_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_AK7755_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_AK7755_BufferAddWriteRead	This is function DRV_AK7755_BufferAddWriteRead.
	DRV_AK7755_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_AK7755_Close	Closes an opened-instance of the AK7755 Codec Driver.
	DRV_AK7755_CommandEventHandlerSet	Allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

	DRV_AK7755_Deinitialize	Deinitializes the specified instance of the AK7755 Codec Driver module.
	DRV_AK7755_Initialize	Initializes hardware and data for the instance of the AK7755 DAC module
	DRV_AK7755_IntExtMicSet	Sets up the codec for the internal or the external microphone use.
	DRV_AK7755_MonoStereoMicSet	Sets up the codec for the Mono or Stereo microphone mode.
	DRV_AK7755_MuteOff	Disables AK7755 output for soft mute.
	DRV_AK7755_MuteOn	Allows AK7755 output for soft mute on.
	DRV_AK7755_Open	Opens the specified AK7755 Codec Driver instance and returns a handle to it
	DRV_AK7755_SamplingRateGet	This function gets the sampling rate set on the AK7755. **Implementation:** Dynamic
	DRV_AK7755_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_AK7755_SetAudioCommunicationMode	This function provides a run time audio format configuration
	DRV_AK7755_Status	Gets the current status of the AK7755 Codec Driver module.
	DRV_AK7755_Tasks	Maintains the driver's control and data interface state machine.
	DRV_AK7755_VersionGet	Returns the version of the AK7755 Codec Driver.
	DRV_AK7755_VersionStrGet	This function returns the version of AK7755 Codec Driver in string format.
	DRV_AK7755_VolumeGet	Gets the volume for the AK7755 Codec Driver.
	DRV_AK7755_VolumeSet	This function sets the volume for AK7755 CODEC.

Macros

	Name	Description
	_DRV_AK7755_H	Include files.
	DRV_AK7755_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_AK7755_COUNT	Number of valid AK7755 Codec Driver indices
	DRV_AK7755_INDEX_0	AK7755 driver index definitions
	DRV_AK7755_INDEX_1	This is macro DRV_AK7755_INDEX_1.
	DRV_AK7755_INDEX_2	This is macro DRV_AK7755_INDEX_2.
	DRV_AK7755_INDEX_3	This is macro DRV_AK7755_INDEX_3.
	DRV_AK7755_INDEX_4	This is macro DRV_AK7755_INDEX_4.
	DRV_AK7755_INDEX_5	This is macro DRV_AK7755_INDEX_5.

Structures

	Name	Description
	DRV_AK7755_INIT	Defines the data required to initialize or reinitialize the AK7755 Codec Driver.

Types

	Name	Description
	DRV_AK7755_BUFFER_EVENT_HANDLER	Pointer to a AK7755 Driver Buffer Event handler function.
	DRV_AK7755_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_AK7755_COMMAND_EVENT_HANDLER	Pointer to a AK7755 Codec Driver command event handler function.

Description

AK7755 CODEC Driver Interface

The AK7755 CODEC device driver interface provides a simple interface to manage the AK7755 16/24-Bit Codec that can be interfaced Microchip Microcontroller. This file provides the interface definition for the AK7755 Codec device driver.

File Name

drv_ak7755.h

Company

Microchip Technology Inc.

drv_ak7755_config_template.h

AK7755 Codec Driver configuration template.

Macros

	Name	Description
	DRV_AK7755_BCLK_BIT_CLK_DIVISOR	Sets up the BCLK to LRCK ratio to generate the audio stream for the specified sampling frequency.
	DRV_AK7755_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_AK7755_INPUT_REFCLOCK	Identifies the input REFCLOCK source to generate the MCLK to the codec.
	DRV_AK7755_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_AK7755_MCLK_SAMPLE_FREQ_MULTPLIER	Sets up the MCLK to LRCK ratio to generate the audio stream for the specified sampling frequency.
	DRV_AK7755_MCLK_SOURCE	Indicates the input clock frequency to generate the MCLK to the codec.

Description

AK7755 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_ak7755_config_template.h

Company

Microchip Technology Inc.

WM8904 Codec Driver Library

This topic describes the WM8904 Codec Driver Library.

Introduction

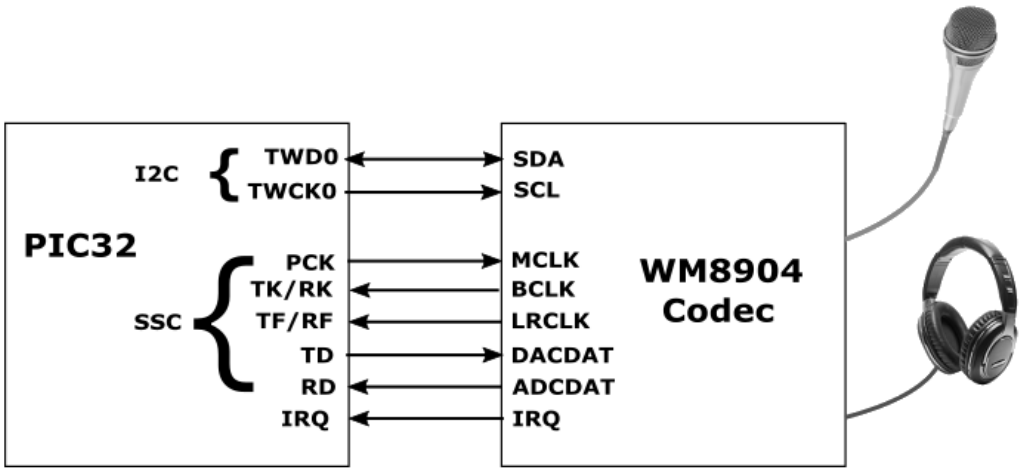
This library provides an Applications Programming Interface (API) to manage the WM8904 Codec that is serially interfaced to the I2C and I2S peripherals of a Microchip PIC32 microcontroller for the purpose of providing audio solutions.

Description

The WM8904 module is 24-bit Audio Codec from Cirrus Logic, which can operate in 16-, 20-, 24-, and 32-bit audio modes. The WM8904 can be interfaced to Microchip microcontrollers through I2C and I2S serial interfaces. The I2C interface is used to send commands and receive status, and the I2S interface is used for audio data output (to headphones or line-out) and input (from microphone or line-in).

The WM8904 can be configured as either an I2S clock slave (receives all clocks from the host), or I2S clock master (generates I2S clocks from a master clock input MCLK). Currently the driver only supports master mode with headphone output and (optionally) microphone input.

A typical interface of WM8904 to a Microchip PIC32 device using an I2C and SSC interface (configured as I2S), with the WM8904 set up as the I2S clock master, is provided in the following diagram:



Features

The WM8904 Codec supports the following features:

- Audio Interface Format: 16-/20-/24-/32-bit interface, LSB justified or I2S format

- Sampling Frequency Range: 8 kHz to 96 kHz
- Digital Volume Control: -71.625 to 0 dB in 192 steps
- Soft mute capability

Using the Library

This topic describes the basic architecture of the WM8904 Codec Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_WM8904.h`

The interface to the WM8904 Codec Driver library is defined in the `drv_WM8904.h` header file. Any C language source (`.c`) file that uses the WM8904 Codec Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

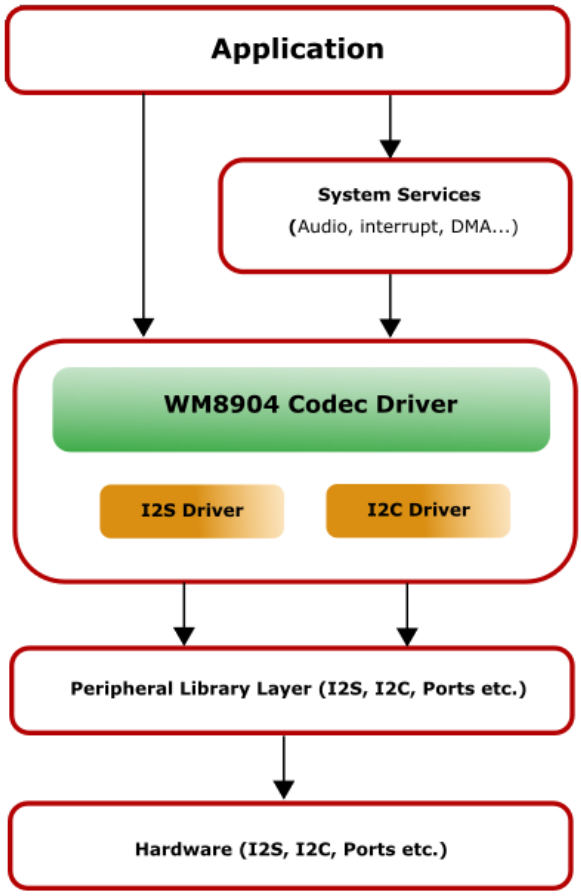
Abstraction Model

This library provides a low-level abstraction of the WM8904 Codec Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The abstraction model shown in the following diagram depicts how the WM8904 Codec Driver is positioned in the MPLAB Harmony framework. The WM8904 Codec Driver uses the I2C and I2S drivers for control and audio data transfers to the WM8904 module.

WM8904 Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The WM8904 Codec Driver Library provides an API interface to transfer control commands and digital audio data to the serially interfaced WM8904 Codec module. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the WM8904 Codec Driver Library.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open and close functions.
Data Transfer Functions	Provides data transfer functions, such as Buffer Read and Write.
Settings Functions	Provides driver specific functions for settings, such as volume control and sampling rate.
Other Functions	Miscellaneous functions, such as getting the driver's version number.
Data Types and Constants	These data types and constants are required while interacting and setting up the WM8904 Codec Driver Library.

Note: All functions and constants in this section are named with the format DRV_ WM8904_xxx, where 'xxx' is a function name or constant. These names are redefined in the appropriate configuration's `system_config.h` file to the format DRV_CODEC_xxx using `#defines` so that code in the application that references the library can be written as generically as possible (e.g., by writing DRV_CODEC_Open instead of DRV_ WM8904_Open etc.). This allows the codec type to be changed in the MHC without having to modify the application's source code.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

System Access

This topic describes system initialization, implementations, and includes a system access code example.

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization in the `system_init.c` file, each instance of the WM8904 module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_WM8904_INIT or by using Initialization Overrides) that are supported by the specific WM8904 device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- I2C driver module index. The module index should be same as the one used in initializing the I2C Driver
- I2S driver module index. The module index should be same as the one used in initializing the I2S Driver
- Sampling rate
- Volume
- Audio data format. The audio data format should match with the audio data format settings done in I2S driver initialization
- Determines whether or not the microphone input is enabled

The DRV_WM8904_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_ WM8904_Deinitialize, DRV_ WM8904_Status and DRV_I2S_Tasks.

Implementations

The WM8904 Codec Driver can has the following implementation:

Description	MPLAB Harmony Components
Dedicated hardware for control (I2C) and data (I2S) interface.	Standard MPLAB Harmony drivers for I2C and I2S interfaces.

Example:

```
SYS_STATUS status;
status = DRV_CODEC_Status(sysObjdrvCodec0);     // see if codec is done initializing
if (SYS_STATUS_READY == status)
{
    // The driver can now be opened.
   codecData->codecClient.handle = DRV_CODEC_Open
        (DRV_CODEC_INDEX_0, DRV_IO_INTENT_WRITE | DRV_IO_INTENT_EXCLUSIVE);
```

```
    if(appData.wm8904Client.handle != DRV_HANDLE_INVALID)
    {
        appData.state = APP_STATE_WM8904_SET_BUFFER_HANDLER;
    }
    else
    {
        SYS_DEBUG(0, "Find out what's wrong \r\n");
    }
}
else
{
    /* driver is not ready */
}
```

Task Routine

The DRV_WM8904_Tasks will be called from the System Task Service.

Client Access

This topic describes driver initialization and provides a code example.

Description

For the application to start using an instance of the module, it must call the DRV_WM8904_Open function. The DRV_WM8904_Open function provides a driver handle to the WM8904 Codec Driver instance for operations. If the driver is deinitialized using the function DRV_WM8904_Deinitialize, the application must call the DRV_WM8904_Open function again to set up the instance of the driver.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

Note: It is necessary to check the status of driver initialization before opening a driver instance. The status of the WM8904 Codec Driver can be known by calling DRV_ WM8904_Status.

Example:

```
DRV_HANDLE handle;
SYS_STATUS wm8904Status;
wm8904Status Status = DRV_WM8904_Status(sysObjects.wm8904Status DevObject);
if (SYS_STATUS_READY == wm8904Status)
{
// The driver can now be opened.
appData.wm8904Client.handle = DRV_WM8904_Open
(DRV_WM8904_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if(appData.wm8904Client.handle != DRV_HANDLE_INVALID)
{
appData.state = APP_STATE_WM8904_SET_BUFFER_HANDLER;
}
else
{
SYS_DEBUG(0, "Find out what's wrong \r\n");
}
}
else
{
/* WM8904 Driver Is not ready */
}
```

Client Operations

This topic provides information on client operations and includes a control command and audio buffered data operation flow diagram.

Description

Client operations provide the API interface for control command and audio data transfer to the WM8904 Codec.

The following WM8904 Codec specific control command functions are provided:

- DRV_WM8904_SamplingRateSet
- DRV_WM8904_SamplingRateGet
- DRV_WM8904_VolumeSet
- DRV_WM8904_VolumeGet

- DRV_WM8904_MuteOn
- DRV_WM8904_MuteOff

These functions schedule a non-blocking control command transfer operation. These functions submit the control command request to the WM8904 Codec. These functions submit the control command request to I2C Driver transmit queue, the request is processed immediately if it is the first request, or processed when the previous request is complete.

DRV_WM8904_BufferAddWrite, DRV_WM8904_BufferAddRead, and DRV_WM8904_BufferAddWriteRead are buffered data operation functions.

These functions schedule non-blocking audio data transfer operations. These functions add the request to I2S Driver transmit or receive buffer queue depends on the request type, and are executed immediately if it is the first buffer, or executed later when the previous buffer is complete. The driver notifies the client with DRV_WM8904_BUFFER_EVENT_COMPLETE, DRV_WM8904_BUFFER_EVENT_ERROR, or DRV_WM8904_BUFFER_EVENT_ABORT events.

Note: It is not necessary to close and reopen the client between multiple transfers.

Configuring the Library

Enumerations

	Name	Description
	DRV_WM8904_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.

Macros

	Name	Description
	_DRV_WM8904_CONFIG_TEMPLATE_H	This is macro _DRV_WM8904_CONFIG_TEMPLATE_H.
	DRV_CODEC_WM8904_MODE	Specifies if codec is in Master or Slave mode.
	DRV_WM8904_BAUD_RATE	Specifies the initial baud rate for the codec.
	DRV_WM8904_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_WM8904_ENABLE_MIC_INPUT	Specifies whether to enable the microphone input.
	DRV_WM8904_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_WM8904_VOLUME	Specifies the initial volume level.

Description

The configuration of the WM8904 Codec Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the WM8904 Codec Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the WM8904 Codec Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

_DRV_WM8904_CONFIG_TEMPLATE_H Macro

File

drv_wm8904_config_template.h

C

```
#define _DRV_WM8904_CONFIG_TEMPLATE_H
```

Description

This is macro _DRV_WM8904_CONFIG_TEMPLATE_H.

DRV_CODEC_WM8904_MODE Macro

Specifies if codec is in Master or Slave mode.

File

drv_wm8904_config_template.h

C

```
#define DRV_CODEC_WM8904_MODE
```

Description

WM8904 Codec Master/Slave Mode

Indicates whether the codec is to be operating in a Master mode (generating word and bit clock as outputs) or Slave mode receiving word and bit clock as inputs).

Remarks

Only Master mode is supported at this time.

DRV_WM8904_AUDIO_DATA_FORMAT Enumeration

Identifies the Serial Audio data interface format.

File

drv_wm8904.h

C

```
typedef enum {
  DATA_16_BIT_LEFT_JUSTIFIED,
  DATA_16_BIT_I2S,
  DATA_32_BIT_LEFT_JUSTIFIED,
  DATA_32_BIT_I2S
} DRV_WM8904_AUDIO_DATA_FORMAT;
```

Description

WM8904 Audio data format

This enumeration identifies Serial Audio data interface format.

DRV_WM8904_BAUD_RATE Macro

Specifies the initial baud rate for the codec.

File

drv_wm8904_config_template.h

C

```
#define DRV_WM8904_BAUD_RATE
```

Description

WM8904 Baud Rate

Sets the initial baud rate (sampling rate) for the codec. Typical values are 8000, 16000, 44100, 48000, 88200 and 96000.

Remarks

None.

DRV_WM8904_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_wm8904_config_template.h

C

```
#define DRV_WM8904_CLIENTS_NUMBER
```

Description

WM8904 Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. Typically only one client could be connected to one hardware instance. This value represents the total number of clients to be supported across all hardware instances.

Remarks

None.

DRV_WM8904_ENABLE_MIC_INPUT Macro

Specifies whether to enable the microphone input.

File

drv_wm8904_config_template.h

C

```
#define DRV_WM8904_ENABLE_MIC_INPUT
```

Description

WM8904 Microphone Enable

Indicates whether the ADC inputs for the two microphone channels (L-R) should be enabled.

Remarks

None.

DRV_WM8904_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_wm8904_config_template.h

C

```
#define DRV_WM8904_INSTANCES_NUMBER
```

Description

WM8904 driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of WM8904 Codec modules that are needed by an application, namely one.

Remarks

None.

DRV_WM8904_VOLUME Macro

Specifies the initial volume level.

File

drv_wm8904_config_template.h

C

```
#define DRV_WM8904_VOLUME
```

Description

WM8904 Volume

Sets the initial volume level, in the range 0-255.

Remarks

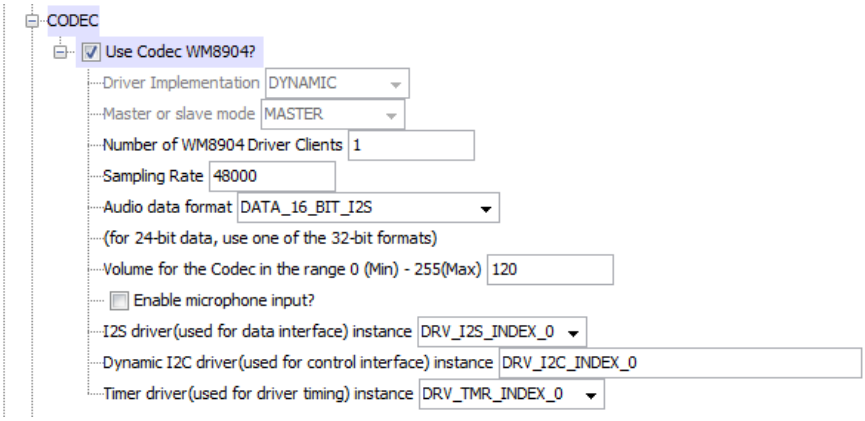
The value is mapped to an internal WM8904 volume level in the range 0-192 using a logarithmic table so the input scale appears linear (128 is half volume).

Configuring the MHC

Provides examples on how to configure the MPLAB Harmony Configurator (MHC) for a specific driver.

Description

The following figure shows an example of an MHC configuration for the WM8904 Codec Driver.



Building the Library

This section lists the files that are available in the WM8904 Codec Driver Library.

Description

This section list the files that are available in the `/src` folder of the WM8904 Codec Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/codec/wm8904`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_wm8904.h`	Header file that exports the driver API.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_wm8904.c`	This file contains implementation of the WM8904 Codec Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The WM8904 Codec Driver Library depends on the following modules:

- I2S Driver Library
- I2C Driver Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_WM8904_Initialize	Initializes hardware and data for the instance of the WM8904 DAC module
	DRV_WM8904_Deinitialize	Deinitializes the specified instance of the WM8904 driver module
	DRV_WM8904_Status	Gets the current status of the WM8904 driver module.
	DRV_WM8904_Tasks	Maintains the driver's control and data interface state machine.

b) Client Setup Functions

	Name	Description
	DRV_WM8904_Open	Opens the specified WM8904 driver instance and returns a handle to it
	DRV_WM8904_Close	Closes an opened-instance of the WM8904 driver
	DRV_WM8904_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_WM8904_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

c) Data Transfer Functions

	Name	Description
	DRV_WM8904_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_WM8904_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_WM8904_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic

d) Settings Functions

	Name	Description
	DRV_WM8904_MuteOff	This function disables WM8904 output for soft mute.
	DRV_WM8904_MuteOn	This function allows WM8904 output for soft mute on.
	DRV_WM8904_SamplingRateGet	This function gets the sampling rate set on the WM8904. **Implementation:** Dynamic
	DRV_WM8904_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_WM8904_SetAudioCommunicationMode	This function provides a run time audio format configuration
	DRV_WM8904_VolumeGet	This function gets the volume for WM8904 Codec.
	DRV_WM8904_VolumeSet	This function sets the volume for WM8904 Codec.

e) Other Functions

	Name	Description
	DRV_WM8904_VersionGet	This function returns the version of WM8904 driver
	DRV_WM8904_VersionStrGet	This function returns the version of WM8904 driver in string format.

f) Data Types and Constants

	Name	Description
	_DRV_WM8904_H	Include files.
	DRV_WM8904_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_WM8904_COUNT	Number of valid WM8904 driver indices
	DRV_WM8904_INDEX_0	WM8904 driver index definitions
	DRV_WM8904_INDEX_1	This is macro DRV_WM8904_INDEX_1.
	DRV_WM8904_INDEX_2	This is macro DRV_WM8904_INDEX_2.
	DRV_WM8904_INDEX_3	This is macro DRV_WM8904_INDEX_3.
	DRV_WM8904_INDEX_4	This is macro DRV_WM8904_INDEX_4.
	DRV_WM8904_INDEX_5	This is macro DRV_WM8904_INDEX_5.
	DRV_WM8904_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_WM8904_BUFFER_EVENT_HANDLER	Pointer to a WM8904 Driver Buffer Event handler function
	DRV_WM8904_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_WM8904_CHANNEL	Identifies Left/Right Audio channel
	DRV_WM8904_COMMAND_EVENT_HANDLER	Pointer to a WM8904 Driver Command Event Handler Function
	DRV_WM8904_INIT	Defines the data required to initialize or reinitialize the WM8904 driver

Description

This section describes the API functions of the WM8904 Codec Driver library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_WM8904_Initialize Function

Initializes hardware and data for the instance of the WM8904 DAC module

File

drv_wm8904.h

C

```
SYS_MODULE_OBJ DRV_WM8904_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the WM8904 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized.

Remarks

This routine must be called before any other WM8904 routine is called.

This routine should only be called once during system initialization unless DRV_WM8904_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

DRV_I2S_Initialize must be called before calling this function to initialize the data interface of this Codec driver. DRV_I2C_Initialize must be called if SPI driver is used for handling the control interface of this Codec driver.

Example

```
DRV_WM8904_INIT                 init;
SYS_MODULE_OBJ                  objectHandle;

init->inUse                             = true;
init->status                            = SYS_STATUS_BUSY;
init->numClients                        = 0;
init->i2sDriverModuleIndex              = wm8904Init->i2sDriverModuleIndex;
init->i2cDriverModuleIndex              = wm8904Init->i2cDriverModuleIndex;
init->samplingRate                      = DRV_WM8904_AUDIO_SAMPLING_RATE;
init->audioDataFormat                   = DRV_WM8904_AUDIO_DATA_FORMAT_MACRO;

init->isInInterruptContext              = false;

init->commandCompleteCallback = (DRV_WM8904_COMMAND_EVENT_HANDLER)0;
init->commandContextData = 0;
init->mclk_multiplier = DRV_WM8904_MCLK_SAMPLE_FREQ_MULTPLIER;


objectHandle = DRV_WM8904_Initialize(DRV_WM8904_0, (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized
init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

SYS_MODULE_OBJ DRV_WM8904_Initialize

(

const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT *const init

);

DRV_WM8904_Deinitialize Function

Deinitializes the specified instance of the WM8904 driver module

File

drv_wm8904.h

C

```
void DRV_WM8904_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the WM8904 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_WM8904_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_WM8904_Initialize
SYS_STATUS          status;


DRV_WM8904_Deinitialize(object);

status = DRV_WM8904_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_WM8904_Initialize routine

Function

void DRV_WM8904_Deinitialize(SYS_MODULE_OBJ object)

DRV_WM8904_Status Function

Gets the current status of the WM8904 driver module.

File

drv_wm8904.h

C

```
SYS_STATUS DRV_WM8904_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This routine provides the current status of the WM8904 driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_WM8904_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_WM8904_Initialize
SYS_STATUS          WM8904Status;

WM8904Status = DRV_WM8904_Status(object);
if (SYS_STATUS_READY == WM8904Status)
{
    // This means the driver can be opened using the
    // DRV_WM8904_Open() function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_WM8904_Initialize routine

Function

SYS_STATUS DRV_WM8904_Status(SYS_MODULE_OBJ object)

DRV_WM8904_Tasks Function

Maintains the driver's control and data interface state machine.

File

drv_wm8904.h

C

```
void DRV_WM8904_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal control and data interface state machine and implement its control and data interface implementations. This function should be called from the SYS_Tasks() function.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_WM8904_Initialize

while (true)
{
    DRV_WM8904_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_WM8904_Initialize)

Function

void DRV_WM8904_Tasks(SYS_MODULE_OBJ object);

b) Client Setup Functions

DRV_WM8904_Open Function

Opens the specified WM8904 driver instance and returns a handle to it

File

drv_wm8904.h

C

```
DRV_HANDLE DRV_WM8904_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_WM8904_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the ioIntent options passed are not relevant to this driver.

Description

This routine opens the specified WM8904 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options are not relevant to this driver. All the data transfer functions of this driver are non blocking.

WM8904 can be opened with DRV_IO_INTENT_WRITE, or DRV_IO_INTENT_READ or DRV_IO_INTENT_WRITEREAD io_intent option. This decides whether the driver is used for headphone output, or microphone input or both modes simultaneously.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_WM8904_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

Function DRV_WM8904_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_WM8904_Open(DRV_WM8904_INDEX_0, DRV_IO_INTENT_WRITEREAD | DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened

ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

DRV_HANDLE DRV_WM8904_Open
(
const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT ioIntent
)

DRV_WM8904_Close Function

Closes an opened-instance of the WM8904 driver

File

drv_wm8904.h

C

```
void DRV_WM8904_Close(const DRV_HANDLE handle);
```

Returns

- None

Description

This routine closes an opened-instance of the WM8904 driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_WM8904_Open before the caller may use the driver again

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_WM8904_Open

DRV_WM8904_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_WM8904_Close(DRV_Handle handle)

DRV_WM8904_BufferEventHandlerSet Function

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

File

drv_wm8904.h

C

```
void DRV_WM8904_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_WM8904_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls DRV_WM8904_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_WM8904_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

// Client registers an event handler with driver

DRV_WM8904_BufferEventHandlerSet(myWM8904Handle,
                APP_WM8904BufferEventHandler, (uintptr_t)&myAppObj);

DRV_WM8904_BufferAddWrite(myWM8904handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_WM8904_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_WM8904BufferEventHandler(DRV_WM8904_BUFFER_EVENT event,
        DRV_WM8904_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_WM8904_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_WM8904_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.

context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_WM8904_BufferEventHandlerSet
(
DRV_HANDLE handle,
const DRV_WM8904_BUFFER_EVENT_HANDLER eventHandler,
const uintptr_t contextHandle
)

DRV_WM8904_CommandEventHandlerSet Function

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

File

drv_wm8904.h

C

```
void DRV_WM8904_CommandEventHandlerSet(DRV_HANDLE handle, const DRV_WM8904_COMMAND_EVENT_HANDLER
eventHandler, const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.

The event handler should be set before the client performs any "WM8904 Codec Specific Client Routines" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the command has completed, it does not need to register a callback.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_WM8904_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

// Client registers an event handler with driver

DRV_WM8904_CommandEventHandlerSet(myWM8904Handle,
                APP_WM8904CommandEventHandler, (uintptr_t)&myAppObj);

DRV_WM8904_DeEmphasisFilterSet(myWM8904Handle, DRV_WM8904_DEEMPHASIS_FILTER_44_1KHZ)

// Event is received when
// the buffer is processed.

void APP_WM8904CommandEventHandler(uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.
```

```
    switch(event)
    {
            // Last Submitted command is completed.
            // Perform further processing here
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_WM8904_CommandEventHandlerSet

(

DRV_HANDLE handle,

const DRV_WM8904_COMMAND_EVENT_HANDLER eventHandler,

const uintptr_t contextHandle

)

c) Data Transfer Functions

DRV_WM8904_BufferAddRead Function

Schedule a non-blocking driver read operation.

File

drv_wm8904.h

C

```
void DRV_WM8904_BufferAddRead(const DRV_HANDLE handle, DRV_WM8904_BUFFER_HANDLE * bufferHandle, void *
buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_WM8904_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_WM8904_BUFFER_HANDLE_INVALID

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_WM8904_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_WM8904_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the WM8904 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another WM8904 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 device instance and the DRV_WM8904_Status must have returned SYS_STATUS_READY.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ must have been specified in the DRV_WM8904_Open call.

Parameters

Parameters	Description
handle	Handle of the WM8904 instance as return by the DRV_WM8904_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

```
void DRV_WM8904_BufferAddRead
(
const       DRV_HANDLE handle,
      DRV_WM8904_BUFFER_HANDLE *bufferHandle,
void *buffer, size_t size
)
```

DRV_WM8904_BufferAddWrite Function

Schedule a non-blocking driver write operation.

File

drv_wm8904.h

C

```
void DRV_WM8904_BufferAddWrite(const DRV_HANDLE handle, DRV_WM8904_BUFFER_HANDLE * bufferHandle, void *
buffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_WM8904_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_WM8904_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0.
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_WM8904_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_WM8904_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the WM8904 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another WM8904 driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 device instance and the DRV_WM8904_Status must have returned SYS_STATUS_READY.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE must have been specified in the DRV_WM8904_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_WM8904_BUFFER_HANDLE bufferHandle;
```

```
// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

// Client registers an event handler with driver

DRV_WM8904_BufferEventHandlerSet(myWM8904Handle,
                APP_WM8904BufferEventHandler, (uintptr_t)&myAppObj);

DRV_WM8904_BufferAddWrite(myWM8904handle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_WM8904_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_WM8904BufferEventHandler(DRV_WM8904_BUFFER_EVENT event,
        DRV_WM8904_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_WM8904_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_WM8904_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the WM8904 instance as return by the DRV_WM8904_Open function.
buffer	Data to be transmitted.
size	Buffer size in bytes.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

Function

void DRV_WM8904_BufferAddWrite
(
const DRV_HANDLE handle,
DRV_WM8904_BUFFER_HANDLE *bufferHandle,
void *buffer, size_t size
)

DRV_WM8904_BufferAddWriteRead Function

Schedule a non-blocking driver write-read operation.

Implementation: Dynamic

File

drv_wm8904.h

C

```
void DRV_WM8904_BufferAddWriteRead(const DRV_HANDLE handle, DRV_WM8904_BUFFER_HANDLE * bufferHandle, void *
transmitBuffer, void * receiveBuffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_WM8904_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_WM8904_BUFFER_EVENT_COMPLETE:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_WM8904_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_WM8904_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the WM8904 Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another WM8904 driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every WM8904 write. The transmit and receive size must be same.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 device instance and the DRV_WM8904_Status must have returned SYS_STATUS_READY.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_WM8904_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_WM8904_BUFFER_HANDLE bufferHandle;

// mywm8904Handle is the handle returned
// by the DRV_WM8904_Open function.

// Client registers an event handler with driver

DRV_WM8904_BufferEventHandlerSet(mywm8904Handle,
                APP_WM8904BufferEventHandler, (uintptr_t)&myAppObj);

DRV_WM8904_BufferAddWriteRead(mywm8904handle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_WM8904_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_WM8904BufferEventHandler(DRV_WM8904_BUFFER_EVENT event,
        DRV_WM8904_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
```

```
    {
        case DRV_WM8904_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_WM8904_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the WM8904 instance as returned by the DRV_WM8904_Open function
bufferHandle	Pointer to an argument that will contain the return buffer handle
transmitBuffer	The buffer where the transmit data will be stored
receiveBuffer	The buffer where the received data will be stored
size	Buffer size in bytes

Function

```
void DRV_WM8904_BufferAddWriteRead
(
const        DRV_HANDLE handle,
        DRV_WM8904_BUFFER_HANDLE *bufferHandle,
void *transmitBuffer,
void *receiveBuffer,
size_t size
)
```

d) Settings Functions

DRV_WM8904_MuteOff Function

This function disables WM8904 output for soft mute.

File

drv_wm8904.h

C

```
void DRV_WM8904_MuteOff(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables WM8904 output for soft mute.

Remarks

None.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

    DRV_WM8904_MuteOff(myWM8904Handle); //WM8904 output soft mute disabled
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_WM8904_MuteOff(DRV_HANDLE handle)

DRV_WM8904_MuteOn Function

This function allows WM8904 output for soft mute on.

File

drv_wm8904.h

C

```
void DRV_WM8904_MuteOn(DRV_HANDLE handle);
```

Returns

None.

Description

This function Enables WM8904 output for soft mute.

Remarks

None.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

DRV_WM8904_MuteOn(myWM8904Handle);  //WM8904 output soft muted
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_WM8904_MuteOn(DRV_HANDLE handle);

DRV_WM8904_SamplingRateGet Function

This function gets the sampling rate set on the WM8904.

Implementation: Dynamic

File

drv_wm8904.h

C

```
uint32_t DRV_WM8904_SamplingRateGet(DRV_HANDLE handle);
```

Description

This function gets the sampling rate set on the DAC WM8904.

Remarks

None.

Example

```
uint32_t baudRate;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

baudRate = DRV_WM8904_SamplingRateGet(myWM8904Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_WM8904_SamplingRateGet(DRV_HANDLE handle)

DRV_WM8904_SamplingRateSet Function

This function sets the sampling rate of the media stream.

File

drv_wm8904.h

C

```
void DRV_WM8904_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate);
```

Returns

None.

Description

This function sets the media sampling rate for the client handle.

Remarks

None.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Example

```
// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

DRV_WM8904_SamplingRateSet(myWM8904Handle, 48000);  //Sets 48000 media sampling rate
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
samplingRate	Sampling frequency in Hz

Function

void DRV_WM8904_SamplingRateSet(DRV_HANDLE handle, uint32_t samplingRate)

DRV_WM8904_SetAudioCommunicationMode Function

This function provides a run time audio format configuration

File

drv_wm8904.h

C

```
void DRV_WM8904_SetAudioCommunicationMode(DRV_HANDLE handle, const DATA_LENGTH dl, const SAMPLE_LENGTH sl);
```

Returns

None

Description

This function sets up audio mode in I2S protocol

Remarks

None.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
dl	Data length for I2S audio interface
sl	Left/Right Sample Length for I2S audio interface

Function

void DRV_WM8904_SetAudioCommunicationMode
(
DRV_HANDLE handle,
const DATA_LENGTH dl,
const SAMPLE_LENGTH sl
)

DRV_WM8904_VolumeGet Function

This function gets the volume for WM8904 Codec.

File

drv_wm8904.h

C

```
uint8_t DRV_WM8904_VolumeGet(DRV_HANDLE handle, DRV_WM8904_CHANNEL channel);
```

Returns

None.

Description

This functions gets the current volume programmed to the Codec WM8904.

Remarks

None.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
uint8_t volume;

// myWM8904Handle is the handle returned
// by the DRV_WM8904_Open function.

  volume = DRV_WM8904_VolumeGet(myWM8904Handle, DRV_WM8904_CHANNEL_LEFT);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
channel	argument indicating Left or Right or Both channel volume to be modified

Function

uint8_t DRV_WM8904_VolumeGet(DRV_HANDLE handle, DRV_WM8904_CHANNEL channel)

DRV_WM8904_VolumeSet Function

This function sets the volume for WM8904 Codec.

File

drv_wm8904.h

C

```
void DRV_WM8904_VolumeSet(DRV_HANDLE handle, DRV_WM8904_CHANNEL channel, uint8_t volume);
```

Returns

None

Description

This functions sets the volume value from 0-255. The codec has DAC value to volume range mapping as :- 00 H : +12dB FF H : -115dB In order to make the volume value to dB mapping monotonically increasing from 00 to FF, re-mapping is introduced which reverses the volume value to dB mapping as well as normalizes the volume range to a more audible dB range. The current driver implementation assumes that all dB values under -60 dB are inaudible to the human ear. Re-Mapped values 00 H : -60 dB FF H : +12 dB

Remarks

None.

Preconditions

The DRV_WM8904_Initialize routine must have been called for the specified WM8904 driver instance.

DRV_WM8904_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_BUFFER_HANDLE bufferHandle;

// myWM8904Handle is the handle returned
```

```
// by the DRV_WM8904_Open function.

    DRV_WM8904_VolumeSet(myWM8904Handle,DRV_WM8904_CHANNEL_LEFT, 120);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
channel	argument indicating Left or Right or Both channel volume to be modified
volume	volume value specified in the range 0-255 (0x00 to 0xFF)

Function

void DRV_WM8904_VolumeSet(DRV_HANDLE handle, DRV_WM8904_CHANNEL channel, uint8_t volume);

e) Other Functions

DRV_WM8904_VersionGet Function

This function returns the version of WM8904 driver

File

drv_wm8904.h

C

```
uint32_t DRV_WM8904_VersionGet();
```

Returns

returns the version of WM8904 driver.

Description

The version number returned from the DRV_WM8904_VersionGet function is an unsigned integer in the following decimal format. * 10000 + * 100 + Where the numbers are represented in decimal and the meaning is the same as above. Note that there is no numerical representation of release type.

Remarks

None.

Preconditions

None.

Example 1

For version "0.03a", return: 0 * 10000 + 3 * 100 + 0 For version "1.00", return: 1 * 100000 + 0 * 100 + 0

Example 2

```
uint32_t WM8904version;
WM8904version = DRV_WM8904_VersionGet();
```

Function

uint32_t DRV_WM8904_VersionGet(void)

DRV_WM8904_VersionStrGet Function

This function returns the version of WM8904 driver in string format.

File

drv_wm8904.h

C

```
int8_t* DRV_WM8904_VersionStrGet();
```

Returns

returns a string containing the version of WM8904 driver.

Description

The DRV_WM8904_VersionStrGet function returns a string in the format: ".[.][]" Where: is the WM8904 driver's version number. is the WM8904 driver's version number. is an optional "patch" or "dot" release number (which is not included in the string if it equals "00"). is an optional release type ("a" for alpha, "b" for beta ? not the entire word spelled out) that is not included if the release is a production version (I.e. Not an alpha or beta).

The String does not contain any spaces. For example, "0.03a" "1.00"

Remarks

None

Preconditions

None.

Example

```
int8_t *WM8904string;
WM8904string = DRV_WM8904_VersionStrGet();
```

Function

int8_t* DRV_WM8904_VersionStrGet(void)

f) Data Types and Constants

_DRV_WM8904_H Macro

File

drv_wm8904.h

C

```
#define _DRV_WM8904_H
```

Description

Include files.

DRV_WM8904_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_wm8904.h

C

```
#define DRV_WM8904_BUFFER_HANDLE_INVALID ((DRV_WM8904_BUFFER_HANDLE)(-1))
```

Description

WM8904 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_WM8904_BufferAddWrite() and the DRV_WM8904_BufferAddRead() function if the buffer add request was not successful.

Remarks

None.

DRV_WM8904_COUNT Macro

Number of valid WM8904 driver indices

File

drv_wm8904.h

C

```
#define DRV_WM8904_COUNT
```

Description

WM8904 Driver Module Count

This constant identifies the maximum number of WM8904 Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of WM8904 instances on this microcontroller.

Remarks

This value is part-specific.

DRV_WM8904_INDEX_0 Macro

WM8904 driver index definitions

File

drv_wm8904.h

C

```
#define DRV_WM8904_INDEX_0 0
```

Description

Driver WM8904 Module Index

These constants provide WM8904 driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_WM8904_Initialize and DRV_WM8904_Open routines to identify the driver instance in use.

DRV_WM8904_INDEX_1 Macro

File

drv_wm8904.h

C

```
#define DRV_WM8904_INDEX_1 1
```

Description

This is macro DRV_WM8904_INDEX_1.

DRV_WM8904_INDEX_2 Macro

File

drv_wm8904.h

C

```
#define DRV_WM8904_INDEX_2 2
```

Description

This is macro DRV_WM8904_INDEX_2.

DRV_WM8904_INDEX_3 Macro

File

drv_wm8904.h

C

```
#define DRV_WM8904_INDEX_3 3
```

Description

This is macro DRV_WM8904_INDEX_3.

DRV_WM8904_INDEX_4 Macro

File

drv_wm8904.h

C

```
#define DRV_WM8904_INDEX_4 4
```

Description

This is macro DRV_WM8904_INDEX_4.

DRV_WM8904_INDEX_5 Macro

File

drv_wm8904.h

C

```
#define DRV_WM8904_INDEX_5 5
```

Description

This is macro DRV_WM8904_INDEX_5.

DRV_WM8904_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_wm8904.h

C

```
typedef enum {
  DRV_WM8904_BUFFER_EVENT_COMPLETE,
  DRV_WM8904_BUFFER_EVENT_ERROR,
  DRV_WM8904_BUFFER_EVENT_ABORT
} DRV_WM8904_BUFFER_EVENT;
```

Members

Members	Description
DRV_WM8904_BUFFER_EVENT_COMPLETE	Data was transferred successfully.
DRV_WM8904_BUFFER_EVENT_ERROR	Error while processing the request
DRV_WM8904_BUFFER_EVENT_ABORT	Data transfer aborted (Applicable in DMA mode)

Description

WM8904 Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_WM8904_BufferAddWrite() or the DRV_WM8904_BufferAddRead() function.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_WM8904_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_WM8904_BUFFER_EVENT_HANDLER Type

Pointer to a WM8904 Driver Buffer Event handler function

File

drv_wm8904.h

C

```
typedef void (* DRV_WM8904_BUFFER_EVENT_HANDLER)(DRV_WM8904_BUFFER_EVENT event, DRV_WM8904_BUFFER_HANDLE
```

```
bufferHandle, uintptr_t contextHandle);
```

Returns

None.

Description

WM8904 Driver Buffer Event Handler Function

This data type defines the required function signature for the WM8904 driver buffer event handling callback function. A client must register a pointer to a buffer event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_WM8904_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_WM8904_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_WM8904_BufferProcessedSizeGet() function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_WM8904_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the data driver(i2S) peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

DRV_WM8904_BufferAddWrite function can be called in the event handler to add a buffer to the driver queue.

Example

```
void APP_MyBufferEventHandler( DRV_WM8904_BUFFER_EVENT event,
                               DRV_WM8904_BUFFER_HANDLE bufferHandle,
                               uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_WM8904_BUFFER_EVENT_COMPLETE:
            // Handle the completed buffer.
        break;

        case DRV_WM8904_BUFFER_EVENT_ERROR:
        default:
            // Handle error.
        break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_WM8904_BUFFER_HANDLE Type

Handle identifying a write buffer passed to the driver.

File

drv_wm8904.h

C

```
typedef uintptr_t DRV_WM8904_BUFFER_HANDLE;
```

Description

WM8904 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_WM8904_BufferAddWrite() or DRV_WM8904_BufferAddRead() function. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer.

The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

DRV_WM8904_CHANNEL Enumeration

Identifies Left/Right Audio channel

File

drv_wm8904.h

C

```
typedef enum {
  DRV_WM8904_CHANNEL_LEFT,
  DRV_WM8904_CHANNEL_RIGHT,
  DRV_WM8904_CHANNEL_LEFT_RIGHT,
  DRV_WM8904_NUMBER_OF_CHANNELS
} DRV_WM8904_CHANNEL;
```

Description

WM8904 Audio Channel

This enumeration identifies Left/Right Audio channel

Remarks

None.

DRV_WM8904_COMMAND_EVENT_HANDLER Type

Pointer to a WM8904 Driver Command Event Handler Function

File

drv_wm8904.h

C

```
typedef void (* DRV_WM8904_COMMAND_EVENT_HANDLER)(uintptr_t contextHandle);
```

Returns

None.

Description

WM8904 Driver Command Event Handler Function

This data type defines the required function signature for the WM8904 driver command event handling callback function.

A command is a control instruction to the WM8904 Codec. Example Mute ON/OFF, Zero Detect Enable/Disable etc.

A client must register a pointer to a command event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive command related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

The occurrence of this call back means that the last control command was transferred successfully.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_WM8904_CommandEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The event handler function executes in the control data driver interrupt context. It is recommended of the application to not perform process intensive or blocking operations with in this function.

Example

```
void APP_WM8904CommandEventHandler( uintptr_t context )
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    // Last Submitted command is completed.
    // Perform further processing here
}
```

Parameters

Parameters	Description
context	Value identifying the context of the application that registered the event handling function.

DRV_WM8904_INIT Structure

Defines the data required to initialize or reinitialize the WM8904 driver

File

drv_wm8904.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX i2sDriverModuleIndex;
  SYS_MODULE_INDEX i2cDriverModuleIndex;
  uint32_t samplingRate;
  uint8_t volume;
  DRV_WM8904_AUDIO_DATA_FORMAT audioDataFormat;
  bool enableMicInput;
} DRV_WM8904_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX i2sDriverModuleIndex;	Identifies data module(I2S) driver ID for data interface of Codec
SYS_MODULE_INDEX i2cDriverModuleIndex;	Identifies data module(I2C) driver ID for control interface of Codec
uint32_t samplingRate;	Sampling rate
uint8_t volume;	Volume
DRV_WM8904_AUDIO_DATA_FORMAT audioDataFormat;	Identifies the Audio data format
bool enableMicInput;	true if mic input path enabled

Description

WM8904 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the WM8904 Codec driver.

Remarks

None.

Files

Files

Name	Description
drv_wm8904_config_template.h	WM8904 Codec Driver Configuration Template.
drv_wm8904.h	WM8904 Codec Driver Interface header file

Description

This section lists the source and header files used by the WM8904Codec Driver Library.

drv_wm8904_config_template.h

WM8904 Codec Driver Configuration Template.

Macros

	Name	Description
	_DRV_WM8904_CONFIG_TEMPLATE_H	This is macro _DRV_WM8904_CONFIG_TEMPLATE_H.
	DRV_CODEC_WM8904_MODE	Specifies if codec is in Master or Slave mode.
	DRV_WM8904_BAUD_RATE	Specifies the initial baud rate for the codec.
	DRV_WM8904_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_WM8904_ENABLE_MIC_INPUT	Specifies whether to enable the microphone input.
	DRV_WM8904_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_WM8904_VOLUME	Specifies the initial volume level.

Description

WM8904 Codec Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_wm8904_config_template.h

Company

Microchip Technology Inc.

drv_wm8904.h

WM8904 Codec Driver Interface header file

Enumerations

	Name	Description
	DATA_LENGTH	in bits
	DRV_WM8904_AUDIO_DATA_FORMAT	Identifies the Serial Audio data interface format.
	DRV_WM8904_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_WM8904_CHANNEL	Identifies Left/Right Audio channel

Functions

	Name	Description
	DRV_WM8904_BufferAddRead	Schedule a non-blocking driver read operation.
	DRV_WM8904_BufferAddWrite	Schedule a non-blocking driver write operation.
	DRV_WM8904_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_WM8904_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_WM8904_Close	Closes an opened-instance of the WM8904 driver
	DRV_WM8904_CommandEventHandlerSet	This function allows a client to identify a command event handling function for the driver to call back when the last submitted command have finished.
	DRV_WM8904_Deinitialize	Deinitializes the specified instance of the WM8904 driver module
	DRV_WM8904_Initialize	Initializes hardware and data for the instance of the WM8904 DAC module
	DRV_WM8904_MuteOff	This function disables WM8904 output for soft mute.
	DRV_WM8904_MuteOn	This function allows WM8904 output for soft mute on.
	DRV_WM8904_Open	Opens the specified WM8904 driver instance and returns a handle to it
	DRV_WM8904_SamplingRateGet	This function gets the sampling rate set on the WM8904. **Implementation:** Dynamic
	DRV_WM8904_SamplingRateSet	This function sets the sampling rate of the media stream.
	DRV_WM8904_SetAudioCommunicationMode	This function provides a run time audio format configuration
	DRV_WM8904_Status	Gets the current status of the WM8904 driver module.

	DRV_WM8904_Tasks	Maintains the driver's control and data interface state machine.
	DRV_WM8904_VersionGet	This function returns the version of WM8904 driver
	DRV_WM8904_VersionStrGet	This function returns the version of WM8904 driver in string format.
	DRV_WM8904_VolumeGet	This function gets the volume for WM8904 Codec.
	DRV_WM8904_VolumeSet	This function sets the volume for WM8904 Codec.

Macros

	Name	Description
	_DRV_WM8904_H	Include files.
	DRV_I2C_INDEX	This is macro DRV_I2C_INDEX.
	DRV_WM8904_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_WM8904_COUNT	Number of valid WM8904 driver indices
	DRV_WM8904_INDEX_0	WM8904 driver index definitions
	DRV_WM8904_INDEX_1	This is macro DRV_WM8904_INDEX_1.
	DRV_WM8904_INDEX_2	This is macro DRV_WM8904_INDEX_2.
	DRV_WM8904_INDEX_3	This is macro DRV_WM8904_INDEX_3.
	DRV_WM8904_INDEX_4	This is macro DRV_WM8904_INDEX_4.
	DRV_WM8904_INDEX_5	This is macro DRV_WM8904_INDEX_5.

Structures

	Name	Description
	DRV_WM8904_INIT	Defines the data required to initialize or reinitialize the WM8904 driver

Types

	Name	Description
	DRV_WM8904_BUFFER_EVENT_HANDLER	Pointer to a WM8904 Driver Buffer Event handler function
	DRV_WM8904_BUFFER_HANDLE	Handle identifying a write buffer passed to the driver.
	DRV_WM8904_COMMAND_EVENT_HANDLER	Pointer to a WM8904 Driver Command Event Handler Function

Description

WM8904 Codec Driver Interface

The WM8904 Codec device driver interface provides a simple interface to manage the WM8904 16/24/32-Bit Codec that can be interfaced to a Microchip microcontroller. This file provides the public interface definitions for the WM8904 Codec device driver.

File Name

drv_wm8904.h

Company

Microchip Technology Inc.

Comparator Driver Library

This section describes the Comparator Driver Library.

Introduction

The Comparator Static Driver provides a high-level interface to manage the Comparator module on the Microchip family of microcontrollers.

Description

Through MHC, this driver provides an API to initialize the Comparator module, as well as reference channels, CVREF, inputs, and interrupts.

Library Interface

Function(s)

	Name	Description
	DRV_CMP_Initialize	Initializes the Comparator instance for the specified driver index. **Implementation:** Static

Description

This section describes the Application Programming Interface (API) functions of the Comparator Driver Library.

Function(s)

DRV_CMP_Initialize Function

Initializes the Comparator instance for the specified driver index.

Implementation: Static

File

help_drv_cmp.h

C

```
void DRV_CMP_Initialize();
```

Returns

None.

Description

This routine initializes the Comparator driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters. The driver instance index is independent of the Comparator module ID. For example, driver instance 0 can be assigned to Comparator 2.

Remarks

This routine must be called before any other Comparator routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Function

void DRV_CMP_Initialize(void)

CPLD XC2C64A Driver Library

This section describes the CPLD XC2C64A Driver Library.

Introduction

This library provides an interface to manage the CPLD XC2C64A devices on Microchip starter kits.

Description

A CPLD is provided on the Multimedia Expansion Board (MEB), which can be used to configure the graphics controller bus interface, SPI channel and Chip Selects used for SPI Flash, the MRF24WBOMA, and the expansion slot. The general I/O inputs are used to change the configuration, which can be done at run-time.

Specific CPLD configuration information is available in the *"Multimedia Expansion Board (MEB) User's Guide"* (DS60001160), which is available from the MEB product page: http://www.microchip.com/Developmenttools/ProductDetails.aspx?PartNO=DM320005

Using the Library

This topic describes the basic architecture of the CPLD XC2C64A Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_xc2c64a.h`

The interface to the CPLD XC2C64A Driver Library is defined in the `drv_xc2c64a.h` header file. Any C language source (`.c`) file that uses the CPLD XC2C64A Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the CPLD XC2C64A Driver.

Library Interface Section	Description
Functions	Provides CPLD XC2C64A initialization and configuration functions.

Configuring the Library

The configuration of the CPLD XC2C64A Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the CPLD XC2C64A Driver. Based on the selections made, the CPLD XC2C64A may support the selected features. These configuration settings will apply to all instances of the CPLD XC2C64A Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Building the Library

This section lists the files that are available in the CPLD XC2C64A Driver Library.

Description

This section list the files that are available in the `/src` folder of the CPLD XC2C64A Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/cpld/xc2c64a`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_xc2c64a.h`	Header file that exports the CPLD XC2C64A Driver API.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_xc2c64a.c`	Basic CPLD XC2C64A Driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The CPLD XC2C64A Driver Library is not dependent on other modules.

Library Interface

a) Functions

	Name	Description
	CPLDGetDeviceConfiguration	Returns the selected device. **Implementation:** Static

	CPLDGetGraphicsConfiguration	Returns the selected PMP bus, 8 or 16-bit, interface to the graphics controller. **Implementation:** Static
	CPLDGetSPIConfiguration	Returns the selected SPI Channel. **Implementation:** Static
	CPLDInitialize	Initializes the control I/O to the CPLD and places the CPLD in a known state. **Implementation:** Static
	CPLDSetGraphicsConfiguration	Selects the PMP bus, 8 or 16-bit, interface to the graphic controller. **Implementation:** Static
	CPLDSetSPIFlashConfiguration	Selects the SPI Flash device. **Implementation:** Static
	CPLDSetWiFiConfiguration	Selects the Wi-Fi device. **Implementation:** Static
	CPLDSetZigBeeConfiguration	Selects the ZigBee/MiWi device. **Implementation:** Static

b) Data Types and Constants

	Name	Description
	CPLD_DEVICE_CONFIGURATION	CPLD device configuration.
	CPLD_GFX_CONFIGURATION	CPLD graphics controller PMP bus configuration.
	CPLD_SPI_CONFIGURATION	CPLD SPI channel selection.

Description

This section describes the API functions of the CPLD XC2C64A Driver Library.

Refer to each section for a detailed description.

a) Functions

CPLDGetDeviceConfiguration Function

Returns the selected device.

Implementation: Static

File

drv_xc2c64a.h

C

```
CPLD_DEVICE_CONFIGURATION CPLDGetDeviceConfiguration();
```

Returns

- CPLD_DEVICE_SPI_FLASH - SPI Flash.
- CPLD_DEVICE_WiFi - Zero G 802.11 Wi-Fi.
- CPLD_DEVICE_ZIGBEE - ZigBee/MiWi.

Description

This routine returns the selected CPLD device.

Remarks

None.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

```
// Initialize the CPLD
CPLDInitialize();

if(CPLDGetDeviceConfiguration() != CPLD_DEVICE_SPI_FLASH)
{
    // error - not setup as default
```

```
}
```

Function

CPLD_DEVICE_CONFIGURATION CPLDGetDeviceConfiguration(void)

CPLDGetGraphicsConfiguration Function

Returns the selected PMP bus, 8 or 16-bit, interface to the graphics controller.

Implementation: Static

File

drv_xc2c64a.h

C

```
CPLD_GFX_CONFIGURATION CPLDGetGraphicsConfiguration();
```

Returns

- CPLD_GFX_CONFIG_8BIT - Graphics controller is configured for 8-bit PMP data bus interface.
- CPLD_GFX_CONFIG_16BIT - Graphics controller is configured for 16-bit PMP data bus interface.

Description

This routine gets the configuration of the PMP, 8 or 16-bit, data bus interface.

Remarks

None.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

```
// Initialize the CPLD
CPLDInitialize();

if(CPLDGetGraphicsConfiguration() != CPLD_GFX_CONFIG_8BIT)
{
    // error - not setup as default
}
```

Function

CPLD_GFX_CONFIGURATION CPLDGetGraphicsConfiguration(void)

CPLDGetSPIConfiguration Function

Returns the selected SPI Channel.

Implementation: Static

File

drv_xc2c64a.h

C

```
CPLD_SPI_CONFIGURATION CPLDGetSPIConfiguration();
```

Returns

- CPLD_SPI2A - SPI Channel 2A with chip select PORT G bit 9 and external interrupt 1 or 3
- CPLD_SPI3A - SPI Channel 3A with chip select PORT F bit 12 and change notice 9
- CPLD_SPI2 - SPI Channel 2 with chip select PORT G bit 9 and external interrupt 1 or 3

Description

This routine returns the selected SPI channel.

Remarks

SPI channels 2 and 2A are located on the same pins. SPI channels 2A and 3A are only available on PIC32MX5xx/6xx/7xx series parts.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

```
// Initialize the CPLD
CPLDInitialize();

if(CPLDGetSPIConfiguration() != CPLD_SPI2A)
{
    // error - not setup as default

}
```

Function

CPLD_SPI_CONFIGURATION CPLDGetSPIConfiguration(void)

CPLDInitialize Function

Initializes the control I/O to the CPLD and places the CPLD in a known state.

Implementation: Static

File

drv_xc2c64a.h

C

```
void CPLDInitialize();
```

Returns

None.

Description

This routine configures the control I/O and places the CPLD in a known state.

- Graphics Controller Bus - 8-bit PMP data interface.
- SPI Channel - SPI2/SPI2A.
- Chip Select - PORT G bit 9.
- External Interrupt 1 or 3
- Device - SPI Flash.

Remarks

None.

Preconditions

None.

Example

```
// Initialize the CPLD
CPLDInitialize();

// CPLD is configured in the default state
```

Function

void CPLDInitialize(void)

CPLDSetGraphicsConfiguration Function

Selects the PMP bus, 8 or 16-bit, interface to the graphic controller.

Implementation: Static

File

drv_xc2c64a.h

C

```
void CPLDSetGraphicsConfiguration(CPLD_GFX_CONFIGURATION configuration);
```

Returns

None.

Description

This routine sets the configuration pins on the graphics controller to select between an 8 or 16-bit data bus interface.

Remarks

The graphics controller interface configuration must be done before initializing the graphics controller.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

Setting the graphics controller to a 16-bit interface

```
// Initialize the CPLD
CPLDInitialize();

// configure the graphics controller for a 16-bit PMP interface.
CPLDSetGraphicsConfiguration(CPLD_GFX_CONFIG_16BIT);
```

Setting the graphics controller to a 8-bit interface

```
// Initialize the CPLD
CPLDInitialize();

// configure the graphics controller for a 8-bit PMP interface.
CPLDSetGraphicsConfiguration(CPLD_GFX_CONFIG_8BIT);
```

Parameters

Parameters	Description
configuration	the type of interface configuration.

Function

void CPLDSetGraphicsConfiguration(CPLD_GFX_CONFIGURATION configuration)

CPLDSetSPIFlashConfiguration Function

Selects the SPI Flash device.

Implementation: Static

File

drv_xc2c64a.h

C

```
void CPLDSetSPIFlashConfiguration(CPLD_SPI_CONFIGURATION configuration);
```

Returns

None.

Description

This routine configures the CPLD to communicate to the SPI Flash device with the selected SPI channel and Chip Select.

Remarks

SPI channels 2 and 2A are located on the same pins. SPI channels 2A and 3A are only available on PIC32MX5xx/6xx/7xx series parts.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

Setting CPLD to SPI Flash using SPI channel 2 and chip select PORT G bit 9

```
// Initialize the CPLD
CPLDInitialize();

// configure the SPI Flash to use SPI channel 2 and chip select PORT G bit 9
CPLDSetSPIFlashConfiguration(CPLD_SPI2);
```

Setting CPLD to SPI Flash using SPI channel 2A and chip select PORT G bit 9

```
// Initialize the CPLD
CPLDInitialize();

// configure the SPI Flash to use SPI channel 2A and chip select PORT G bit 9
CPLDSetSPIFlashConfiguration(CPLD_SPI2A);
```

Setting CPLD to SPI Flash using SPI channel 3A and chip select PORT F bit 12

```
// Initialize the CPLD
CPLDInitialize();

// configure the SPI Flash to use SPI channel 3A and chip select PORT F bit 12
CPLDSetSPIFlashConfiguration(CPLD_SPI3A);
```

Parameters

Parameters	Description
configuration	the type of SPI channel used by the SPI Flash device.

Function

void CPLDSetSPIFlashConfiguration(CPLD_SPI_CONFIGURATION configuration)

CPLDSetWiFiConfiguration Function

Selects the Wi-Fi device.

Implementation: Static

File

drv_xc2c64a.h

C

```
void CPLDSetWiFiConfiguration(CPLD_SPI_CONFIGURATION configuration);
```

Returns

None.

Description

This routine configures the CPLD to communicate to the Wi-Fi device with the selected SPI channel, chip select and external interrupt or change notice.

Remarks

SPI channels 2 and 2A are located on the same pins. SPI channels 2A and 3A are only available on PIC32MX5xx/6xx/7xx series parts.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

Setting CPLD to Wi-Fi using SPI channel 2, chip select PORT G bit 9 and external interrupt 3

```
// Initialize the CPLD
CPLDInitialize();

// configure the Wi-Fi to use SPI channel 2, chip select PORT G bit 9 and external interrupt 3
CPLDSetWiFiConfiguration(CPLD_SPI2);
```

Setting CPLD to Wi-Fi using SPI channel 2A, chip select PORT G bit 9 and external interrupt 3

```
// Initialize the CPLD
CPLDInitialize();

// configure the Wi-Fi to use SPI channel 2A, chip select PORT G bit 9 and external interrupt 3
CPLDSetWiFiConfiguration(CPLD_SPI2A);
```

Setting CPLD to Wi-Fi using SPI channel 3A, chip select PORT F bit 12 and change notice 9

```
// Initialize the CPLD
CPLDInitialize();

// configure the Wi-Fi to use SPI channel 3A, chip select PORT F bit 12 and change notice 9
CPLDSetWiFiConfiguration(CPLD_SPI3A);
```

Parameters

Parameters	Description
configuration	the type of SPI channel used by the Wi-Fi device.

Function

void CPLDSetWiFiConfiguration(CPLD_SPI_CONFIGURATION configuration)

CPLDSetZigBeeConfiguration Function

Selects the ZigBee/MiWi device.

Implementation: Static

File

drv_xc2c64a.h

C

```
void CPLDSetZigBeeConfiguration(CPLD_SPI_CONFIGURATION configuration);
```

Returns

None.

Description

This routine configures the CPLD to communicate to the ZigBee/MiWi device with the selected SPI channel, chip select and external interrupt or change notice.

Remarks

SPI channels 2 and 2A are located on the same pins. SPI channels 2A and 3A are only available on PIC32MX5xx/6xx/7xx series parts.

Preconditions

The initialization routine, CPLDInitialize, must be called.

Example

Setting CPLD to ZigBee/MiWi using SPI channel 2, chip select PORT G bit 9 and external interrupt 3

```
// Initialize the CPLD
CPLDInitialize();

// configure the ZigBee/MiWi to use SPI channel 2, chip select PORT G bit 9 and external interrupt 3
CPLDSetZigBeeConfiguration(CPLD_SPI2);
```

Setting CPLD to ZigBee/MiWi using SPI channel 2A, chip select PORT G bit 9 and external interrupt 3

```
// Initialize the CPLD
CPLDInitialize();

// configure the ZigBee/MiWi to use SPI channel 2A, chip select PORT G bit 9 and external interrupt 3
CPLDSetZigBeeConfiguration(CPLD_SPI2A);
```

Setting CPLD to ZigBee/MiWi using SPI channel 3A, chip select PORT F bit 12 and change notice 9

```
// Initialize the CPLD
CPLDInitialize();

// configure the ZigBee/MiWi to use SPI channel 3A, chip select PORT F bit 12 and change notice 9
CPLDSetZigBeeConfiguration(CPLD_SPI3A);
```

Parameters

Parameters	Description
configuration	the type of SPI channel used by the ZigBee/MiWi device.

Function

void CPLDSetZigBeeConfiguration(CPLD_SPI_CONFIGURATION configuration)

b) Data Types and Constants

CPLD_DEVICE_CONFIGURATION Enumeration

CPLD device configuration.

File

drv_xc2c64a.h

C

```
typedef enum {
  CPLD_DEVICE_SPI_FLASH,
  CPLD_DEVICE_WiFi,
  CPLD_DEVICE_ZIGBEE
} CPLD_DEVICE_CONFIGURATION;
```

Members

Members	Description
CPLD_DEVICE_SPI_FLASH	SPI Flash
CPLD_DEVICE_WiFi	Zero G Wi-Fi
CPLD_DEVICE_ZIGBEE	ZigBee/MiWi

Description

The CPLD can be configured to communicate to three different devices. The application may call routine, CPLDGetDeviceConfiguration, to obtain what device the CPLD is configured to communicate with.

Remarks

None.

Example

```
// select 16-bit PMP data bus
if(CPLDGetDeviceConfiguration() != CPLD_DEVICE_SPI_FLASH)
{
    // error - not default configuration
}
```

CPLD_GFX_CONFIGURATION Enumeration

CPLD graphics controller PMP bus configuration.

File

drv_xc2c64a.h

C

```
typedef enum {
  CPLD_GFX_CONFIG_8BIT,
  CPLD_GFX_CONFIG_16BIT
} CPLD_GFX_CONFIGURATION;
```

Members

Members	Description
CPLD_GFX_CONFIG_8BIT	Configure the Graphics Controller to use 8-bit PMP data bus

CPLD_GFX_CONFIG_16BIT	Configure the Graphics Controller to use 16-bit PMP data bus

Description

The application can select what PMP bus configuration, 8 or 16-bit data bus, when interfacing with the graphics controller.

Remarks

None.

Example

```
// select 16-bit PMP data bus
CPLDSetGraphicsConfiguration(CPLD_GFX_CONFIG_16BIT);
```

CPLD_SPI_CONFIGURATION Enumeration

CPLD SPI channel selection.

File

drv_xc2c64a.h

C

```
typedef enum {
  CPLD_SPI2A,
  CPLD_SPI3A,
  CPLD_SPI2
} CPLD_SPI_CONFIGURATION;
```

Members

Members	Description
CPLD_SPI2A	PIC32 SPI Channel 2A and chip select PORT G bit 9
CPLD_SPI3A	PIC32 SPI Channel 3A and chip select PORT F bit 12
CPLD_SPI2	PIC32 SPI Channel 2 and chip select PORT G bit 9

Description

The application can select what SPI channel will be used as the communication interface. It will also select the Chip Select use for the device.

Remarks

Only one SPI channel can be select for a device. SPI channels 2 and 2A are located on the same pins. SPI channels 2A and 3A are only available on PIC32MX5xx/6xx/7xx series devices.

Example

```
// select SPI channel two for SPI Flash
CPLDSetSPIFlashConfiguration(CPLD_SPI2);
```

Files

Files

Name	Description
drv_xc2c64a.h	This file contains the interface definition for the CUPLD controller.

Description

This section lists the source and header files used by the SPI Flash Driver Library.

drv_xc2c64a.h

This file contains the interface definition for the CUPLD controller.

Enumerations

	Name	Description
	CPLD_DEVICE_CONFIGURATION	CPLD device configuration.
	CPLD_GFX_CONFIGURATION	CPLD graphics controller PMP bus configuration.

	CPLD_SPI_CONFIGURATION	CPLD SPI channel selection.

Functions

	Name	Description
	CPLDGetDeviceConfiguration	Returns the selected device. **Implementation:** Static
	CPLDGetGraphicsConfiguration	Returns the selected PMP bus, 8 or 16-bit, interface to the graphics controller. **Implementation:** Static
	CPLDGetSPIConfiguration	Returns the selected SPI Channel. **Implementation:** Static
	CPLDInitialize	Initializes the control I/O to the CPLD and places the CPLD in a known state. **Implementation:** Static
	CPLDSetGraphicsConfiguration	Selects the PMP bus, 8 or 16-bit, interface to the graphic controller. **Implementation:** Static
	CPLDSetSPIFlashConfiguration	Selects the SPI Flash device. **Implementation:** Static
	CPLDSetWiFiConfiguration	Selects the Wi-Fi device. **Implementation:** Static
	CPLDSetZigBeeConfiguration	Selects the ZigBee/MiWi device. **Implementation:** Static

Description

CUPLD Controller Interface File.

This library provides a low-level abstraction of the CUPLD device. It can be used to simplify low-level access to the device without the necessity of interacting directly with the communication module's registers, thus hiding differences from one serial device variant to another.

File Name

drv_xc2c64a.h

Company

Microchip Technology Inc.

CTR Driver Library

This section describes the Cycle Time Register (CTR) Driver Library.

Introduction

This library provides a low-level abstraction of the Cycle Time Register (CTR) module on Microchip microcontrollers with a convenient C language interface. It can be used to simplify low-level access to the module without the necessity of interacting directly with the module's registers, thus hiding differences from one microcontroller variant to another.

Description

The CTR is a hardware block that can be used to track specific signals from subsystems to internally log corresponding system time. Subsystems can include network clock synchronization, Media Clock synchronization, USB start of frame (SoF), and so on.

Using the Library

This section describes the basic architecture of the CTR Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_ctr.h`

The interface to the CTR Module Library is defined in the `drv_ctr.h` header file. Any C language source (`.c`) file that uses the CTR Driver Library should include this header.

Refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the CTR Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The CTR driver provides an interface to perform a one-time configuration of the CTR peripheral. Initialization steps include selecting the mode of operation, interrupt and trigger sources, latch configurations, and so on.

In addition, the driver allows the client to register a callback that is executed when the desired event has been triggered.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the CTR module.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Other Functions	Provides driver miscellaneous functions, data transfer status function, version identification functions, and so on.
Data Types and Constants	Provides data types and macros.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

Note: Not all modes are available on all devices, please refer to the specific device data sheet to determine the modes that are supported for your device.

Configuring the Library

The configuration of the driver is based on the file `system_config.h`.

Description

The header file contains the configuration selection for the driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Building the Library

This section lists the files that are available in the CTR Driver Library.

Description

This section list the files that are available in the `/src` folder of the CTR Driver Library. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/ctr/`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_ctr.h`	Header file that exports the driver API.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_ctr.c	Basic CTR Driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The CTR Driver Library depends on the following modules:

- Clock System Service Library

Optional Dependencies

- DMA System Service Library (used when operating in DMA mode)
- Interrupt System Service Library (used when task is running in Interrupt mode)

Library Interface

This section describes the API functions of the CTR Driver Library.

Refer to each section for a detailed description.

a) System Interaction Functions

Functions

	Name	Description
	DRV_CTR_Deinitialize	Deinitializes the specified instance of the CTR driver module. **Implementation:** Dynamic
	DRV_CTR_Initialize	Initializes the CTR Driver instance for the specified driver index. **Implementation:** Dynamic
	DRV_CTR_Status	Gets the current status of the CTR Driver module. **Implementation:** Dynamic

Description

DRV_CTR_Deinitialize Function

Deinitializes the specified instance of the CTR driver module.

Implementation: Dynamic

File

drv_ctr.h

C

```
void DRV_CTR_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the CTR Driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This function will NEVER block waiting for hardware.

Preconditions

Function DRV_CTR_Initialize should have been called before calling this function.

Example

```
// This code snippet shows an example of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_CTR_Initialize
SYS_STATUS          status;


DRV_CTR_Deinitialize(object);

status = DRV_CTR_Status(object);
if (SYS_STATUS_UNINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_CTR_Initialize

Function

void DRV_CTR_Deinitialize(SYS_MODULE_OBJ object)

DRV_CTR_Initialize Function

Initializes the CTR Driver instance for the specified driver index.

Implementation: Dynamic

File

drv_ctr.h

C

```
SYS_MODULE_OBJ DRV_CTR_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes the CTR driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This function must be called before any other CTR function is called.

This function should only be called once during system initialization unless DRV_CTR_Deinitialize is called to deinitialize the driver instance.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the CTR Driver. All
the CTR initialization is done in #defines mentioned, and the init structure
is initialized with corresponding #defines and then passed to initialize
function.

// *****************************************************************************
// CTR Driver Configuration Options

#define DRV_CTR_POWER_STATE             SYS_MODULE_POWER_RUN_FULL
#define DRV_CTR_MODULE_ID               CTR_ID_0
#define DRV_CTR_CLIENTS_NUMBER          1
#define DRV_CTR_INSTANCES_NUMBER        1
#define DRV_CTR_EVENT_INTERRUPT_SOURCE  INT_SOURCE_CTR1_EVENT
#define DRV_CTR_EVENT_INTERRUPT_MODE    CTR_LATCH_TRIG
```

```
#define DRV_CTR_TRIGGER_INTERRUPT_SOURCE INT_SOURCE_CTR1_TRG
#define DRV_CTR_M_0                      0x000000
#define DRV_CTR_N_0                      0x000000
#define DRV_CTR_LSB_0                    0x00
#define DRV_CTR_MODE_0                   CTR_US
#define DRV_CTR_M_1                      0x000000
#define DRV_CTR_N_1                      0x000000
#define DRV_CTR_LSB_1                    0x00
#define DRV_CTR_MODE_1                   CTR_US
#define DRV_CTR_COUNTER_SEL              CTR_CTR0_LIN
#define DRV_CTR_DIVIDER                  0
#define DRIVER_MODE                      WIFI_MODE
#define DRV_CTR_LATCH0_TRIG              CTR_WIFI_TM_1
#define DRV_CTR_LATCH1_TRIG              CTR_WIFI_TM_2
#define DRV_CTR_LATCH2_TRIG              CTR_WIFI_TM_3
#define DRV_CTR_LATCH3_TRIG              CTR_WIFI_TM_4
#define DRV_CTR_TRIGGER_SOURCE               CTR_CTR0_LIN
#define DRV_CTR_TRIGGER_PHASE                0x000

DRV_CTR_INIT   CTRInitData;
SYS_MODULE_OBJ      objectHandle;

CTRInitData.moduleInit = DRV_CTR_POWER_STATE,
CTRInitData.ctrEventInterruptSource = DRV_CTR_EVENT_INTERRUPT_SOURCE,
CTRInitData.ctrLatchEventMode = DRV_CTR_EVENT_INTERRUPT_MODE,
CTRInitData.ctrTriggerInterruptSource = DRV_CTR_TRIGGER_INTERRUPT_SOURCE,
CTRInitData.ctrCounter[0].M = DRV_CTR_M_0,
CTRInitData.ctrCounter[0].N = DRV_CTR_N_0,
CTRInitData.ctrCounter[0].LSB = DRV_CTR_LSB_0,
CTRInitData.ctrCounter[1].M = DRV_CTR_M_1,
CTRInitData.ctrCounter[1].N = DRV_CTR_N_1,
CTRInitData.ctrCounter[1].LSB = DRV_CTR_LSB_1,
CTRInitData.ctrLatch[0].ctrSel = DRV_CTR_COUNTER_SEL,
CTRInitData.ctrLatch[1].ctrSel = DRV_CTR_COUNTER_SEL,
CTRInitData.ctrLatch[2].ctrSel = DRV_CTR_COUNTER_SEL,
CTRInitData.ctrLatch[3].ctrSel = DRV_CTR_COUNTER_SEL,
CTRInitData.ctrLatch[0].trigSel = DRV_CTR_LATCH0_TRIG,
CTRInitData.ctrLatch[1].trigSel = DRV_CTR_LATCH1_TRIG,
CTRInitData.ctrLatch[2].trigSel = DRV_CTR_LATCH2_TRIG,
CTRInitData.ctrLatch[3].trigSel = DRV_CTR_LATCH3_TRIG,
CTRInitData.ctrLatch[0].divider = DRV_CTR_DIVIDER,
CTRInitData.ctrLatch[1].divider = DRV_CTR_DIVIDER,
CTRInitData.ctrLatch[2].divider = DRV_CTR_DIVIDER,
CTRInitData.ctrLatch[3].divider = DRV_CTR_DIVIDER,
CTRInitData.ctrTrigger.trigSource = DRV_CTR_TRIGGER_SOURCE,
CTRInitData.ctrTrigger.phase = DRV_CTR_TRIGGER_PHASE,
CTRInitData.drvMode = DRIVER_MODE

objectHandle = DRV_CTR_Initialize(DRV_CTR_INDEX_0,
                                  (SYS_MODULE_INIT*)CTRInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to a data structure containing data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_CTR_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

);

DRV_CTR_Status Function

Gets the current status of the CTR Driver module.

Implementation: Dynamic

File

drv_ctr.h

C

```
SYS_STATUS DRV_CTR_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and accept requests for new operations

SYS_STATUS_UNINITIALIZED - Indicates that the driver is not initialized

Description

This function provides the current status of the CTR Driver module.

Remarks

A driver can only be opened when its status is SYS_STATUS_READY.

Preconditions

Function DRV_CTR_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_CTR_Initialize
SYS_STATUS          CTRStatus;

CTRStatus = DRV_CTR_Status(object);
if (SYS_STATUS_ERROR == CTRStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_CTR_Initialize

Function

SYS_STATUS DRV_CTR_Status(SYS_MODULE_OBJ object)

b) Other Functions

Functions

	Name	Description
	DRV_CTR_Adjust	Sets the adjust value for a given CTR counter. **Implementation:** Dynamic
	DRV_CTR_ClientStatus	Gets current client-specific status of the CTR driver. **Implementation:** Dynamic
	DRV_CTR_Close	Closes an opened-instance of the CTR driver. **Implementation:** Dynamic
	DRV_CTR_Drift	Sets the drift value for a given CTR counter. **Implementation:** Dynamic
	DRV_CTR_EventISR	Interrupt Service Routine called for the CTR event interrupt. **Implementation:** Dynamic
	DRV_CTR_Open	Opens the specified CTR driver instance and returns a handle to it. **Implementation:** Dynamic

	DRV_CTR_RegisterCallBack	Registers a callback function for the event interrupt of CTR. **Implementation:** Dynamic
	DRV_CTR_TriggerISR	Interrupt Service Routine called for the CTR Trigger interrupt. **Implementation:** Dynamic

Description

DRV_CTR_Adjust Function

Sets the adjust value for a given CTR counter.

Implementation: Dynamic

File

drv_ctr.h

C

```
void DRV_CTR_Adjust(DRV_HANDLE handle, CTR_LATCH_CTR_SELECT ctrSel, uint16_t adjustVal);
```

Returns

None.

Description

This function sets the adjust value for a given CTR counter.

Preconditions

The DRV_CTR_Initialize function must have been called.

DRV_CTR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle; // handle returned by open function
uint16_t adjustVal = 0xFFF;

DRV_CTR_Adjust(handle, CTR_CTR0_LIN, adjustVal);
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's open

Function

void DRV_CTR_Adjust(DRV_HANDLE handle, CTR_LATCH_CTR_SELECT ctrSel,

uint16_t adjustVal);

ctrSel - CTR counter to be selected out of the 4 counters available.

adjustVal - Adjust value to be set

DRV_CTR_ClientStatus Function

Gets current client-specific status of the CTR driver.

Implementation: Dynamic

File

drv_ctr.h

C

```
DRV_CTR_CLIENT_STATUS DRV_CTR_ClientStatus(const DRV_HANDLE handle);
```

Returns

A DRV_CTR_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the CTR driver associated with the given handle.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_CTR_Initialize function must have been called.

DRV_CTR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      handle;         // Returned from DRV_CTR_Open
DRV_CTR_CLIENT_STATUS     clientStatus;

clientStatus = DRV_CTR_ClientStatus(handle);
if(DRV_CTR_CLIENT_STATUS_READY == clientStatus)
{
    // do the tasks
}
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's open

Function

DRV_CTR_CLIENT_STATUS DRV_CTR_ClientStatus(DRV_HANDLE handle);

DRV_CTR_Close Function

Closes an opened-instance of the CTR driver.

Implementation: Dynamic

File

drv_ctr.h

C

```
void DRV_CTR_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened-instance of the CTR driver, invalidating the handle.

Remarks

After calling this function, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_CTR_Open before the caller may use the driver again.

Usually, there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_CTR_Initialize function must have been called for the specified CTR driver instance.

DRV_CTR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_CTR_Open

DRV_CTR_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

void DRV_CTR_Close(DRV_Handle handle);

DRV_CTR_Drift Function

Sets the drift value for a given CTR counter.

Implementation: Dynamic

File

drv_ctr.h

C

```
void DRV_CTR_Drift(DRV_HANDLE handle, CTR_LATCH_CTR_SELECT ctrSel, uint32_t driftVal);
```

Returns

None.

Description

This function sets the drift value for a given CTR counter.

Preconditions

The DRV_CTR_Initialize function must have been called.

DRV_CTR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle; // handle returned by open function
uint16_t driftVal = 0xFFF;

DRV_CTR_Drift(handle, CTR_CTR0_LIN, driftVal);
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's open

Function

void DRV_CTR_Drift(DRV_HANDLE handle, CTR_LATCH_CTR_SELECT ctrSel,

uint16_t driftVal);

ctrSel - CTR counter to be selected out of the 4 counters available.

adjustVal - Drift value to be set

DRV_CTR_EventISR Function

Interrupt Service Routine called for the CTR event interrupt.

Implementation: Dynamic

File

drv_ctr.h

C

```
void DRV_CTR_EventISR(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is called when the interrupt is generated for CTR event interrupt. The latch buffers are read and stored in a local buffer, and all the registered client callback functions will be called from this function. The number of latches to be read depends upon the use-case configured. For wifi, 4 latches are read, and for USBSoF and GPIO, only 1 latch is read. Number of buffers to read in each latch depends on the interrupt mode configuration. For Full, all 4 buffers needs to be read, whereas for half-full, only 2 buffers needs to be read and for every trigger, only 1 buffer is

read.

Remarks

All the handling specific for a client should be done in the respective callback functions. This function should not be modified.

Preconditions

None.

Example

This function is not called from clients/system. This function will be called when the interrupt for event is generated.

Parameters

Parameters	Description
object	The driver instance handle returned after the initialization.

Function

void DRV_CTR_EventISR(SYS_MODULE_OBJ object);

DRV_CTR_Open Function

Opens the specified CTR driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_ctr.h

C

```
DRV_HANDLE DRV_CTR_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the function returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_CTR_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver status is not ready.

Description

This function opens the specified CTR driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The driver will always work in Non-Blocking mode even if IO-intent is selected as blocking.

The handle returned is valid until the DRV_CTR_Close function is called.

This function will NEVER block waiting for hardware.

Preconditions

Function DRV_CTR_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_CTR_Open(DRV_CTR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_CTR_Open
(
const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT ioIntent
);

DRV_CTR_RegisterCallBack Function

Registers a callback function for the event interrupt of CTR.

Implementation: Dynamic

File

drv_ctr.h

C

```
void DRV_CTR_RegisterCallBack(const DRV_HANDLE handle, const DRV_CTR_CALLBACK callback, const bool
oneTimeCallback, const uintptr_t context);
```

Returns

None.

Description

This function registers a client callback function for the event interrupt associated with the use-case. For Wifi usecase, Only Latch 3 interrupt will be enabled, as the last event timestamp will be filled in latch 3 for IEEE 802.11v. For USBSoF and GPIO use-cases, only one latch is needed and the interrupt will be enabled for the same latch. As per user's configuration of interrupt mode for full, half-full or every trigger, the interrupt will be generated and the client callback functions will be called from the ISR. The flag oneTimeCallback is passed as an argument for this function. If the value of this flag is TRUE, then the callback will be called only once. If client needs one more callback, he needs to register the callback once more. If this value is false, then whenever interrupt is generated, the callback function will be called until the client call the close function.

Remarks

The registered callback function will be called from ISR. So, it is recommended to keep the callback functions light and not process intensive.

Preconditions

The DRV_CTR_Initialize function must have been called.

DRV_CTR_Open must have been called to obtain a valid opened device handle.

Example

```
#define CLIENT_ID 0x01
DRV_HANDLE      handle;         // Returned from DRV_CTR_Open
void ClientCallack( uintptr_t context, uint32_t * timestampbuffer,
                    uint8_t BufferSize);

DRV_CTR_RegisterCallBack(handle, ClientCallack, FALSE, CLIENT_ID);
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's open

Function

void DRV_CTR_RegisterCallBack(
const DRV_HANDLE handle,
const DRV_CTR_CALLBACK callback,
const bool oneTimeCallback,

const uintptr_t context
);

callback - A function pointer for client callback function
oneTimeCallback - If client needs callback to be called only once, then
this flag must be true.
context - The value of parameter will be passed back to the client
unchanged, when the callback function is called. It can
be used to identify any client specific data object that
identifies the instance of the client module (for example,
it may be a pointer to the client module's state structure).

DRV_CTR_TriggerISR Function

Interrupt Service Routine called for the CTR Trigger interrupt.

Implementation: Dynamic

File

drv_ctr.h

C

```
void DRV_CTR_TriggerISR(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is called when the interrupt is generated for CTR trigger interrupt. The interrupt handling for this interrupt is application specific. So, this function is kept open for the clients to modify.

Remarks

Specific interrupt handling can be taken care of by application developer, as the need for this interrupt is application specific.

Preconditions

None.

Example

This function is not called from clients/system. This function will be called when the interrupt for event is generated.

Parameters

Parameters	Description
object	The driver instance handle returned after the initialization.

Function

void DRV_CTR_TriggerISR(SYS_MODULE_OBJ object);

c) Data Types and Constants

Enumerations

	Name	Description
	DRV_CTR_CLIENT_STATUS	Defines the client status. **Implementation:** Dynamic
	DRV_MODE	Defines the driver mode. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_CTR_COUNTER_NUM	Number of counters in CTR module

	DRV_CTR_INDEX_0	CTR driver index definitions
	DRV_CTR_LATCH_FIFO_CNT	FIFO size for each latch in CTR module
	DRV_CTR_LATCH_NUM	Number of latches in CTR module

Structures

	Name	Description
	DRV_CTR_COUNTER	Contains all the data necessary to initialize the CTR counter. **Implementation:** Dynamic
	DRV_CTR_INIT	Contains all the data necessary to initialize the CTR. **Implementation:** Dynamic
	DRV_CTR_LATCH	Contains all the data necessary to initialize the CTR Latches. **Implementation:** Dynamic
	DRV_CTR_TRIGGER	Contains all the data necessary to initialize the CTR Triggers. **Implementation:** Dynamic

Types

	Name	Description
	DRV_CTR_CALLBACK	Callback function definition for CTR event interrupt. **Implementation:** Dynamic

Description

DRV_CTR_CALLBACK Type

Callback function definition for CTR event interrupt.

Implementation: Dynamic

File

drv_ctr.h

C

```
typedef void (* DRV_CTR_CALLBACK)(uintptr_t context, uint32_t * timestampbuffer, uint8_t BufferSize);
```

Description

CTR Event interrupt callback function

The clients must define their callback functions in the same prototype as DRV_CTR_CALLBACK. All the registered callbacks will be called from drive ISR for CTR event.

Remarks

This structure is a part of initialization structure, which is used to initialize the CTR module.

DRV_CTR_CLIENT_STATUS Enumeration

Defines the client status.

Implementation: Dynamic

File

drv_ctr.h

C

```
typedef enum {
  DRV_CTR_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0,
  DRV_CTR_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY,
  DRV_CTR_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED,
  DRV_CTR_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR
} DRV_CTR_CLIENT_STATUS;
```

Members

Members	Description
DRV_CTR_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0	Up and running, ready to start new operations
DRV_CTR_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY	Operation in progress, unable to start a new one
DRV_CTR_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED	Client is closed
DRV_CTR_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR	Client Error

Description

CTR Client Status

Defines the various client status codes.

Remarks

None.

DRV_CTR_COUNTER Structure

Contains all the data necessary to initialize the CTR counter.

Implementation: Dynamic

File

drv_ctr.h

C

```
typedef struct {
  uint32_t M;
  uint32_t N;
  uint8_t LSB;
  CTR_MODE_SELECT Mode;
} DRV_CTR_COUNTER;
```

Description

CTR Counter init structure

This structure contains all of the data necessary to initialize the CTR counter increment steps and the resolution.

Remarks

This structure is a part of initialization structure, which is used to initialize the CTR module.

DRV_CTR_INIT Structure

Contains all the data necessary to initialize the CTR.

Implementation: Dynamic

File

drv_ctr.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  CTR_MODULE_ID ctrId;
  INT_SOURCE ctrEventInterruptSource;
  CTR_LATCH_INT_MODE ctrLatchEventMode;
  INT_SOURCE ctrTriggerInterruptSource;
  DRV_CTR_COUNTER ctrCounter[DRV_CTR_COUNTER_NUM];
  DRV_CTR_LATCH ctrLatch[DRV_CTR_LATCH_NUM];
  DRV_CTR_TRIGGER ctrTrigger;
  DRV_MODE drvMode;
} DRV_CTR_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
CTR_MODULE_ID ctrId;	Identifies the CTR peripheral instance
INT_SOURCE ctrEventInterruptSource;	CTR Event Interrupt Source
CTR_LATCH_INT_MODE ctrLatchEventMode;	CTR Event Interrupt Mode
INT_SOURCE ctrTriggerInterruptSource;	CTR Triggetr Interrupt Source
DRV_CTR_COUNTER ctrCounter[DRV_CTR_COUNTER_NUM];	Counter Init Data
DRV_CTR_LATCH ctrLatch[DRV_CTR_LATCH_NUM];	Latch Init Data
DRV_MODE drvMode;	Driver Mode

Description

CTR Driver Initialization Data

This structure contains all of the data necessary to initialize the CTR.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_CTR_Initialize function.

DRV_CTR_LATCH Structure

Contains all the data necessary to initialize the CTR Latches.

Implementation: Dynamic

File

drv_ctr.h

C

```
typedef struct {
  CTR_LATCH_TRIGGER_SELECT trigSel;
  CTR_LATCH_CTR_SELECT ctrSel;
  uint8_t divider;
} DRV_CTR_LATCH;
```

Description

CTR Latch init structure

This structure contains all of the data necessary to initialize the CTR Latches for mapping the trigger source and counter for a given latch.

Remarks

This structure is a part of initialization structure, which is used to initialize the CTR module.

DRV_CTR_TRIGGER Structure

Contains all the data necessary to initialize the CTR Triggers.

Implementation: Dynamic

File

drv_ctr.h

C

```
typedef struct {
  CTR_LATCH_CTR_SELECT trigSource;
  uint16_t phase;
} DRV_CTR_TRIGGER;
```

Description

CTR Trigger init structure

This structure contains all of the data necessary to initialize the CTR Triggers for generating triggers from CTR.

Remarks

This structure is a part of initialization structure, which is used to initialize the CTR module.

DRV_MODE Enumeration

Defines the driver mode.

Implementation: Dynamic

File

drv_ctr.h

C

```
typedef enum {
  WIFI_MODE = 0,
  USB_MODE,
  GPIO_MODE
} DRV_MODE;
```

Description

CTR Driver mode

Driver can be configured to use for either of Wifi, USB or GPIO.

Remarks

None.

DRV_CTR_COUNTER_NUM Macro

Number of counters in CTR module

File

drv_ctr.h

C

```
#define DRV_CTR_COUNTER_NUM 2
```

Description

Counters present in the CTR module

These constants provide Number of counters in CTR module.

Remarks

These constants should be used in place of hard-coded numeric literals.

DRV_CTR_INDEX_0 Macro

CTR driver index definitions

File

drv_ctr.h

C

```
#define DRV_CTR_INDEX_0 0
```

Description

Driver CTR Module Index reference

These constants provide CTR driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_CTR_Initialize and DRV_CTR_Open routines to identify the driver instance in use.

DRV_CTR_LATCH_FIFO_CNT Macro

FIFO size for each latch in CTR module

File

drv_ctr.h

C

```
#define DRV_CTR_LATCH_FIFO_CNT 4
```

Description

FIFO size for each latch in the CTR module

These constants provide Number of FIFO location available in each latch in CTR module.

Remarks

These constants should be used in place of hard-coded numeric literals.

DRV_CTR_LATCH_NUM Macro

Number of latches in CTR module

File

drv_ctr.h

C

```
#define DRV_CTR_LATCH_NUM 6
```

Description

Latches present in the CTR module

These constants provide Number of latches in CTR module.

Remarks

These constants should be used in place of hard-coded numeric literals.

Files

Files

Name	Description
drv_ctr.h	CTR Driver Interface Definition

Description

This section lists the source and header files used by the CTR Driver Library.

drv_ctr.h

CTR Driver Interface Definition

Enumerations

	Name	Description
	DRV_CTR_CLIENT_STATUS	Defines the client status. **Implementation:** Dynamic
	DRV_MODE	Defines the driver mode. **Implementation:** Dynamic

Functions

	Name	Description
	DRV_CTR_Adjust	Sets the adjust value for a given CTR counter. **Implementation:** Dynamic

	DRV_CTR_ClientStatus	Gets current client-specific status of the CTR driver. **Implementation:** Dynamic
	DRV_CTR_Close	Closes an opened-instance of the CTR driver. **Implementation:** Dynamic
	DRV_CTR_Deinitialize	Deinitializes the specified instance of the CTR driver module. **Implementation:** Dynamic
	DRV_CTR_Drift	Sets the drift value for a given CTR counter. **Implementation:** Dynamic
	DRV_CTR_EventISR	Interrupt Service Routine called for the CTR event interrupt. **Implementation:** Dynamic
	DRV_CTR_Initialize	Initializes the CTR Driver instance for the specified driver index. **Implementation:** Dynamic
	DRV_CTR_Open	Opens the specified CTR driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_CTR_RegisterCallBack	Registers a callback function for the event interrupt of CTR. **Implementation:** Dynamic
	DRV_CTR_Status	Gets the current status of the CTR Driver module. **Implementation:** Dynamic
	DRV_CTR_TriggerISR	Interrupt Service Routine called for the CTR Trigger interrupt. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_CTR_COUNTER_NUM	Number of counters in CTR module
	DRV_CTR_INDEX_0	CTR driver index definitions
	DRV_CTR_LATCH_FIFO_CNT	FIFO size for each latch in CTR module
	DRV_CTR_LATCH_NUM	Number of latches in CTR module

Structures

	Name	Description
	DRV_CTR_COUNTER	Contains all the data necessary to initialize the CTR counter. **Implementation:** Dynamic
	DRV_CTR_INIT	Contains all the data necessary to initialize the CTR. **Implementation:** Dynamic
	DRV_CTR_LATCH	Contains all the data necessary to initialize the CTR Latches. **Implementation:** Dynamic
	DRV_CTR_TRIGGER	Contains all the data necessary to initialize the CTR Triggers. **Implementation:** Dynamic

Types

	Name	Description
	DRV_CTR_CALLBACK	Callback function definition for CTR event interrupt. **Implementation:** Dynamic

Description

CTR Driver Interface Definition

The CTR device driver provides a simple interface to manage the CTR Module This file defines the interface definition for the CTR Driver.

File Name

drv_CTR.h

Company

Microchip Technology Inc.

Data EEPROM Driver Library

This section describes the Data EEPROM Driver Library.

Introduction

The MPLAB Harmony Data EEPROM Driver provides a high-level interface to manage the Data EEPROM module on the Microchip family of microcontrollers.

Description

The Data EEPROM Driver provides the following features:

- Application-ready routines to perform block operations on the Data EEPROM
- Multi-client operation support
- Data transfer events
- Supports Non-blocking mode of operation only

The Data EEPROM Driver supports multi-client operation, which allows multiple application clients to access the same memory device. Multiple instances of the driver can be used when multiple EEPROM devices are required to be part of the system.

Using the Library

This topic describes the basic architecture of the Data EEPROM Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_eeprom.h`

The interface to the EEPROM Driver Library is defined in the `drv_eeprom.h` header file. Any C language source (`.c`) file that uses the Data EPROM Driver Library should include `drv_eeprom.h`.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

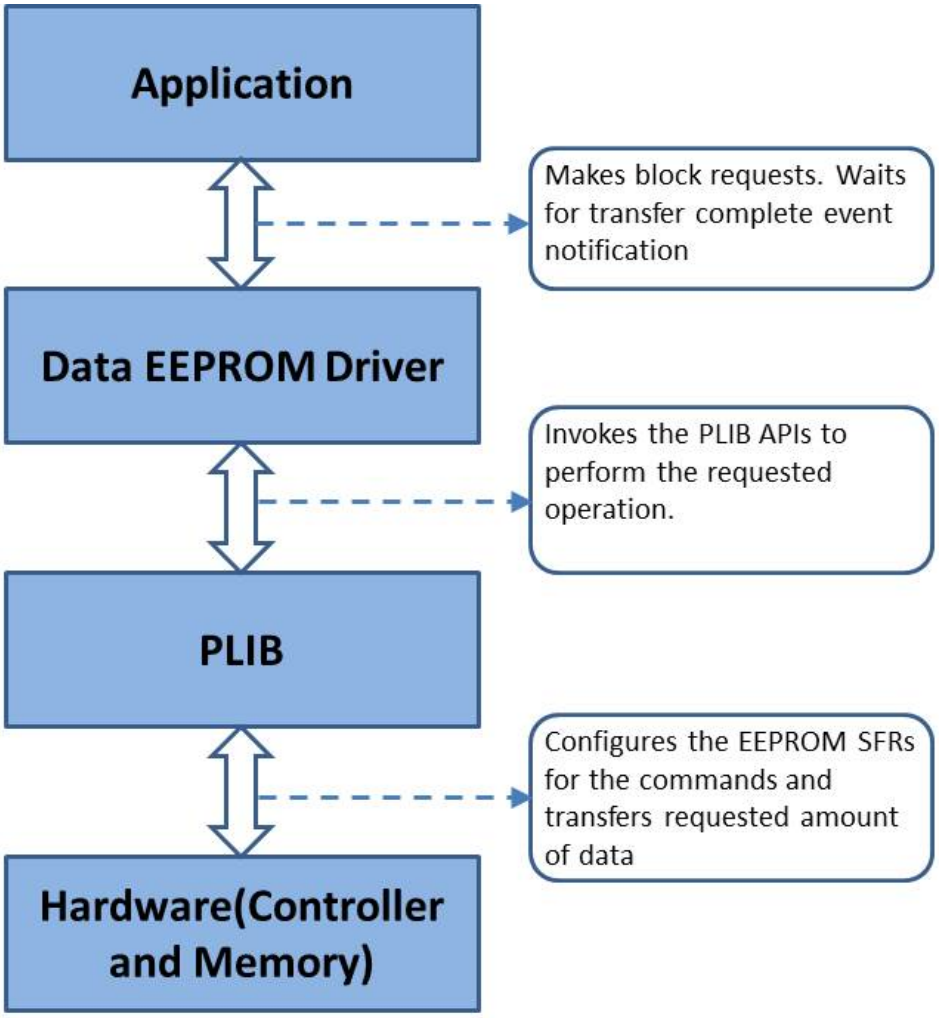
Abstraction Model

This library provides a low-level abstraction of the Data EEPROM Driver Library on the Microchip family of microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The Data EEPROM driver provides a set of APIs that can be used to perform Erase, Write, and Read operations. The following diagram depicts the communication between different modules. As shown in the diagram, the Data EEPROM Driver sits between the Peripheral Libraries and the application or system layer to facilitate block and file access to the EEPROM.

Data EEPROM Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Data EEPROM Driver.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, tasks, and status functions.
Client Core Functions	Provides open, close, and other setup functions.
Block Operation Functions	Provides read, write, and erase functions to perform data transfer operations on the EEPROM device.
Media Interface Functions	Provides functions to query the EEPROM geometry and media status.

How the Library Works

Provides information on system, client core, block operation, and media interface functions.

Description

System Functions

Data EEPROM Driver Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During

system initialization, each instance of the Data EEPROM Driver would be initialized with the following configuration settings passed dynamically at run time using DRV_EEPROM_INIT, that are supported by the specific EEPROM driver:

- Device requested power state: One of the system module power states. For specific details please refer to "Data Types and Constants" in the Library Interfacesection.
- The actual peripheral ID enumerated as the PLIB level module ID (e.g., NVM_ID_0)
- EEPROM Media Geometry

The DRV_EEPROM_Initialize function configures and initializes the EEPROM driver using the configuration information provided. It returns an object handle of the type SYS_MODULE_OBJ. This object handle would be used by other system interfaces such as DRV_EEPROM_Status, DRV_EEPROM_Tasks and DRV_EEPROM_Deinitialize.

Example:

```
/*** Data EEPROM Driver Initialization Data ***/
SYS_FS_MEDIA_REGION_GEOMETRY EEPROMGeometryTable[3] =
{
    {
        .blockSize = 4,
        .numBlocks = (DRV_EEPROM_MEDIA_SIZE * 1024),
    },
    {
       .blockSize = 4,
       .numBlocks = ((DRV_EEPROM_MEDIA_SIZE * 1024)/4)
    },
    {
       .blockSize = 4,
       .numBlocks = ((DRV_EEPROM_MEDIA_SIZE * 1024)/4)
    }
};

const SYS_FS_MEDIA_GEOMETRY EEPROMGeometry =
{
    .mediaProperty = SYS_FS_MEDIA_WRITE_IS_BLOCKING,
    .numReadRegions = 1,
    .numWriteRegions = 1,
    .numEraseRegions = 1,
    .geometryTable = (SYS_FS_MEDIA_REGION_GEOMETRY *)&EEPROMGeometryTable
};

const DRV_EEPROM_INIT drvEepromInit =
{
    .moduleInit.sys.powerState = SYS_MODULE_POWER_RUN_FULL,
    .eepromId = NVM_ID_0,
    .eepromMediaGeometry = (SYS_FS_MEDIA_GEOMETRY *)&EEPROMGeometry
};

/* Initialize the Data EEPROM Driver */
sysObj.drvEeprom = DRV_EEPROM_Initialize(DRV_EEPROM_INDEX_0, (SYS_MODULE_INIT *)&drvEepromInit);
```

Data EEPROM Driver Task Routine

The Data EEPROM Driver task routine DRV_EEPROM_Tasks, will be called from the system task routine, SYS_Tasks. The driver task routine is responsible maintaining the driver state machine. The block operation requests from the application or from other modules are added to the driver queue.

Data EEPROM Driver Status

DRV_EEPROM_Status() returns the current status of the Data EEPROM Driver and is called by the MPLAB Harmony System. The application may not find the need to call this function directly.

Example:

```
SYS_MODULE_OBJ object;
// Returned from DRV_EEPROM_Initialize
SYS_STATUS eepromStatus;

eepromStatus = DRV_EEPROM_Status(object);
if (SYS_STATUS_ERROR >= eepromStatus)
{
    // Handle error
}
```

Client Core Functions

Opening the Driver

For the application to start using an instance of the module, it must call the DRV_EEPROM_Open function repeatedly until a valid handle is returned by the driver. The application client uses this driver handle to access the driver functions.

For the various options available for I/O INTENT please refer to Data Types and Constants in the Library Interface section.

Example:

```
eepromHandle = DRV_EEPROM_Open(DRV_EEPROM_INDEX_0, DRV_IO_INTENT_READWRITE);
if (DRV_HANDLE_INVALID == eepromHandle)
{
    /* Call until the function returns a valid handle. */
}
else
{
    /* Do further processing. */
}
```

Closing the Driver

Closes an opened-instance of the Data EEPROM Driver. This invalidates the driver handle. The application must open the driver again to obtain a valid handle.

Example:

```
DRV_HANDLE eepromHandle; // Returned from DRV_EEPROM_Open
DRV_EEPROM_Close(eepromHandle);
```

Client Block Operation Functions

The driver provides client interfaces to perform operations in terms of blocks. A block is a unit that represents the minimum amount of data that can be erased, written, or read. The block sizes may differ for Erase, Write, and Read operations. The DRV_EEPROM_GeometryGet function can be used to read out the geometry of the EEPROM device. The geometry indicates the number of read, write and erase regions, blocks per region and the size of each block.

The DRV_EEPROM_Erase, DRV_EEPROM_Write, and DRV_EEPROM_Read functions are used to erase, write, and read the data to/from EEPROM devices. In addition to these functions, the driver also provides the DRV_EEPROM_BulkErase function that erases the entire EEPROM.

These functions are non-blocking in nature and queue the operation request into the driver queue. All of the requests in the queue are executed by the DRV_EEPROM_Tasks function one-by-one. A command handle associated with the operation request is returned to the application client when the operation request is queued at the driver. This handle allows the application client to track the request as it progresses through the queue. The handle expires when the request processing is complete. The driver provides events (DRV_EEPROM_EVENT) that indicate the completion of the requests.

The following steps can be performed for a simple Block Data Operation:

1. The system should have completed necessary initialization of the Data EEPROM Driver, and the DRV_EEPROM_Tasks function should be running in a polled environment.
2. Open the driver using DRV_EEPROM_Open with the necessary intent.
3. Set an event handler callback using the function DRV_EEPROM_EventHandlerSet.
4. Request for block operations using the functions, DRV_EEPROM_Erase, DRV_EEPROM_Write, DRV_EEPROM_Read and DRV_EEPROM_BulkErase with the appropriate parameters.
5. Wait for event handler callback to occur and check the status of the block operation using the callback function parameter of type DRV_EEPROM_ EVENT.
6. After performing the required block operations, the client can close the driver using the function , DRV_EEPROM_Close .

Example:

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_EEPROM_COMMAND_HANDLE commandHandle;

// drvEEPROMHandle is the handle returned by the DRV_EEPROM_Open // function. Client registers an event
handler with driver. This // is done once.

DRV_EEPROM_EventHandlerSet(drvEEPROMHandle, APP_EEPROMEventHandler, (uintptr_t)&myAppObj);

DRV_EEPROM_Read(drvEEPROMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_EEPROM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
```

```
}

// Event Processing Technique. Event is received when operation // is done.

void APP_EEPROMEventHandler
(
    DRV_EEPROM_EVENT event,
    DRV_EEPROM_COMMAND_HANDLE commandHandle,
    uintptr_t contextHandle
)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event
    // handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_EEPROM_EVENT_COMMAND_COMPLETE:
            // Operation completed successfully.
            break;

        case DRV_EEPROM_EVENT_COMMAND_ERROR:
            // Error handling here.
            break;

        default:
            break;
    }
}
```

Media Interface Functions

Reading the Device Geometry

The application can call the DRV_EEPROM_GeometryGet function to obtain the geometry of the EEPROM device. The geometry indicates the number of read, write and erase regions, number of blocks per region and the size of each block.

Example:

```
SYS_FS_MEDIA_GEOMETRY * eepromGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalSize;

eepromGeometry = DRV_EEPROM_GeometryGet(eepromOpenHandle1);

readBlockSize = eepromGeometry->geometryTable->blockSize;
nReadBlocks = eepromGeometry->geometryTable->numBlocks;
nReadRegions = eepromGeometry->numReadRegions;

writeBlockSize = (eepromGeometry->geometryTable +1)->blockSize;
eraseBlockSize = (eepromGeometry->geometryTable +2)->blockSize;

//The below expression provides the EEPROM memory size.
totalSize = readBlockSize * nReadBlocks * nReadRegions;
```

Configuring the Library

Macros

	Name	Description
	DRV_EEPROM_BUFFER_OBJECT_NUMBER	Selects the maximum number of buffer objects
	DRV_EEPROM_CLIENTS_NUMBER	Selects the maximum number of clients
	DRV_EEPROM_INSTANCES_NUMBER	Selects the maximum number of Driver instances that can be supported by the dynamic driver.
	DRV_EEPROM_MEDIA_SIZE	Specifies the EEPROM Media size.
	DRV_EEPROM_SYS_FS_REGISTER	Register to use with the File system

Description

The configuration of the Data EEPROM Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the Data EEPROM Driver. Based on the selections made, the Data EEPROM Driver may support the selected features. These configuration settings will apply to all instances of the Data EEPROM Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_EEPROM_BUFFER_OBJECT_NUMBER Macro

Selects the maximum number of buffer objects

File

drv_eeprom_config_template.h

C

```
#define DRV_EEPROM_BUFFER_OBJECT_NUMBER 5
```

Description

EEPROM Driver maximum number of buffer objects

This definition selects the maximum number of buffer objects. This indirectly also specifies the queue depth. The EEPROM Driver can queue up to DRV_EEPROM_BUFFER_OBJECT_NUMBER of read/write requests before returning a DRV_EEPROM_BUFFER_HANDLE_INVALID due to the queue being full. Buffer objects are shared by all instances of the driver. Increasing this number increases the RAM requirement of the driver.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_EEPROM_CLIENTS_NUMBER Macro

Selects the maximum number of clients

File

drv_eeprom_config_template.h

C

```
#define DRV_EEPROM_CLIENTS_NUMBER 1
```

Description

EEPROM maximum number of clients

This definition selects the maximum number of clients that the EEPROM driver can support at run time. This constant defines the total number of EEPROM driver clients that will be available to all instances of the EEPROM driver.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_EEPROM_INSTANCES_NUMBER Macro

Selects the maximum number of Driver instances that can be supported by the dynamic driver.

File

drv_eeprom_config_template.h

C

```
#define DRV_EEPROM_INSTANCES_NUMBER 1
```

Description

EEPROM Driver instance configuration

This definition selects the maximum number of Driver instances that can be supported by the dynamic driver. In case of this driver, multiple instances of the driver could use the same hardware instance.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_EEPROM_MEDIA_SIZE Macro

Specifies the EEPROM Media size.

File

drv_eeprom_config_template.h

C

```
#define DRV_EEPROM_MEDIA_SIZE 32
```

Description

EEPROM Media Size

This definition specifies the EEPROM Media Size to be used. The size is specified in number of Kilo Bytes. The media size MUST never exceed physical available EEPROM Memory size. Application code requirements should be kept in mind while defining this parameter.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_EEPROM_SYS_FS_REGISTER Macro

Register to use with the File system

File

drv_eeprom_config_template.h

C

```
#define DRV_EEPROM_SYS_FS_REGISTER
```

Description

EEPROM Driver Register with File System

Specifying this macro enables the EEPROM driver to register its services with the SYS FS.

Remarks

This macro is optional and should be specified only if the EEPROM driver is to be used with the File System.

Building the Library

This section lists the files that are available in the Data EEPROM Driver Library.

Description

This section lists the files that are available in the `\src` folder of the Data EEPROM Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/eeprom`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_eeprom.h`	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_eeprom.c`	Basic Data EEPROM Driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The Data EEPROM Driver Library is not dependent upon any modules.

Library Interface

a) System Functions

	Name	Description
	DRV_EEPROM_Initialize	Initializes the EEPROM instance for the specified driver index.
	DRV_EEPROM_Deinitialize	Deinitializes the specified instance of the EEPROM driver module
	DRV_EEPROM_Status	Gets the current status of the EEPROM driver module.
	DRV_EEPROM_Tasks	Handles the read or write requests queued to the driver.

b) Client Core Functions

	Name	Description
	DRV_EEPROM_Close	Closes an opened-instance of the EEPROM driver
	DRV_EEPROM_Open	Opens the specified EEPROM driver instance and returns a handle to it

c) Block Operation Functions

	Name	Description
	DRV_EEPROM_BulkErase	Performs a bulk erase of the entire Data EEPROM.
	DRV_EEPROM_Erase	Erases blocks of data starting from the specified block address.
	DRV_EEPROM_Read	Reads blocks of data from the specified address in EEPROM memory.
	DRV_EEPROM_Write	Writes blocks of data starting from the specified address in EEPROM memory.

d) Media Interface Functions

	Name	Description
	DRV_EEPROM_AddressGet	Returns the EEPROM media start address
	DRV_EEPROM_CommandStatus	Gets the current status of the command.
	DRV_EEPROM_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.
	DRV_EEPROM_GeometryGet	Returns the geometry of the device.
	DRV_EEPROM_IsAttached	Returns the physical attach status of the EEPROM.
	DRV_EEPROM_IsWriteProtected	Returns the write protect status of the EEPROM.

e) Data Types and Constants

	Name	Description
	DRV_EEPROM_COMMAND_HANDLE_INVALID	This value defines the EEPROM Driver's Invalid Command Handle.
	DRV_EEPROM_INDEX_0	EEPROM driver index definition
	DRV_EEPROM_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_EEPROM_COMMAND_STATUS	Specifies the status of the command for read or write requests.
	DRV_EEPROM_EVENT	Identifies the possible events that can result from a request.
	DRV_EEPROM_EVENT_HANDLER	Pointer to a EEPROM Driver Event handler function
	DRV_EEPROM_INIT	Defines the data required to initialize the EEPROM driver

Description

This section describes the Application Programming Interface (API) functions of the Data EEPROM Driver Library.

a) System Functions

DRV_EEPROM_Initialize Function

Initializes the EEPROM instance for the specified driver index.

File

drv_eeprom.h

C

```
SYS_MODULE_OBJ DRV_EEPROM_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the EEPROM driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This routine must be called before any other EEPROM routine is called.

This routine should only be called once during system initialization unless DRV_EEPROM_Deinitialize is called to deinitialize the driver instance.

This routine will NEVER block for hardware access. The system must use DRV_EEPROM_Status to find out when the driver is in the ready state.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this routine.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the EEPROM Driver.

SYS_MODULE_OBJ  objectHandle;

SYS_FS_MEDIA_REGION_GEOMETRY EEPROMGeometryTable[3] =
{
    {
        .blockSize = 4,
        .numBlocks = (DRV_EEPROM_MEDIA_SIZE * 1024),
    },
    {
        .blockSize = 4,
        .numBlocks = ((DRV_EEPROM_MEDIA_SIZE * 1024)/4)
    },
    {
        .blockSize = 4,
        .numBlocks = ((DRV_EEPROM_MEDIA_SIZE * 1024)/4)
    }
};

const SYS_FS_MEDIA_GEOMETRY EEPROMGeometry =
{
    .mediaProperty = SYS_FS_MEDIA_WRITE_IS_BLOCKING,
    .numReadRegions = 1,
    .numWriteRegions = 1,
    .numEraseRegions = 1,
    .geometryTable = (SYS_FS_MEDIA_REGION_GEOMETRY *)&EEPROMGeometryTable
};

// EEPROM Driver Initialization Data
const DRV_EEPROM_INIT drvEepromInit =
{
    .moduleInit.sys.powerState = SYS_MODULE_POWER_RUN_FULL,
    .eepromId = NVM_ID_0,
    .eepromMediaGeometry = (SYS_FS_MEDIA_GEOMETRY *)&EEPROMGeometry
};
```

```
objectHandle = DRV_EEPROM_Initialize(DRV_EEPROM_INDEX_0, (SYS_MODULE_INIT*)&drvEepromInit);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized.
init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_EEPROM_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

);

DRV_EEPROM_Deinitialize Function

Deinitializes the specified instance of the EEPROM driver module

File

drv_eeprom.h

C

```
void DRV_EEPROM_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the EEPROM driver module, disabling its operation. Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

Preconditions

Function DRV_EEPROM_Initialize should have been called before calling this function.

Parameter: object - Driver object handle, returned from the DRV_EEPROM_Initialize routine

Example

```
// This code snippet shows an example of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_EEPROM_Initialize
SYS_STATUS          status;

DRV_EEPROM_Deinitialize(object);

status = DRV_EEPROM_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know when the driver is
    // deinitialized.
}
```

Function

void DRV_EEPROM_Deinitialize

(

SYS_MODULE_OBJ object

);

DRV_EEPROM_Status Function

Gets the current status of the EEPROM driver module.

File

drv_eeprom.h

C

```
SYS_STATUS DRV_EEPROM_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and accept requests for new operations.

SYS_STATUS_UNINITIALIZED - Indicates the driver is not initialized.

Description

This routine provides the current status of the EEPROM driver module.

Remarks

None.

Preconditions

Function DRV_EEPROM_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_EEPROM_Initialize
SYS_STATUS          EEPROMStatus;

EEPROMStatus = DRV_EEPROM_Status(object);
if (EEPROMStatus == SYS_STATUS_READY)
{
    // Driver is ready to perform operations.
}
else
{
    // Driver is not ready.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_EEPROM_Initialize routine

Function

SYS_STATUS DRV_EEPROM_Status

(

SYS_MODULE_OBJ object

);

DRV_EEPROM_Tasks Function

Handles the read or write requests queued to the driver.

File

drv_eeprom.h

C

```
void DRV_EEPROM_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to handle the read or write requests queued to the driver.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

Preconditions

The DRV_EEPROM_Initialize routine must have been called for the specified EEPROM driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_EEPROM_Initialize

while (true)
{
    DRV_EEPROM_Tasks (object);
    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_EEPROM_Initialize)

Function

void DRV_EEPROM_Tasks

(

SYS_MODULE_OBJ object

);

b) Client Core Functions

DRV_EEPROM_Close Function

Closes an opened-instance of the EEPROM driver

File

drv_eeprom.h

C

```
void DRV_EEPROM_Close(const DRV_HANDLE handle);
```

Returns

None

Description

This routine closes an opened-instance of the EEPROM driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_EEPROM_Open before the caller may use the driver again. Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_EEPROM_Initialize routine must have been called for the specified EEPROM driver instance.

DRV_EEPROM_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_EEPROM_Open

DRV_EEPROM_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_EEPROM_Close
(
const DRV_HANDLE handle
);

DRV_EEPROM_Open Function

Opens the specified EEPROM driver instance and returns a handle to it

File

drv_eeprom.h

C

```
DRV_HANDLE DRV_EEPROM_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, DRV_HANDLE_INVALID is returned. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_EEPROM_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver hardware instance being opened is invalid

Description

This routine opens the specified EEPROM driver instance and provides a handle. This handle must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_EEPROM_Close routine is called. This routine will NEVER block waiting for hardware. If the driver has already been opened, it cannot be opened exclusively.

Preconditions

DRV_EEPROM_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_EEPROM_Open(DRV_EEPROM_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_EEPROM_Open
(
const SYS_MODULE_INDEX index,
const DRV_IO_INTENT ioIntent
);

c) Block Operation Functions

DRV_EEPROM_BulkErase Function

Performs a bulk erase of the entire Data EEPROM.

File

drv_eeprom.h

C

```
void DRV_EEPROM_BulkErase(const DRV_HANDLE handle, DRV_EEPROM_COMMAND_HANDLE * commandHandle);
```

Returns

If the request was queued successfully then a valid command handle is returned in the commandHandle argument. Otherwise DRV_EEPROM_COMMAND_HANDLE_INVALID is returned if the request was not successful.

Description

This function schedules a non-blocking bulk erase operation of the entire Data EEPROM. The function returns with a valid handle in the commandHandle argument if the erase request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_EEPROM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer object could not be allocated to the request
- if the client opened the driver for read only
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_EEPROM_EVENT_COMMAND_COMPLETE event if the command was processed successfully or DRV_EEPROM_EVENT_COMMAND_ERROR event if the command was not processed successfully.

Remarks

None

Refer to drv_eeprom.h for usage information.

Preconditions

The DRV_EEPROM_Initialize() routine must have been called for the specified EEPROM driver instance.

DRV_EEPROM_Open() routine must have been called to obtain a valid opened device handle. DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified as a parameter to this routine.

Example

```
DRV_EEPROM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myEEPROMHandle is the handle returned by the DRV_EEPROM_Open function.
// Client registers an event handler with driver

DRV_EEPROM_EventHandlerSet(myEEPROMHandle, APP_EEPROMEventHandler, (uintptr_t)&myAppObj);
DRV_EEPROM_BulkErase(myEEPROMHandle, &commandHandle);

if(DRV_EEPROM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_EEPROMEventHandler
(
    DRV_EEPROM_EVENT event,
    DRV_EEPROM_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
```

```
    // context points to myAppObj.
    switch(event)
    {
        case DRV_EEPROM_EVENT_COMMAND_COMPLETE:

            // Bulk Erase operation is complete.
            break;

        case DRV_EEPROM_EVENT_COMMAND_ERROR:

            // Bulk Erase operation failed.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle

Function

void DRV_EEPROM_BulkErase
(
const DRV_HANDLE handle,
DRV_EEPROM_COMMAND_HANDLE * commandHandle
);

DRV_EEPROM_Erase Function

Erases blocks of data starting from the specified block address.

File

drv_eeprom.h

C

```
void DRV_EEPROM_Erase(const DRV_HANDLE handle, DRV_EEPROM_COMMAND_HANDLE * commandHandle, uint32_t
blockStart, uint32_t nBlock);
```

Returns

If the request was queued successfully then a valid command handle is returned in the commandHandle argument. Otherwise DRV_EEPROM_COMMAND_HANDLE_INVALID is returned if the request was not successful.

Description

This function schedules a non-blocking erase operation for erasing blocks of memory. The function returns with a valid handle in the commandHandle argument if the erase request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_EEPROM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer object could not be allocated to the request
- if the client opened the driver for read only
- if the number of blocks to be erased is either zero or more than the number of blocks actually available
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_EEPROM_EVENT_COMMAND_COMPLETE event if the command was processed successfully or DRV_EEPROM_EVENT_COMMAND_ERROR event if the command was not processed successfully.

Remarks

None

Preconditions

The DRV_EEPROM_Initialize() routine must have been called for the specified EEPROM driver instance.

DRV_EEPROM_Open() routine must have been called to obtain a valid opened device handle. DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified as a parameter to this routine.

Example

```
uint32_t blockStart = 0;
uint32_t nBlock = 2;
DRV_EEPROM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myEEPROMHandle is the handle returned by the DRV_EEPROM_Open function.
// Client registers an event handler with driver

DRV_EEPROM_EventHandlerSet(myEEPROMHandle, APP_EEPROMEventHandler, (uintptr_t)&myAppObj);
DRV_EEPROM_Erase(myEEPROMHandle, &commandHandle, blockStart, nBlock);

if(DRV_EEPROM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_EEPROMEventHandler
(
    DRV_EEPROM_EVENT event,
    DRV_EEPROM_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    // context points to myAppObj.
    switch(event)
    {
        case DRV_EEPROM_EVENT_COMMAND_COMPLETE:

            // Erase operation is complete.
            break;

        case DRV_EEPROM_EVENT_COMMAND_ERROR:

            // Erase operation failed.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
blockStart	block start addess for the erase operation.
nBlock	Total number of blocks to be erased.

Function

void DRV_EEPROM_Erase

(

const DRV_HANDLE handle,

DRV_EEPROM_COMMAND_HANDLE * commandHandle,

uint32_t blockStart,

uint32_t nBlock

);

DRV_EEPROM_Read Function

Reads blocks of data from the specified address in EEPROM memory.

File

drv_eeprom.h

C

```
void DRV_EEPROM_Read(const DRV_HANDLE handle, DRV_EEPROM_COMMAND_HANDLE * commandHandle, void * buffer,
uint32_t blockStart, uint32_t nBlock);
```

Returns

If the request was queued successfully then a valid command handle is returned in the commandHandle argument. Otherwise DRV_EEPROM_COMMAND_HANDLE_INVALID is returned if the request was not successful.

Description

This function schedules a non-blocking read operation for reading blocks of data from the EEPROM memory. The function returns with a valid handle in the commandHandle argument if the read request was scheduled successfully. The function adds the request to the driver instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_EEPROM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer object could not be allocated to the request
- if the buffer pointer is NULL
- if the queue size is full or queue depth is insufficient
- if the driver handle is invalid
- if the number of blocks to be read is zero or more than the actual number of blocks available
- if the client opened the driver in write only mode

If the requesting client registered an event callback with the driver, the driver will issue a DRV_EEPROM_EVENT_COMMAND_COMPLETE event if the command was processed successfully or DRV_EEPROM_EVENT_COMMAND_ERROR event if the command was not processed successfully.

Remarks

None.

Preconditions

The DRV_EEPROM_Initialize routine must have been called for the specified EEPROM driver instance.

DRV_EEPROM_Open must have been called with DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE as the ioIntent to obtain a valid opened device handle.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];
// address should be block aligned.
uint32_t blockStart = EEPROM_BASE_ADDRESS_TO_READ_FROM;
uint32_t nBlock = 2;
DRV_EEPROM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myEEPROMHandle is the handle returned by the DRV_EEPROM_Open function.
DRV_EEPROM_EventHandlerSet(myEEPROMHandle, APP_EEPROMEventHandler, (uintptr_t)&myAppObj);
DRV_EEPROM_Read(myEEPROMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_EEPROM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}
else
{
    // Read queued successfully.
}

// Event is received when the buffer is processed.

void APP_EEPROMEventHandler
(
    DRV_EEPROM_EVENT event,
```

```
    DRV_EEPROM_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    // context points to myAppObj.

    switch(event)
    {
        case DRV_EEPROM_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_EEPROM_EVENT_COMMAND_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
buffer	Buffer into which the data read from the EEPROM memory will be placed
blockStart	Start block address in EEPROM memory from where the read should begin.
nBlock	Total number of blocks to be read.

Function

void DRV_EEPROM_Read
(
const DRV_HANDLE handle,
DRV_EEPROM_COMMAND_HANDLE * commandHandle,
void * buffer,
uint32_t blockStart,
uint32_t nBlock
);

DRV_EEPROM_Write Function

Writes blocks of data starting from the specified address in EEPROM memory.

File

drv_eeprom.h

C

```
void DRV_EEPROM_Write(const DRV_HANDLE handle, DRV_EEPROM_COMMAND_HANDLE * commandHandle, void * buffer,
uint32_t blockStart, uint32_t nBlock);
```

Returns

If the request was queued successfully then a valid command handle is returned in the commandHandle argument. Otherwise DRV_EEPROM_COMMAND_HANDLE_INVALID is returned if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data into memory. The function returns with a valid handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_EEPROM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer object could not be allocated to the request
- if the buffer pointer is NULL

- if the client opened the driver for read only
- if the number of blocks to be written is either zero or more than the number of blocks actually available
- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_EEPROM_EVENT_COMMAND_COMPLETE event if the command was processed successfully or DRV_EEPROM_EVENT_COMMAND_ERROR event if the command was not processed successfully.

Remarks

None

Preconditions

The DRV_EEPROM_Initialize() routine must have been called for the specified EEPROM driver instance.

DRV_EEPROM_Open() routine must have been called to obtain a valid opened device handle. DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified as a parameter to this routine.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

uint32_t blockStart = EEPROM_BASE_ADDRESS_TO_WRITE_TO;
uint32_t nBlock = 2;
DRV_EEPROM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myEEPROMHandle is the handle returned by the DRV_EEPROM_Open function.
// Client registers an event handler with driver

DRV_EEPROM_EventHandlerSet(myEEPROMHandle, APP_EEPROMEventHandler, (uintptr_t)&myAppObj);
DRV_EEPROM_Write(myEEPROMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_EEPROM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_EEPROMEventHandler
(
    DRV_EEPROM_EVENT event,
    DRV_EEPROM_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    // context points to myAppObj.
    switch(event)
    {
        case DRV_EEPROM_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_EEPROM_EVENT_COMMAND_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle

buffer	The buffer containing data to be programmed into EEPROM memory
blockStart	Start block address of EEPROM memory where the write should begin.
nBlock	Total number of blocks to be written.

Function

void DRV_EEPROM_Write
(
const DRV_HANDLE handle,
DRV_EEPROM_COMMAND_HANDLE * commandHandle,
void * buffer,
uint32_t blockStart,
uint32_t nBlock
);

d) Media Interface Functions

DRV_EEPROM_AddressGet Function

Returns the EEPROM media start address

File

drv_eeprom.h

C

```
uintptr_t DRV_EEPROM_AddressGet(const DRV_HANDLE handle);
```

Returns

Start address of the EEPROM Media if the handle is valid otherwise NULL.

Description

This function returns the EEPROM Media start address.

Remarks

None.

Preconditions

The DRV_EEPROM_Initialize() routine must have been called for the specified EEPROM driver instance.

The DRV_EEPROM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
uintptr_t startAddress;
startAddress = DRV_EEPROM_AddressGet(drvEEPROMHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

uintptr_t DRV_EEPROM_AddressGet
(
const DRV_HANDLE handle
);

DRV_EEPROM_CommandStatus Function

Gets the current status of the command.

File

drv_eeprom.h

C

```
DRV_EEPROM_COMMAND_STATUS DRV_EEPROM_CommandStatus(const DRV_HANDLE handle, const DRV_EEPROM_COMMAND_HANDLE
commandHandle);
```

Returns

A DRV_EEPROM_COMMAND_STATUS value describing the current status of the command.

Description

This routine gets the current status of the command. The application must use this routine where the status of a scheduled command needs to be polled on. The function may return DRV_EEPROM_COMMAND_HANDLE_INVALID in a case where the command handle has expired. A command handle expires when the internal buffer object is re-assigned to another read, write or erase request. It is recommended that this function be called regularly in order to track the command status correctly.

The application can alternatively register an event handler to receive read, write or erase operation completion events.

Remarks

This routine will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_EEPROM_Initialize() routine must have been called.

The DRV_EEPROM_Open() must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE                handle;         // Returned from DRV_EEPROM_Open
DRV_EEPROM_COMMAND_HANDLE commandHandle;
DRV_EEPROM_COMMAND_STATUS status;

status = DRV_EEPROM_CommandStatus(handle, commandHandle);
if(status == DRV_EEPROM_COMMAND_COMPLETED)
{
    // Operation Done
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
commandHandle	A valid command handle returned from read, write or erase request.

Function

DRV_EEPROM_COMMAND_STATUS DRV_EEPROM_CommandStatus

(

const DRV_HANDLE handle,

const DRV_EEPROM_COMMAND_HANDLE commandHandle

);

DRV_EEPROM_EventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

File

drv_eeprom.h

C

```
void DRV_EEPROM_EventHandlerSet(const DRV_HANDLE handle, const void * eventHandler, const uintptr_t
context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client calls a read, write or a erase function, it is provided with a handle identifying the command that was added to the driver's command queue. The driver will pass this handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any read, write or erase operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_EEPROM_Initialize() routine must have been called for the specified EEPROM driver instance.

The DRV_EEPROM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_EEPROM_COMMAND_HANDLE commandHandle;

// drvEEPROMHandle is the handle returned by the DRV_EEPROM_Open function.
// Client registers an event handler with driver. This is done once.

DRV_EEPROM_EventHandlerSet(drvEEPROMHandle, APP_EEPROMEventHandler, (uintptr_t)&myAppObj);

DRV_EEPROM_Read(drvEEPROMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_EEPROM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_EEPROMEventHandler
(
    DRV_EEPROM_EVENT event,
    DRV_EEPROM_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_EEPROM_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_EEPROM_EVENT_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_EEPROM_EventHandlerSet
(
const DRV_HANDLE handle,
const void * eventHandler,
const uintptr_t context
);

DRV_EEPROM_GeometryGet Function

Returns the geometry of the device.

File

drv_eeprom.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_EEPROM_GeometryGet(const DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Pointer to structure which holds the media geometry information.

Description

This API gives the following geometrical details of the EEPROM memory:

- Media Property
- Number of Read/Write/Erase regions
- Number of Blocks and their size in each region of the device

Remarks

None.

Preconditions

The DRV_EEPROM_Initialize() routine must have been called for the specified EEPROM driver instance.

The DRV_EEPROM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
SYS_FS_MEDIA_GEOMETRY * eepromGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalSize;

eepromGeometry = DRV_EEPROM_GeometryGet(eepromOpenHandle1);

readBlockSize  = eepromGeometry->geometryTable->blockSize;
nReadBlocks = eepromGeometry->geometryTable->numBlocks;
nReadRegions = eepromGeometry->numReadRegions;

writeBlockSize  = (eepromGeometry->geometryTable +1)->blockSize;
eraseBlockSize  = (eepromGeometry->geometryTable +2)->blockSize;

totalSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY * DRV_EEPROM_GeometryGet
(
const DRV_HANDLE handle
);

DRV_EEPROM_IsAttached Function

Returns the physical attach status of the EEPROM.

File

drv_eeprom.h

C

```
bool DRV_EEPROM_IsAttached(const DRV_HANDLE handle);
```

Returns

Returns true always

Description

This function returns the physical attach status of the EEPROM.

Remarks

None.

Preconditions

The DRV_EEPROM_Initialize() routine must have been called for the specified EEPROM driver instance.

The DRV_EEPROM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// The EEPROM media is always attached and so the below always returns
// true.

bool isEEPROMAttached;
isEEPROMAttached = DRV_EEPROM_isAttached(drvEEPROMHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_EEPROM_IsAttached
(
const DRV_HANDLE handle
);

DRV_EEPROM_IsWriteProtected Function

Returns the write protect status of the EEPROM.

File

drv_eeprom.h

C

```
bool DRV_EEPROM_IsWriteProtected(const DRV_HANDLE handle);
```

Returns

Always returns false.

Description

This function returns the physical attach status of the EEPROM. This function always returns false.

Remarks

None.

Preconditions

The DRV_EEPROM_Initialize() routine must have been called for the specified EEPROM driver instance.

The DRV_EEPROM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// The EEPROM media is treated as always writeable.
bool isWriteProtected;
isWriteProtected = DRV_EEPROM_IsWriteProtected(drvEEPROMHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_EEPROM_IsWriteProtected

(

const DRV_HANDLE handle

);

e) Data Types and Constants

DRV_EEPROM_COMMAND_HANDLE_INVALID Macro

This value defines the EEPROM Driver's Invalid Command Handle.

File

drv_eeprom.h

C

```
#define DRV_EEPROM_COMMAND_HANDLE_INVALID SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID
```

Description

EEPROM Driver Invalid Command Handle.

This value defines the EEPROM Driver Invalid Command Handle. This value is returned by read or write routines when the command request was not accepted.

Remarks

None.

DRV_EEPROM_INDEX_0 Macro

EEPROM driver index definition

File

drv_eeprom.h

C

```
#define DRV_EEPROM_INDEX_0 0
```

Description

Driver EEPROM Module Index reference

This constant provides EEPROM driver index definition.

Remarks

This constant should be used in place of hard-coded numeric literals. This value should be passed into the DRV_EEPROM_Initialize and DRV_EEPROM_Open routines to identify the driver instance in use.

DRV_EEPROM_COMMAND_HANDLE Type

Handle identifying commands queued in the driver.

File

drv_eeprom.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_EEPROM_COMMAND_HANDLE;
```

Description

EEPROM Driver command handle.

A command handle is returned by a call to the read or write functions. This handle allows the application to track the completion of the operation. This command handle is also returned to the client along with the event that has occurred with respect to the command. This allows the application to connect the event to a specific command in case where multiple commands are queued.

The command handle associated with the command request expires when the client has been notified of the completion of the command (after event handler function that notifies the client returns) or after the command has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_EEPROM_COMMAND_STATUS Enumeration

Specifies the status of the command for read or write requests.

File

drv_eeprom.h

C

```
typedef enum {
  DRV_EEPROM_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED,
  DRV_EEPROM_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED,
  DRV_EEPROM_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS,
  DRV_EEPROM_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN
} DRV_EEPROM_COMMAND_STATUS;
```

Members

Members	Description
DRV_EEPROM_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED	Done OK and ready
DRV_EEPROM_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED	Scheduled but not started
DRV_EEPROM_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS	Currently being in transfer
DRV_EEPROM_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN	Unknown Command

Description

EEPROM Driver Command Status

EEPROM Driver command Status

This type specifies the status of the command for the read or write requests.

Remarks

None.

DRV_EEPROM_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_eeprom.h

C

```
typedef enum {
  DRV_EEPROM_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_EEPROM_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR
} DRV_EEPROM_EVENT;
```

Members

Members	Description
DRV_EEPROM_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE	Operation has been completed successfully.
DRV_EEPROM_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR	There was an error during the operation

Description

EEPROM Driver Events

This enumeration identifies the possible events that can result from a read or write request caused by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_EEPROM_EventHandlerSet function when a request is completed.

DRV_EEPROM_EVENT_HANDLER Type

Pointer to a EEPROM Driver Event handler function

File

drv_eeprom.h

C

```
typedef SYS_FS_MEDIA_EVENT_HANDLER DRV_EEPROM_EVENT_HANDLER;
```

Returns

None.

Description

EEPROM Driver Event Handler Function Pointer

This data type defines the required function signature for the EEPROM event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event callbacks from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_EEPROM_EVENT_COMMAND_COMPLETE, it means that the scheduled operation was completed successfully.

If the event is DRV_EEPROM_EVENT_COMMAND_ERROR, it means that the scheduled operation was not completed successfully.

The context parameter contains the handle to the client context, provided at the time the event handling function was registered using the DRV_EEPROM_EventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that scheduled the request.

The event handler function executes in the driver's context. It is recommended of the application to not perform process intensive or blocking operations within this function.

Example

```
void APP_MyEepromEventHandler
(
    DRV_EEPROM_EVENT event,
    DRV_EEPROM_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_EEPROM_EVENT_COMMAND_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_EEPROM_EVENT_COMMAND_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read or Write requests
context	Value identifying the context of the application that registered the event handling function

DRV_EEPROM_INIT Structure

Defines the data required to initialize the EEPROM driver

File

drv_eeprom.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  NVM_MODULE_ID eepromId;
  const SYS_FS_MEDIA_GEOMETRY * eepromMediaGeometry;
} DRV_EEPROM_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
NVM_MODULE_ID eepromId;	Identifies hardware module (PLIB-level) ID
const SYS_FS_MEDIA_GEOMETRY * eepromMediaGeometry;	EEPROM Media geometry object.

Description

EEPROM Driver Initialization Data

This data type defines the data required to initialize the EEPROM driver.

Remarks

None.

Files

Files

Name	Description
drv_eeprom.h	EEPROM Driver Interface Definition
drv_eeprom_config_template.h	EEPROM driver configuration definitions.

Description

drv_eeprom.h

EEPROM Driver Interface Definition

Enumerations

	Name	Description
	DRV_EEPROM_COMMAND_STATUS	Specifies the status of the command for read or write requests.
	DRV_EEPROM_EVENT	Identifies the possible events that can result from a request.

Functions

	Name	Description
	DRV_EEPROM_AddressGet	Returns the EEPROM media start address
	DRV_EEPROM_BulkErase	Performs a bulk erase of the entire Data EEPROM.
	DRV_EEPROM_Close	Closes an opened-instance of the EEPROM driver
	DRV_EEPROM_CommandStatus	Gets the current status of the command.
	DRV_EEPROM_Deinitialize	Deinitializes the specified instance of the EEPROM driver module
	DRV_EEPROM_Erase	Erases blocks of data starting from the specified block address.
	DRV_EEPROM_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.
	DRV_EEPROM_GeometryGet	Returns the geometry of the device.
	DRV_EEPROM_Initialize	Initializes the EEPROM instance for the specified driver index.
	DRV_EEPROM_IsAttached	Returns the physical attach status of the EEPROM.
	DRV_EEPROM_IsWriteProtected	Returns the write protect status of the EEPROM.
	DRV_EEPROM_Open	Opens the specified EEPROM driver instance and returns a handle to it
	DRV_EEPROM_Read	Reads blocks of data from the specified address in EEPROM memory.
	DRV_EEPROM_Status	Gets the current status of the EEPROM driver module.
	DRV_EEPROM_Tasks	Handles the read or write requests queued to the driver.
	DRV_EEPROM_Write	Writes blocks of data starting from the specified address in EEPROM memory.

Macros

	Name	Description
	DRV_EEPROM_COMMAND_HANDLE_INVALID	This value defines the EEPROM Driver's Invalid Command Handle.
	DRV_EEPROM_INDEX_0	EEPROM driver index definition

Structures

	Name	Description
	DRV_EEPROM_INIT	Defines the data required to initialize the EEPROM driver

Types

	Name	Description
	DRV_EEPROM_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_EEPROM_EVENT_HANDLER	Pointer to a EEPROM Driver Event handler function

Description

EEPROM Driver Interface Definition

The EEPROM driver provides a simple interface to manage the EEPROM Memory on Microchip microcontrollers. This file defines the interface

definition for the EEPROM driver.

File Name

drv_eeprom.h

Company

Microchip Technology Inc.

drv_eeprom_config_template.h

EEPROM driver configuration definitions.

Macros

	Name	Description
	DRV_EEPROM_BUFFER_OBJECT_NUMBER	Selects the maximum number of buffer objects
	DRV_EEPROM_CLIENTS_NUMBER	Selects the maximum number of clients
	DRV_EEPROM_INSTANCES_NUMBER	Selects the maximum number of Driver instances that can be supported by the dynamic driver.
	DRV_EEPROM_MEDIA_SIZE	Specifies the EEPROM Media size.
	DRV_EEPROM_SYS_FS_REGISTER	Register to use with the File system

Description

EEPROM Driver Configuration Template Header file.

This template file describes all the mandatory and optional configuration macros that are needed for building the EEPROM driver. Do not include this file in source code.

File Name

drv_eeprom_config_template.h

Company

Microchip Technology Inc.

ENC28J60 Driver Library Help

This section provides information on the ENC28J60 Driver Library.

Introduction

This library provides a driver-level abstraction of the ENC28J60 integrated Ethernet MAC and 10Base-T PHY that can be connected to the PIC32. The driver implements the virtual MAC driver model that the MPLAB Harmony TCP/IP Stack requires. Please see the TCP/IP Stack Library MAC Driver Module for details.

The "Host-To-Network"_layer of a TCP/IP stack organization covers the Data Link and Physical Layers of the standard OSI stack. The Ethernet Controller provides the Data Link or Media Access Control Layer, in addition to other functions discussed in this section.

Description

The ENC28J60 External MAC and PHY is an external module to the PIC32 that is connected through a Serial Peripheral Interface (SPI). This driver interfaces with the SPI driver to communicate with the external device to implement a complete Ethernet node in a system.

The following are some of the key features of this module:

- Supports 10 Mbps physical-to-physical layer Ethernet data transfer
  - Full-Duplex and Half-Duplex operation
  - Broadcast, Multicast and Unicast packets
  - Hardware flow control for both Full and Half-Duplex mode
- Fully configurable interrupts
- Configurable receive packet filtering using:
  - 64-bit Hash Table
  - 64-byte Pattern Match
  - Magic Packet™ Filtering
- Supports Packet Payload Checksum calculation
- CRC Check
- Supports SPI interface

Using the Library

This topic describes the basic architecture and functionality of the software driver for the ENC28J60 stand-alone Ethernet Controller with SPI, and is meant for advanced users or TCP/IP stack driver developers.

Description

The user of this driver is the MPLAB Harmony TCP/IP stack. This Ethernet driver is not intended as a system-wide driver that the application or other system modules may use. It is intended for the sole use of the MPLAB Harmony TCP/IP stack and implements the virtual MAC model required by the stack.

Interface Header File: `drv_enc28j60.h`

The interface to the ENC28J60 Driver Library is defined in the `drv_enc28j60.h` header file. Any C language source (`.c`) file that uses the ENC28J60 Driver Library should include `drv_enc28j60.h`.

Library File: The ENC28J60 Driver Library archive (`.a`) file is installed with MPLAB Harmony.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Abstraction Model

The ENC28J60 Driver Library provides the low-level abstraction of the communications protocol to communicate to the ENC28J60 external MAC though the SPI peripheral on the Microchip family of microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the ENC28J60 Driver Library interface.

Description

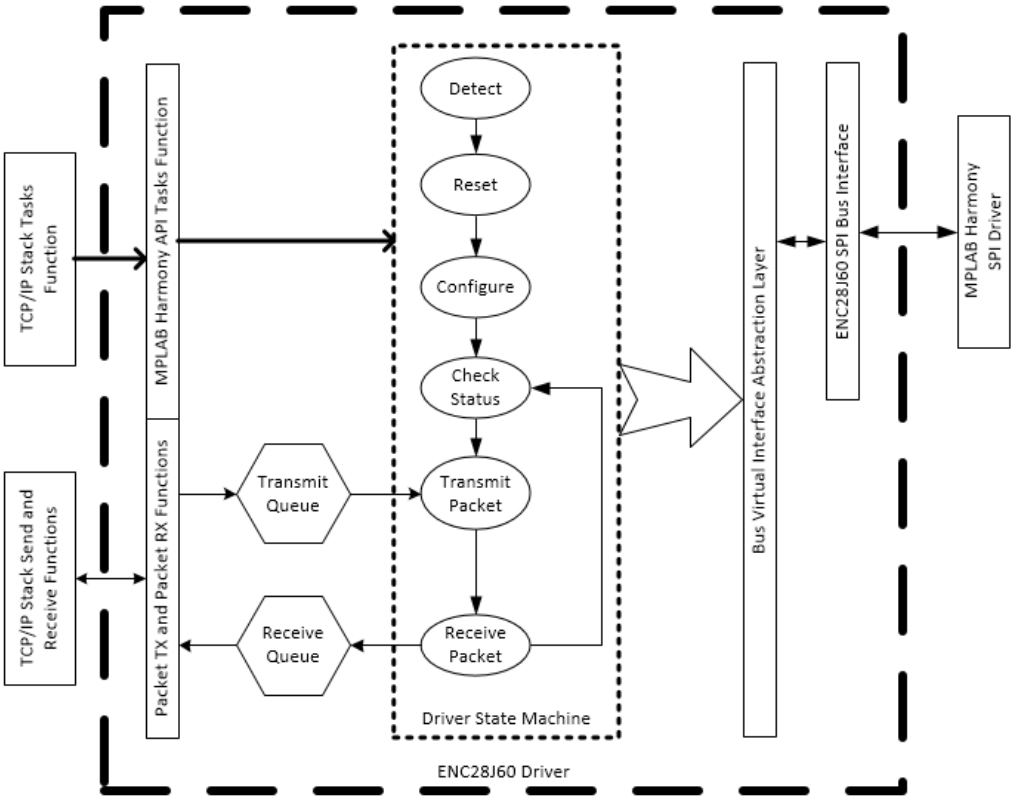
The ENC28J60 Driver library has several different layers to it, as illustrated in the following figure. The interface layer has two main sections that are used the most often: The Tasks function, and the TCP/IP Send and Receive functions.

The Tasks function manages the internal state machine which detects, resets, and then configures the ENC28J60 External MAC. It also handles the monitoring of the hardware status, sending and receiving packets.

The TCP/IP Send and Receive functions interact with the RAM-based queue of packets that are queued to send and packets that have been queued waiting for pick-up by the stack.

The main state machine does not interface directly to the SPI bus, but instead, interfaces to a virtual bus abstraction layer that allows for the replacement of the specific underlying bus implementation.

Abstraction Model



Library Overview

Refer to the section Driver Overview for how the driver operates in a system.

The library interface routines are divided into various sub-sections, each sub-section addresses one of the blocks or the overall operation of the ENC28J60 Driver Library.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Data Transfer Functions	Provides data transfer functions available in the configuration.
Status Functions	Provides status functions.
Miscellaneous Functions	Provides miscellaneous driver functions.

How the Library Works

The library provides interfaces to support the TCP/IP virtual MAC interface.

Configuring the SPI Driver

This section describes the configuration settings for the ENC28J60 Driver Library.

Description

Configuration

The ENC hardware requires a specific configuration of the SPI driver to work correctly. Inside the MHC SPI Driver configuration be sure to select:

- Run the SPI at frequencies of at least 8 MHz
- Clock mode of DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL
- Input phase of SPI_INPUT_SAMPLING_PHASE_AT_END

Recommended Settings

- Interrupt Driver mode
- Enhanced Buffer mode
- DMA mode enabled:
  - DMA block transfer size of at least 1600 bytes
  - Size of DMA buffer for dummy data of at least 1600 bytes
  - Ensure when setting up DMA in interrupt mode that the DMA interrupts are a higher priority than the SPI Driver interrupt

Example:

```
/*** SPI Driver Static Allocation Options ***/
#define DRV_SPI_INSTANCES_NUMBER           1
#define DRV_SPI_CLIENTS_NUMBER             1
#define DRV_SPI_ELEMENTS_PER_QUEUE         30

/*** SPI Driver DMA Options ***/
#define DRV_SPI_DMA_TXFER_SIZE             2048
#define DRV_SPI_DMA_DUMMY_BUFFER_SIZE      2048

/* SPI Driver Instance 0 Configuration */
#define DRV_SPI_SPI_ID_IDX0              SPI_ID_1
#define DRV_SPI_TASK_MODE_IDX0           DRV_SPI_TASK_MODE_ISR
#define DRV_SPI_SPI_MODE_IDX0            DRV_SPI_MODE_MASTER
#define DRV_SPI_ALLOW_IDLE_RUN_IDX0      false
#define DRV_SPI_SPI_PROTOCOL_TYPE_IDX0   DRV_SPI_PROTOCOL_TYPE_STANDARD
#define DRV_SPI_SPI_PROTOCOL_TYPE_IDX0   DRV_SPI_PROTOCOL_TYPE_STANDARD
#define DRV_SPI_COMM_WIDTH_IDX0          SPI_COMMUNICATION_WIDTH_8BITS
#define DRV_SPI_SPI_CLOCK_IDX0           CLK_BUS_PERIPHERAL_2
#define DRV_SPI_BAUD_RATE_IDX0           13333333
#define DRV_SPI_BUFFER_TYPE_IDX0         DRV_SPI_BUFFER_TYPE_ENHANCED
#define DRV_SPI_CLOCK_MODE_IDX0          DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL
#define DRV_SPI_INPUT_PHASE_IDX0         SPI_INPUT_SAMPLING_PHASE_AT_END
#define DRV_SPI_TX_INT_SOURCE_IDX0       INT_SOURCE_SPI_1_TRANSMIT
```

```
#define DRV_SPI_RX_INT_SOURCE_IDX0      INT_SOURCE_SPI_1_RECEIVE
#define DRV_SPI_ERROR_INT_SOURCE_IDX0   INT_SOURCE_SPI_1_ERROR
#define DRV_SPI_INT_VECTOR_IDX0         INT_VECTOR_SPI1
#define DRV_SPI_INT_PRIORITY_IDX0       INT_PRIORITY_LEVEL1
#define DRV_SPI_INT_SUB_PRIORITY_IDX0   INT_SUBPRIORITY_LEVEL0
#define DRV_SPI_QUEUE_SIZE_IDX0         30
#define DRV_SPI_RESERVED_JOB_IDX0       1
#define DRV_SPI_TX_DMA_CHANNEL_IDX0     DMA_CHANNEL_1
#define DRV_SPI_TX_DMA_THRESHOLD_IDX0   16
#define DRV_SPI_RX_DMA_CHANNEL_IDX0     DMA_CHANNEL_0
#define DRV_SPI_RX_DMA_THRESHOLD_IDX0   16 Driver Library
```

Configuring the Library

Macros

	Name	Description
	DRV_ENC28J60_CLIENT_INSTANCES	Selects the maximum number of clients.
	DRV_ENC28J60_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.

Description

The configuration of the ENC28J60 Driver Library is based on the file `sys_config.h`.

This header file contains the configuration selection for the ENC28J60 Driver Library. Based on the selections made, the ENC28J60 Driver Library may support the selected features. These configuration settings will apply to all instances of the ENC28J60 Driver Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_ENC28J60_CLIENT_INSTANCES Macro

Selects the maximum number of clients.

File

drv_enc28j60_config_template.h

C

```
#define DRV_ENC28J60_CLIENT_INSTANCES 1
```

Description

enc28j60 maximum number of clients

This definition selects the maximum number of clients that the enc28j60 driver can support at run-time.

Remarks

Mandatory definition.

DRV_ENC28J60_INSTANCES_NUMBER Macro

Selects the maximum number of hardware instances that can be supported by the dynamic driver.

File

drv_enc28j60_config_template.h

C

```
#define DRV_ENC28J60_INSTANCES_NUMBER 1
```

Description

enc28j60 hardware instance configuration

This definition selects the maximum number of hardware instances that can be supported by the dynamic driver.

Remarks

Mandatory definition.

Building the Library

This section lists the files that are available in the ENC28J60 Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/enc28j60`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source Folder Name	Description
/drv_enc28j60.h	This file provides the interface definitions of the ENC28J60 Driver.

Required File(s)

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

All of the required files listed in the following table are automatically loaded into the MPLAB X IDE project by the MHC.

Source Folder Name	Description
/src/dynamic/drv_drv_enc28j60_api.c	This file contains the API function implementations.
/src/dynamic/drv_enc28j60_main_state.c	This file contains the main state machine functions.
/src/dynamic/drv_enc28j60_utils.c	This file contains functions that are used throughout the driver.
/src/dynamic/bus/spi/drv_enc28j60_spi_bus.c	This file contains the functions to interface with the SPI bus.
/src/dynamic/closed_state/drv_enc28j60_closed_state.c	This file contains the functions for handling the driver closed state.
/src/dynamic/initialization_state/drv_enc28j60_configure_state.c	This file contains the functions for configuring the ENC hardware.
/src/dynamic/initialization_state/drv_enc28j60_detect_state.c	This file contains the functions for detecting the ENC hardware.
/src/dynamic/initialization_state/drv_enc28j60_initialization_state.c	This file contains the functions for the initialization state machine.
/src/dynamic/initialization_state/drv_enc28j60_reset_state.c	This file contains the functions for resetting the ENC hardware.
/src/dynamic/packet/drv_enc28j60_rx_packet.c	This file contains the functions for receiving a packet from the ENC hardware.
/src/dynamic/packet/drv_enc28j60_tx_packet.c	This file contains the functions for sending a packet to the ENC hardware.
/src/dynamic/running_state/drv_enc28j60_change_duplex_state.c	This file contains the functions for configuring the duplex mode of the ENC hardware.
/src/dynamic/running_state/drv_enc28j60_check_int_state.c	This file contains the functions for checking and processing the ENC hardware interrupts.
/src/dynamic/running_state/drv_enc28j60_check_status_state.c	This file contains the functions for checking the status of the ENC hardware.
/src/dynamic/running_state/drv_enc28j60_check_tx_status_state.c	This file contains the functions for checking the status of a transmitted packet.
/src/dynamic/running_state/drv_enc28j60_running_state.c	This file contains the functions for managing the running state machine.
/src/dynamic/running_state/drv_enc28j60_reset_rx_state.c	This file contains the functions for managing the RX state machine reset requirement during run-time.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source Folder Name	Description
N/A	No optional files exist for this library.

Module Dependencies

The ENC28J60 Driver Library depends on the following modules:

- SPI Driver Library
- TCP/IP Stack Library
- TCP/IP Stack MAC Driver Module

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_ENC28J60_Deinitialize	Deinitializes the ENC28J60 Driver Instance. **Implementation:** Dynamic
	DRV_ENC28J60_Initialize	Initializes the ENC28J60 Driver Instance, with the configuration data. **Implementation:** Dynamic
	DRV_ENC28J60_Process	Additional processing that happens outside the tasks function. **Implementation:** Dynamic
	DRV_ENC28J60_Reinitialize	Reinitializes the instance of the ENC28J60 driver. **Implementation:** Dynamic
	DRV_ENC28J60_SetMacCtrlInfo	This function sets the MAC control information for the driver. **Implementation:** Dynamic
	DRV_ENC28J60_StackInitialize	This function initializes the driver with a TCPIP_MAC_INIT object. **Implementation:** Dynamic
	DRV_ENC28J60_Tasks	Main task function for the driver. **Implementation:** Dynamic

b) Client Level Functions

	Name	Description
	DRV_ENC28J60_Close	Closes a client handle to the driver. **Implementation:** Dynamic
	DRV_ENC28J60_ConfigGet	Gets the current configuration. **Implementation:** Dynamic
	DRV_ENC28J60_LinkCheck	This function returns the status of the link. **Implementation:** Dynamic
	DRV_ENC28J60_Open	This function is called by the client to open a handle to a driver instance. **Implementation:** Dynamic
	DRV_ENC28J60_ParametersGet	Get the parameters of the device. **Implementation:** Dynamic
	DRV_ENC28J60_PowerMode	This function sets the power mode of the device. **Implementation:** Dynamic
	DRV_ENC28J60_RegisterStatisticsGet	Get the register statistics. **Implementation:** Dynamic
	DRV_ENC28J60_StatisticsGet	Retrieve the devices statistics. **Implementation:** Dynamic
	DRV_ENC28J60_Status	Gets the current status of the driver. **Implementation:** Dynamic

c) Receive Functions

	Name	Description
	DRV_ENC28J60_PacketRx	Receive a packet from the driver. **Implementation:** Dynamic

	DRV_ENC28J60_RxFilterHashTableEntrySet	This function adds an entry to the hash table. **Implementation:** Dynamic

d) Transmit Functions

	Name	Description
	DRV_ENC28J60_PacketTx	This function queues a packet for transmission. **Implementation:** Dynamic

e) Event Functions

	Name	Description
	DRV_ENC28J60_EventAcknowledge	Acknowledges an event. **Implementation:** Dynamic
	DRV_ENC28J60_EventMaskSet	Sets the event mask. **Implementation:** Dynamic
	DRV_ENC28J60_EventPendingGet	Gets the current events. **Implementation:** Dynamic

f) Data Types and Constants

	Name	Description
	_DRV_ENC28J60_Configuration	Defines the data required to initialize or reinitialize the ENC28J60 Driver.
	DRV_ENC28J60_Configuration	Defines the data required to initialize or reinitialize the ENC28J60 Driver.
	DRV_ENC28J60_MDIX_TYPE	Defines the enumeration for controlling the MDIX select.
	DRV_ENC28J60_MACObject	ENC28J60 External MAC Virtualization Table

Description

This section describes the Application Programming Interface (API) functions of the ENC28J60 Driver Library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_ENC28J60_Deinitialize Function

Deinitializes the ENC28J60 Driver Instance.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
void DRV_ENC28J60_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

ENC28J60 Deinitialization

This function deallocates any resources allocated by the initialization function.

Preconditions

The driver had to be successfully initialized with DRV_ENC28J60_Initialize.

Parameters

Parameters	Description
Object	the valid object returned from DRV_ENC28J60_Initialize

DRV_ENC28J60_Initialize Function

Initializes the ENC28J60 Driver Instance, with the configuration data.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
SYS_MODULE_OBJ DRV_ENC28J60_Initialize(SYS_MODULE_INDEX index, SYS_MODULE_INIT * init);
```

Returns

- Valid handle to the driver instance - If successful
- SYS_MODULE_OBJ_INVALID - If unsuccessful

Description

ENC28J60 Initialization

This function initializes the ENC28J60 Driver with configuration data passed into it by either the system_init function or by the DRV_ENC28J60_StackInitialize function. Calling this function alone is not enough to initialize the driver, DRV_ENC28J60_SetMacCtrlInfo must be called with valid data before the driver is ready to be opened.

Preconditions

None.

Parameters

Parameters	Description
index	This is the index of the driver instance to be initialized. The definition DRV_ENC28J60_NUM_DRV_INSTANCES controls how many instances are available.
init	This is a pointer to a DRV_ENC28J60_CONFIG structure.

DRV_ENC28J60_Process Function

Additional processing that happens outside the tasks function.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
TCPIP_MAC_RES DRV_ENC28J60_Process(DRV_HANDLE hMac);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OP_ERR - if the hMac is valid

Description

ENC28J60 Process

This function does additional processing that is not done inside the tasks function.

Remarks

This function does nothing in the first release.

Preconditions

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

DRV_ENC28J60_Reinitialize Function

Reinitializes the instance of the ENC28J60 driver.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
void DRV_ENC28J60_Reinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
```

Returns

None

Description

ENC28J60 Reinitialization

This function will deinitialize and initialize the driver instance. As with DRV_ENC28J60_Initialize DRV_ENC28J60_SetMacCtrlInfo must be called for the driver to be useful.

Remarks

This function is not planned to be implemented for the first release.

Preconditions

The driver had to be successfully initialized with DRV_ENC28J60_Initialize.

DRV_ENC28J60_SetMacCtrlInfo Function

This function sets the MAC control information for the driver.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
void DRV_ENC28J60_SetMacCtrlInfo(SYS_MODULE_OBJ object, TCPIP_MAC_MODULE_CTRL * init);
```

Returns

None.

Description

ENC28J60 Set MAC Control Information

This function is used to pass in the TCPIP_MAC_CONTROL_INIT information that is used for allocation and deallocation of memory, event signaling, etc. This function is needed to be called so that the driver can enter initialization state when the tasks function is called.

Preconditions

The driver had to be successfully initialized with ENC28J60_Initialize.

DRV_ENC28J60_StackInitialize Function

This function initializes the driver with a TCPIP_MAC_INIT object.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
SYS_MODULE_OBJ DRV_ENC28J60_StackInitialize(SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

Returns a valid handle to the driver instance - If successful SYS_MODULE_OBJ_INVALID - If unsuccessful

Description

ENC28J60 Stack Initialization

This function is used by the TCP/IP stack to fully initialize the driver with both the ENC28J60 specific configuration and the MAC control information. With this function there is no need to call DRV_ENC28J60_SetMacCtrlInfo.

Preconditions

None.

Parameters

Parameters	Description
index	This is the index of the driver instance to be initialized. The definition DRV_ENC28J60_NUM_DRV_INSTANCES controls how many instances are available.
init	This is a pointer to a TCPIP_MAC_INIT structure.

DRV_ENC28J60_Tasks Function

Main task function for the driver.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
void DRV_ENC28J60_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

ENC28J60 Tasks

This function will execute the main state machine for the ENC28J60 driver.

Preconditions

The driver had to be successfully initialized with DRV_ENC28J60_Initialize.

Parameters

Parameters	Description
object	The object valid passed back to DRV_ENC28J60_Initialize

b) Client Level Functions

DRV_ENC28J60_Close Function

Closes a client handle to the driver.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
void DRV_ENC28J60_Close(DRV_HANDLE handle);
```

Returns

None.

Description

ENC28J60 Close

This function closes a handle to the driver. If it is the last client open, the driver will send an RX Disable command to the ENC hardware and move

to the closed state.

Preconditions

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
handle	The successfully opened handle

DRV_ENC28J60_ConfigGet Function

Gets the current configuration.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
size_t DRV_ENC28J60_ConfigGet(DRV_HANDLE hMac, void* configBuff, size_t buffSize, size_t* pConfigSize);
```

Returns

Number of bytes copied to the buffer

Description

ENC28J60 Get Configuration

Gets the current configuration.

Remarks

This function does nothing in the first release.

Preconditions

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
configBuff	location to copy the configuration too
buffSize	buffer size
pConfigSize	configuration size needed

DRV_ENC28J60_LinkCheck Function

This function returns the status of the link.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
bool DRV_ENC28J60_LinkCheck(DRV_HANDLE hMac);
```

Returns

- true - if the link is active
- false - all other times

Description

ENC28J60 Link Check

This function checks the status of the link and returns it to the caller.

Preconditions

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

DRV_ENC28J60_Open Function

This function is called by the client to open a handle to a driver instance.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
DRV_HANDLE DRV_ENC28J60_Open(SYS_MODULE_INDEX index, DRV_IO_INTENT intent);
```

Returns

Returns a valid handle - If successful INVALID_HANDLE - If unsuccessful

Description

ENC28J60 Open

The client will call this function to open a handle to the driver. When the first instance is opened than the driver will send the RX enabled command to the ENC hardware.

Preconditions

The driver had to be successfully initialized with DRV_ENC28J60_Initialize.

Parameters

Parameters	Description
index	This is the index of the driver instance to be initialized. The definition DRV_ENC28J60_NUM_DRV_INSTANCES controls how many instances are available.
intent	The intent to use when opening the driver. Only exclusive is supported

DRV_ENC28J60_ParametersGet Function

Get the parameters of the device.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
TCPIP_MAC_RES DRV_ENC28J60_ParametersGet(DRV_HANDLE hMac, TCPIP_MAC_PARAMETERS* pMacParams);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OK - if the hMac is valid

Description

ENC28J60 Get Parameters

Get the parameters of the device, which includes that it is an Ethernet device and what it's MAC address is. Users of the ENC28J60 must generate a unique MAC address for each controller used.

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

pMacParams	pointer to put the parameters

DRV_ENC28J60_PowerMode Function

This function sets the power mode of the device.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
bool DRV_ENC28J60_PowerMode(DRV_HANDLE hMac, TCPIP_MAC_POWER_MODE pwrMode);
```

Returns

- false - This functionality is not supported in this version of the driver

Description

ENC28J60 Power Mode

This function sets the power mode of the ENC28J60.

Remarks

This functionality is not implemented in the first release.

Preconditions

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
pwrMode	the power mode to set

DRV_ENC28J60_RegisterStatisticsGet Function

Get the register statistics.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
TCPIP_MAC_RES DRV_ENC28J60_RegisterStatisticsGet(DRV_HANDLE hMac, TCPIP_MAC_STATISTICS_REG_ENTRY*
pRegEntries, int nEntries, int* pHwEntries);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OP_ERR - if the hMac is valid

Description

ENC28J60 Get Register Statistics

Get the device specific statistics.

Remarks

Statistics are not planned for the first release

Preconditions

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

pRegEntries	
nEntries	
pHwEntries	

DRV_ENC28J60_StatisticsGet Function

Retrieve the devices statistics.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
TCPIP_MAC_RES DRV_ENC28J60_StatisticsGet(DRV_HANDLE hMac, TCPIP_MAC_RX_STATISTICS* pRxStatistics,
TCPIP_MAC_TX_STATISTICS* pTxStatistics);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OP_ERR - if the hMac is valid

Description

ENC28J60 Get Statistics

Get the current statistics stored in the driver.

Remarks

Statistics are not planned for the first release.

Preconditions

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

DRV_ENC28J60_Status Function

Gets the current status of the driver.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
SYS_STATUS DRV_ENC28J60_Status(SYS_MODULE_OBJ obect);
```

Returns

- SYS_STATUS_ERROR - if an invalid handle has been passed in
- SYS_STATUS_UNINITIALIZED - if the driver has not completed initialization
- SYS_STATUS_BUSY - if the driver is closing and moving to the closed state
- SYS_STATUS_READY - if the driver is ready for client commands

Description

ENC28J60 Status

This function will get the status of the driver instance.

Preconditions

The driver had to be successfully initialized with DRV_ENC28J60_Initialize().

Parameters

Parameters	Description
object	The object valid passed back to DRV_ENC28J60_Initialize()

c) Receive Functions

DRV_ENC28J60_PacketRx Function

Receive a packet from the driver.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
TCPIP_MAC_PACKET* DRV_ENC28J60_PacketRx(DRV_HANDLE hMac, TCPIP_MAC_RES* pRes, const
TCPIP_MAC_PACKET_RX_STAT** ppPktStat);
```

Returns

- Pointer to a valid packet - if successful
- NULL - if unsuccessful

Description

ENC28J60 Receive Packet

This function retrieves a packet from the driver. The packet needs to be acknowledged with the linked acknowledge function so it can be reused.

Remarks

ppPktStat is ignored in the first release.

Preconditions

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
pRes	the result of the operation
ppPktStat	pointer to the receive statistics

DRV_ENC28J60_RxFilterHashTableEntrySet Function

This function adds an entry to the hash table.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
TCPIP_MAC_RES DRV_ENC28J60_RxFilterHashTableEntrySet(DRV_HANDLE hMac, const TCPIP_MAC_ADDR* DestMACAddr);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OP_ERR - if the hMac is valid

Description

ENC28J60 Receive Filter Hash Table Entry Set

This function adds to the MAC's hash table for hash table matching.

Remarks

This functionality is not implemented in the first release.

Preconditions

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
DestMACAddr	MAC address to add to the hash table

d) Transmit Functions

DRV_ENC28J60_PacketTx Function

This function queues a packet for transmission.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
TCPIP_MAC_RES DRV_ENC28J60_PacketTx(DRV_HANDLE hMac, TCPIP_MAC_PACKET * ptrPacket);
```

Returns

- TCPIP_MAC_RES_OP_ERR - if the client handle is invalid
- TCPIP_MAC_RES_IS_BUSY - if the driver is not in the run state
- TCPIP_MAC_RES_QUEUE_TX_FULL - if there are no free descriptors
- TCPIP_MAC_RES_OK - on successful queuing of the packet

Description

ENC28J60 Packet Transmit

This function will take a packet and add it to the queue for transmission. When the packet has finished transmitting the driver will call the packets acknowledge function. When that acknowledge function is complete the driver will forget about the packet.

Preconditions

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
ptrPacket	pointer to the packet

e) Event Functions

DRV_ENC28J60_EventAcknowledge Function

Acknowledges an event.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
bool DRV_ENC28J60_EventAcknowledge(DRV_HANDLE hMac, TCPIP_MAC_EVENT macEvents);
```

Returns

- true - if successful
- false - if not successful

Description

ENC28J60 Acknowledge Event

This function acknowledges an event.

Preconditions

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
macEvents	the events to acknowledge

DRV_ENC28J60_EventMaskSet Function

Sets the event mask.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
bool DRV_ENC28J60_EventMaskSet(DRV_HANDLE hMac, TCPIP_MAC_EVENT macEvents, bool enable);
```

Returns

- true - if the mask could be set
- false - if the mast could not be set

Description

ENC28J60 Set Event Mask

Sets the event mask to what is passed in.

Preconditions

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
macEvents	the mask to enable or disable
enable	to enable or disable events

DRV_ENC28J60_EventPendingGet Function

Gets the current events.

Implementation: Dynamic

File

drv_enc28j60.h

C

```
TCPIP_MAC_EVENT DRV_ENC28J60_EventPendingGet(DRV_HANDLE hMac);
```

Returns

- TCPIP_MAC_EV_NONE - Returned on an error
- List of events - Returned on event other than an error

Description

ENC28J60 Get Events

This function gets the current events.

Preconditions

The client had to be successfully opened with DRV_ENC28J60_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

f) Data Types and Constants

DRV_ENC28J60_Configuration Structure

Defines the data required to initialize or reinitialize the ENC28J60 Driver.

File

drv_enc28j60.h

C

```
typedef struct _DRV_ENC28J60_Configuration {
  uint16_t txDescriptors;
  uint16_t rxDescriptors;
  uint16_t rxDescBufferSize;
  SYS_MODULE_INDEX spiDrvIndex;
  uint32_t spiBps;
  uint16_t rxBufferSize;
  uint16_t maxFrameSize;
  PORTS_MODULE_ID spiSSPortModule;
  PORTS_CHANNEL spiSSPortChannel;
  PORTS_BIT_POS spiSSPortPin;
  bool intEnable;
  PORTS_MODULE_ID intPortModule;
  PORTS_CHANNEL intPortChannel;
  PORTS_BIT_POS intPortPin;
  DRV_ENC28J60_MDIX_TYPE mdixControl;
  PORTS_MODULE_ID mdixPortModule;
  PORTS_CHANNEL mdixPortChannel;
  PORTS_BIT_POS mdixPortPin;
} DRV_ENC28J60_Configuration;
```

Members

Members	Description
uint16_t txDescriptors;	Number of TX Descriptors to Allocate
uint16_t rxDescriptors;	Number of RX Descriptors to Allocate
uint16_t rxDescBufferSize;	Size of the buffer each RX Descriptor will use. Make sure its not smaller than maxFrameSize
SYS_MODULE_INDEX spiDrvIndex;	Index of the SPI driver to use
uint32_t spiBps;	Bus speed to use for the SPI interface. Section 1.0 of the ENC28J60 data sheets says the maximum is 20000000 Hz. It is not recommended to go above this value.
uint16_t rxBufferSize;	The ENC28J60 hardware has a 8 k dram. rxBufferSize defines how much of that memory is used by the rxBuffer
uint16_t maxFrameSize;	The maximum frame size to be supported by the hardware. 1536 is the default
PORTS_MODULE_ID spiSSPortModule;	Port Module of the GPIO pin hooked up to the CS/SS pin of the ENC28J60
PORTS_CHANNEL spiSSPortChannel;	Port Channel of the GPIO pin hooked up to the CS/SS pin of the ENC28J60
PORTS_BIT_POS spiSSPortPin;	Pin position of the GPIO pin hooked up to the CS/SS pin of the ENC28J60
bool intEnable;	Use Interrupts or not.
PORTS_MODULE_ID intPortModule;	Port Module of the GPIO pin hooked up to the INT pin of the ENC28J60
PORTS_CHANNEL intPortChannel;	Port Channel of the GPIO pin hooked up to the INT pin of the ENC28J60
PORTS_BIT_POS intPortPin;	Pin Position of the GPIO pin hooked up to the INT pin of the ENC28J60

DRV_ENC28J60_MDIX_TYPE mdixControl;	To select the control type of the MDIX. This is only needed for hooking up to switches that don't have auto-mdix.
PORTS_MODULE_ID mdixPortModule;	Port Module of the GPIO pin hooked up to the MDIX select pin
PORTS_CHANNEL mdixPortChannel;	Port Channel of the GPIO pin hooked up to the MDIX select pin
PORTS_BIT_POS mdixPortPin;	Pin Position of the GPIO pin hooked up to the MDIX select pin

Description

ENC28J60 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the ENC28J60 driver. If the driver is built statically, the members of this data structure are statically over-ridden by static override definitions in the system_config.h file.

Remarks

None.

DRV_ENC28J60_MDIX_TYPE Enumeration

Defines the enumeration for controlling the MDIX select.

File

drv_enc28j60.h

C

```
typedef enum {
  DRV_ENC28J60_NO_CONTROL = 0,
  DRV_ENC28J60_NORMAL,
  DRV_ENC28J60_REVERSE = 0
} DRV_ENC28J60_MDIX_TYPE;
```

Members

Members	Description
DRV_ENC28J60_NO_CONTROL = 0	No Control
DRV_ENC28J60_NORMAL	Normal MDIX
DRV_ENC28J60_REVERSE = 0	Reverse MDIX

Description

ENC28J60 Driver MDIX Control type

This type defines the enumeration for controlling the MDIX select.

Remarks

None.

DRV_ENC28J60_MACObject Variable

File

drv_enc28j60.h

C

```
const TCPIP_MAC_OBJECT DRV_ENC28J60_MACObject;
```

Description

ENC28J60 External MAC Virtualization Table

Files

Files

Name	Description
drv_enc28j60.h	ENC28J60 Driver interface definition.
drv_enc28j60_config_template.h	enc28j60 Driver configuration definitions template.

Description

drv_enc28j60.h

ENC28J60 Driver interface definition.

Enumerations

	Name	Description
	DRV_ENC28J60_MDIX_TYPE	Defines the enumeration for controlling the MDIX select.

Functions

	Name	Description
	DRV_ENC28J60_Close	Closes a client handle to the driver. **Implementation:** Dynamic
	DRV_ENC28J60_ConfigGet	Gets the current configuration. **Implementation:** Dynamic
	DRV_ENC28J60_Deinitialize	Deinitializes the ENC28J60 Driver Instance. **Implementation:** Dynamic
	DRV_ENC28J60_EventAcknowledge	Acknowledges an event. **Implementation:** Dynamic
	DRV_ENC28J60_EventMaskSet	Sets the event mask. **Implementation:** Dynamic
	DRV_ENC28J60_EventPendingGet	Gets the current events. **Implementation:** Dynamic
	DRV_ENC28J60_Initialize	Initializes the ENC28J60 Driver Instance, with the configuration data. **Implementation:** Dynamic
	DRV_ENC28J60_LinkCheck	This function returns the status of the link. **Implementation:** Dynamic
	DRV_ENC28J60_Open	This function is called by the client to open a handle to a driver instance. **Implementation:** Dynamic
	DRV_ENC28J60_PacketRx	Receive a packet from the driver. **Implementation:** Dynamic
	DRV_ENC28J60_PacketTx	This function queues a packet for transmission. **Implementation:** Dynamic
	DRV_ENC28J60_ParametersGet	Get the parameters of the device. **Implementation:** Dynamic
	DRV_ENC28J60_PowerMode	This function sets the power mode of the device. **Implementation:** Dynamic
	DRV_ENC28J60_Process	Additional processing that happens outside the tasks function. **Implementation:** Dynamic
	DRV_ENC28J60_RegisterStatisticsGet	Get the register statistics. **Implementation:** Dynamic
	DRV_ENC28J60_Reinitialize	Reinitializes the instance of the ENC28J60 driver. **Implementation:** Dynamic
	DRV_ENC28J60_RxFilterHashTableEntrySet	This function adds an entry to the hash table. **Implementation:** Dynamic
	DRV_ENC28J60_SetMacCtrlInfo	This function sets the MAC control information for the driver. **Implementation:** Dynamic
	DRV_ENC28J60_StackInitialize	This function initializes the driver with a TCPIP_MAC_INIT object. **Implementation:** Dynamic
	DRV_ENC28J60_StatisticsGet	Retrieve the devices statistics. **Implementation:** Dynamic
	DRV_ENC28J60_Status	Gets the current status of the driver. **Implementation:** Dynamic
	DRV_ENC28J60_Tasks	Main task function for the driver. **Implementation:** Dynamic

Structures

	Name	Description
	_DRV_ENC28J60_Configuration	Defines the data required to initialize or reinitialize the ENC28J60 Driver.
	DRV_ENC28J60_Configuration	Defines the data required to initialize or reinitialize the ENC28J60 Driver.

Variables

	Name	Description
	DRV_ENC28J60_MACObject	ENC28J60 External MAC Virtualization Table

Description

ENC28J60 Driver Public Interface

This file defines the interface definition for the ENC28J60 Driver.

File Name

drv_enc28j60.h

Company

Microchip Technology Inc.

drv_enc28j60_config_template.h

enc28j60 Driver configuration definitions template.

Macros

	Name	Description
	DRV_ENC28J60_CLIENT_INSTANCES	Selects the maximum number of clients.
	DRV_ENC28J60_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.

Description

enc28j60 Driver Configuration Definitions for the Template Version

These definitions statically define the driver's mode of operation.

File Name

drv_enc28j60_config_template.h

Company

Microchip Technology Inc.

ENCx24J600 Driver Library Help

This section provides information on the ENCx24J600 Driver Library.

Introduction

This library provides a driver-level abstraction of the ENCx24J600 Ethernet MAC that can be connected to the PIC32. The driver implements the virtual MAC driver model that the MPLAB Harmony TCP/IP Stack requires. Please see the TCP/IP Stack Library MAC Driver Module for details.

The "Host-To-Network"_layer of a TCP/IP stack organization covers the Data Link and Physical Layers of the standard OSI stack. The Ethernet Controller provides the Data Link or Media Access Control Layer, in addition to other functions discussed in this section.

Description

The ENCx24J600 External MAC is an external module to the PIC32 that is connected through a SPI or PSP interface. This driver interfaces with the SPI driver to communicate with the external device to implement a complete Ethernet node in a system.

The following are some of the key features of this module:

- Supports 10/100 Ethernet
  - Full-Duplex and Half-Duplex operation
  - Broadcast, Multicast and Unicast packets
  - Manual and automatic flow control
  - Supports Auto-MDIX

- Fully configurable interrupts
- Configurable receive packet filtering using:
  - 64-bit Hash Table
  - 64-byte Pattern Match
  - Magic Packet™ Filtering
  - Runt Packet Detection and Filtering
- Supports Packet Payload Checksum calculation
- CRC Check
- Supports SPI interface

Using the Library

This topic describes the basic architecture and functionality of the Ethernet MAC driver and is meant for advanced users or TCP/IP stack driver developers.

Description

The user of this driver is the MPLAB Harmony TCP/IP stack. This Ethernet driver is not intended as a system-wide driver that the application or other system modules may use. It is intended for the sole use of the MPLAB Harmony TCP/IP stack and implements the virtual MAC model required by the stack.

Interface Header File: `drv_encx24j600.h`

The interface to the ENCx24J600 Driver Library is defined in the `drv_encx24j600.h` header file. Any C language source (`.c`) file that uses the ENCx24J600 Driver Library should include `drv_encx24j600.h`.

Library File: The ENCx24J600 Driver Library archive (`.a`) file is installed with MPLAB Harmony.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

The ENCx24J600 Driver Library provides the low-level abstraction of the communications protocol to communicate to the ENCx24J600 external MAC though the SPI peripheral on the Microchip family of microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the ENCx24J600 Driver Library interface.

Description

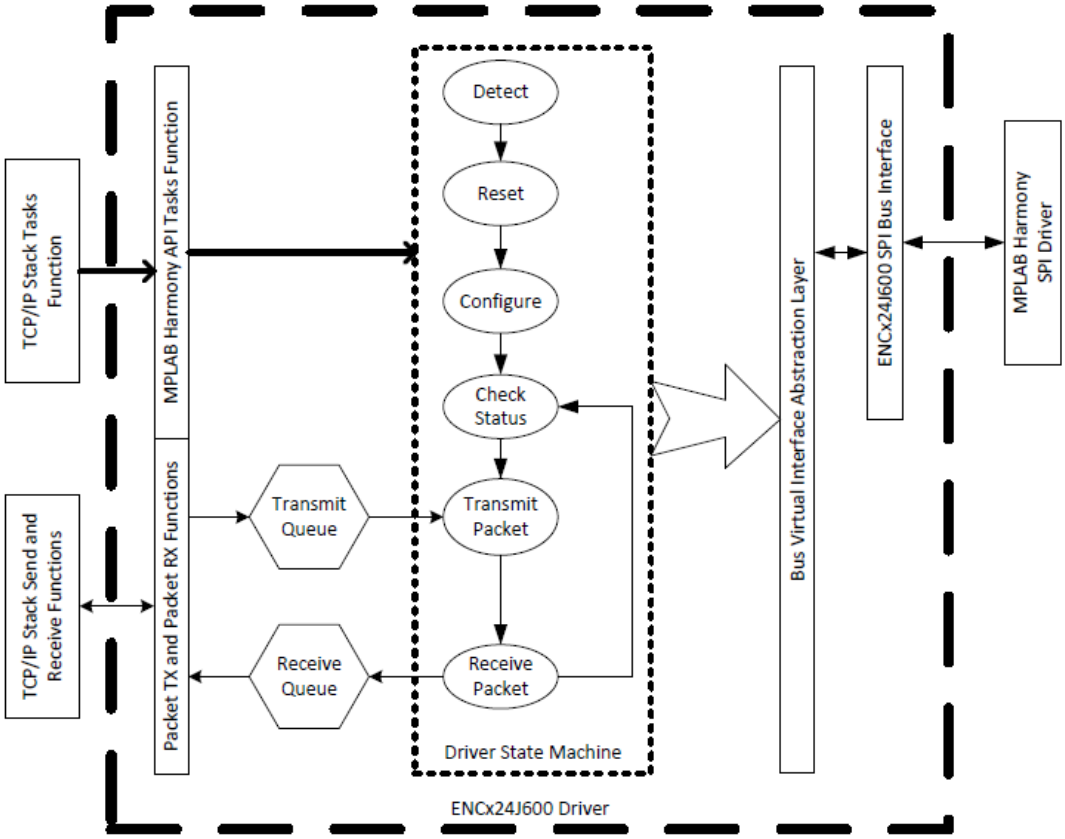
The ENCx24J600 Driver library has several different layers to it, as illustrated in the following figure. The interface layer has two main sections that are used the most often: The Tasks function, and the TCP/IP Send and Receive functions.

The Tasks function manages the internal state machine which detects, resets, and then configures the ENCx24J600 External MAC. It also handles the monitoring of the hardware status, sending and receiving packets.

The TCP/IP Send and Receive functions interact with the RAM-based queue of packets that are queued to send and packets that have been queued waiting for pick-up by the stack.

The main state machine does not interface directly to the SPI bus, but instead, interfaces to a virtual bus abstraction layer that allows for the replacement of the specific underlying bus implementation.

Abstraction Model



Library Overview

Refer to the section Driver Overview for how the driver operates in a system.

The library interface routines are divided into various sub-sections, each sub-section addresses one of the blocks or the overall operation of the ENCx24J600 Driver Library.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Data Transfer Functions	Provides data transfer functions available in the configuration.
Status Functions	Provides status functions.
Miscellaneous Functions	Provides miscellaneous driver functions.

How the Library Works

The library provides interfaces to support the TCP/IP virtual MAC interface.

Configuring the SPI Driver

This section describes the configuration settings for the ENCx24J600 Driver Library.

Description

Configuration

The ENC hardware requires a specific configuration of the SPI driver to work correctly. Inside the MHC SPI Driver configuration be sure to select:

- SPI clock rate of 14000000 or less. With a PB clock of 80 MHz, 13333333 is the clock rate.
- Clock mode of DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL
- Input phase of SPI_INPUT_SAMPLING_PHASE_AT_END

Recommended Settings

- Interrupt Driver mode
- Enhanced Buffer mode
- DMA mode enabled:
  - DMA block transfer size of at least 1600 bytes
  - Size of DMA buffer for dummy data of at least 1600 bytes
  - Ensure when setting up DMA in interrupt mode that the DMA interrupts are a higher priority than the SPI Driver interrupt

Example:

```
/*** SPI Driver Static Allocation Options ***/
#define DRV_SPI_INSTANCES_NUMBER           1
#define DRV_SPI_CLIENTS_NUMBER             1
#define DRV_SPI_ELEMENTS_PER_QUEUE         30

/*** SPI Driver DMA Options ***/
#define DRV_SPI_DMA_TXFER_SIZE             2048
#define DRV_SPI_DMA_DUMMY_BUFFER_SIZE      2048

/* SPI Driver Instance 0 Configuration */
#define DRV_SPI_SPI_ID_IDX0              SPI_ID_1
#define DRV_SPI_TASK_MODE_IDX0           DRV_SPI_TASK_MODE_ISR
#define DRV_SPI_SPI_MODE_IDX0            DRV_SPI_MODE_MASTER
#define DRV_SPI_ALLOW_IDLE_RUN_IDX0      false
#define DRV_SPI_SPI_PROTOCOL_TYPE_IDX0   DRV_SPI_PROTOCOL_TYPE_STANDARD
#define DRV_SPI_SPI_PROTOCOL_TYPE_IDX0   DRV_SPI_PROTOCOL_TYPE_STANDARD
#define DRV_SPI_COMM_WIDTH_IDX0          SPI_COMMUNICATION_WIDTH_8BITS
#define DRV_SPI_SPI_CLOCK_IDX0           CLK_BUS_PERIPHERAL_2
#define DRV_SPI_BAUD_RATE_IDX0           13333333
#define DRV_SPI_BUFFER_TYPE_IDX0         DRV_SPI_BUFFER_TYPE_ENHANCED
#define DRV_SPI_CLOCK_MODE_IDX0          DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL
#define DRV_SPI_INPUT_PHASE_IDX0         SPI_INPUT_SAMPLING_PHASE_AT_END
#define DRV_SPI_TX_INT_SOURCE_IDX0       INT_SOURCE_SPI_1_TRANSMIT
#define DRV_SPI_RX_INT_SOURCE_IDX0       INT_SOURCE_SPI_1_RECEIVE
#define DRV_SPI_ERROR_INT_SOURCE_IDX0    INT_SOURCE_SPI_1_ERROR
#define DRV_SPI_INT_VECTOR_IDX0          INT_VECTOR_SPI1
#define DRV_SPI_INT_PRIORITY_IDX0        INT_PRIORITY_LEVEL1
#define DRV_SPI_INT_SUB_PRIORITY_IDX0    INT_SUBPRIORITY_LEVEL0
#define DRV_SPI_QUEUE_SIZE_IDX0          30
#define DRV_SPI_RESERVED_JOB_IDX0        1
#define DRV_SPI_TX_DMA_CHANNEL_IDX0      DMA_CHANNEL_1
#define DRV_SPI_TX_DMA_THRESHOLD_IDX0    16
#define DRV_SPI_RX_DMA_CHANNEL_IDX0      DMA_CHANNEL_0
#define DRV_SPI_RX_DMA_THRESHOLD_IDX0    16 Driver Library
```

Configuring the Library

The configuration of the ENCx24J600 Driver Library is based on the file `sys_config.h`.

This header file contains the configuration selection for the ENCX24J600 Driver Library. Based on the selections made, the ENCx24J600 Driver Library may support the selected features. These configuration settings will apply to all instances of the ENCx24J600 Driver Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Building the Library

This section lists the files that are available in the ENCx24J600 Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/encx24j600`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source Folder Name	Description
/drv_encx24j600.h	This file provides the interface definitions of the ENCx24J600 Driver.

Required File(s)

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

Source Folder Name	Description
/src/dynamic/drv_drv_encx24J600_api.c	This file contains the API function implementations.
/src/dynamic/drv_encx24J600_main_state.c	This file contains the main state machine functions.
/src/dynamic/drv_encx24J600_utils.c	This file contains functions that are used throughout the driver.
/src/dynamic/bus/spi/drv_encx24J600_spi_bus.c	This file contains the functions to interface with the SPI bus.
/src/dynamic/closed_state/drv_encx24J600_closed_state.c	This file contains the functions for handling the driver closed state.
/src/dynamic/initialization_state/drv_encx24J600_configure_state.c	This file contains the functions for configuring the ENC hardware.
/src/dynamic/initialization_state/drv_encx24J600_detect_state.c	This file contains the functions for detecting the ENC hardware.
/src/dynamic/initialization_state/drv_encx24J600_initialization_state.c	This file contains the functions for the initialization state machine.
/src/dynamic/initialization_state/drv_encx24J600_reset_state.c	This file contains the functions for resetting the ENC hardware.
/src/dynamic/packet/drv_encx24J600_rx_packet.c	This file contains the functions for receiving a packet from the ENC hardware.
/src/dynamic/packet/drv_encx24J600_tx_packet.c	This file contains the functions for sending a packet to the ENC hardware.
/src/dynamic/running_state/drv_encx24J600_change_duplex_state.c	This file contains the functions for configuring the duplex mode of the ENC hardware.
/src/dynamic/running_state/drv_encx24J600_check_int_state.c	This file contains the functions for checking and processing the ENC hardware interrupts.
/src/dynamic/running_state/drv_encx24J600_check_status_state.c	This file contains the functions for checking the status of the ENC hardware.
/src/dynamic/running_state/drv_encx24J600_check_tx_status_state.c	This file contains the functions for checking the status of a transmitted packet.
/src/dynamic/running_state/drv_encx24J600_running_state.c	This file contains the functions for managing the running state machine.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source Folder Name	Description
N/A	No optional files exist for this library.

Module Dependencies

The ENCx24J600 Driver Library depends on the following modules:

- SPI Driver Library
- TCP/IP Stack Library
- TCP/IP Stack MAC Driver Module

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_ENCX24J600_Deinitialize	Deinitializes the ENCx24J600 Driver Instance. **Implementation:** Dynamic
	DRV_ENCX24J600_Initialize	Initializes the ENCx24J600 Driver Instance, with the configuration data. **Implementation:** Dynamic
	DRV_ENCX24J600_Reinitialize	Reinitializes the instance of the ENCX24J600 driver. **Implementation:** Dynamic
	DRV_ENCX24J600_Tasks	Main task function for the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_SetMacCtrlInfo	This function sets the MAC control information for the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_StackInitialize	This function initializes the driver with a TCPIP_MAC_INIT object. **Implementation:** Dynamic
	DRV_ENCX24J600_Process	Additional processing that happens outside the tasks function. **Implementation:** Dynamic

b) Client Level Functions

	Name	Description
	DRV_ENCX24J600_Close	Closes a client handle to the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_ConfigGet	Gets the current configuration. **Implementation:** Dynamic
	DRV_ENCX24J600_LinkCheck	This function returns the status of the link. **Implementation:** Dynamic
	DRV_ENCX24J600_Open	This function is called by the client to open a handle to a driver instance. **Implementation:** Dynamic
	DRV_ENCX24J600_ParametersGet	Get the parameters of the device. **Implementation:** Dynamic
	DRV_ENCX24J600_PowerMode	This function sets the power mode of the device. **Implementation:** Dynamic
	DRV_ENCX24J600_RegisterStatisticsGet	Get the register statistics. **Implementation:** Dynamic
	DRV_ENCX24J600_StatisticsGet	Retrieve the devices statistics. **Implementation:** Dynamic
	DRV_ENCX24J600_Status	Gets the current status of the driver. **Implementation:** Dynamic

c) Receive Functions

	Name	Description
	DRV_ENCX24J600_PacketRx	Receive a packet from the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_RxFilterHashTableEntrySet	This function adds an entry to the hash table. **Implementation:** Dynamic

d) Transmit Functions

	Name	Description
	DRV_ENCX24J600_PacketTx	This function queues a packet for transmission. **Implementation:** Dynamic

e) Event Functions

	Name	Description
	DRV_ENCX24J600_EventAcknowledge	Acknowledges an event. **Implementation:** Dynamic

	DRV_ENCX24J600_EventMaskSet	Sets the event mask. **Implementation:** Dynamic
	DRV_ENCX24J600_EventPendingGet	Gets the current events. **Implementation:** Dynamic

f) Data Types and Constants

	Name	Description
	_DRV_ENCX24J600_Configuration	Defines the data required to initialize or reinitialize the ENCX24J600 Driver.
	DRV_ENCX24J600_Configuration	Defines the data required to initialize or reinitialize the ENCX24J600 Driver.
	DRV_ENCX24J600_MDIX_TYPE	Defines the enumeration for controlling the MDIX select.

Description

This section describes the Application Programming Interface (API) functions of the ENCx24J600 Driver Library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_ENCX24J600_Deinitialize Function

Deinitializes the ENCx24J600 Driver Instance.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
void DRV_ENCX24J600_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

ENCX24J600 Deinitialization

This function deallocates any resources allocated by the initialization function.

Preconditions

The driver had to be successfully initialized with DRV_ENCX24J600_Initialize.

Parameters

Parameters	Description
Object	the valid object returned from DRV_ENCX24J600_Initialize

DRV_ENCX24J600_Initialize Function

Initializes the ENCx24J600 Driver Instance, with the configuration data.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
SYS_MODULE_OBJ DRV_ENCX24J600_Initialize(SYS_MODULE_INDEX index, SYS_MODULE_INIT * init);
```

Returns

- Valid handle to the driver instance - If successful
- SYS_MODULE_OBJ_INVALID - If unsuccessful

Description

ENCX24J600 Initialization

This function initializes the ENCx24J600 Driver with configuration data passed into it by either the system_init function or by the DRV_ENCX24J600_StackInitialize function. Calling this function alone is not enough to initialize the driver, DRV_ENCX24J600_SetMacCtrlInfo must be called with valid data before the driver is ready to be opened.

Preconditions

None.

Parameters

Parameters	Description
index	This is the index of the driver instance to be initialized. The definition DRV_ENCX24J600_NUM_DRV_INSTANCES controls how many instances are available.
init	This is a pointer to a DRV_ENX24J600_CONFIG structure.

DRV_ENCX24J600_Reinitialize Function

Reinitializes the instance of the ENCX24J600 driver.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
void DRV_ENCX24J600_Reinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
```

Returns

None

Description

ENCX24J600 Reinitialization

This function will deinitialize and initialize the driver instance. As with DRV_ENCX24J600_Initialize DRV_ENCX24J600_SetMacCtrlInfo must be called for the driver to be useful.

Remarks

This function is not planned to be implemented for the first release.

Preconditions

The driver had to be successfully initialized with DRV_ENCX24J600_Initialize.

DRV_ENCX24J600_Tasks Function

Main task function for the driver.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
void DRV_ENCX24J600_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

ENCX24J600 Tasks

This function will execute the main state machine for the ENCX24J600 driver.

Preconditions

The driver had to be successfully initialized with DRV_ENCX24J600_Initialize.

Parameters

Parameters	Description
object	The object valid passed back to DRV_ENCX24J600_Initialize

DRV_ENCX24J600_SetMacCtrlInfo Function

This function sets the MAC control information for the driver.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
void DRV_ENCX24J600_SetMacCtrlInfo(SYS_MODULE_OBJ object, TCPIP_MAC_MODULE_CTRL * init);
```

Returns

None.

Description

ENCX24J600 Set MAC Control Information

This function is used to pass in the TCPIP_MAC_CONTROL_INIT information that is used for allocation and deallocation of memory, event signaling, etc. This function is needed to be called so that the driver can enter initialization state when the tasks function is called.

Preconditions

The driver had to be successfully initialized with DRV_ENCX24J600_Initialize.

DRV_ENCX24J600_StackInitialize Function

This function initializes the driver with a TCPIP_MAC_INIT object.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
SYS_MODULE_OBJ DRV_ENCX24J600_StackInitialize(SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

Returns a valid handle to the driver instance - If successful SYS_MODULE_OBJ_INVALID - If unsuccessful

Description

ENCX24J600 Stack Initialization

This function is used by the TCP/IP stack to fully initialize the driver with both the ENCX24J600 specific configuration and the MAC control information. With this function there is no need to call DRV_ENCX24J600_SetMacCtrlInfo.

Preconditions

None.

Parameters

Parameters	Description
index	This is the index of the driver instance to be initialized. The definition DRV_ENCX24J600_NUM_DRV_INSTANCES controls how many instances are available.
init	This is a pointer to a TCPIP_MAC_INIT structure.

DRV_ENCX24J600_Process Function

Additional processing that happens outside the tasks function.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_RES DRV_ENCX24J600_Process(DRV_HANDLE hMac);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OP_ERR - if the hMac is valid

Description

ENCX24J600 Process

This function does additional processing that is not done inside the tasks function.

Remarks

This function does nothing in the first release.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

b) Client Level Functions

DRV_ENCX24J600_Close Function

Closes a client handle to the driver.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
void DRV_ENCX24J600_Close(DRV_HANDLE handle);
```

Returns

None.

Description

ENCX24J600 Close

This function closes a handle to the driver. If it is the last client open, the driver will send an RX Disable command to the ENC hardware and move to the closed state.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
handle	The successfully opened handle

DRV_ENCX24J600_ConfigGet Function

Gets the current configuration.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
size_t DRV_ENCX24J600_ConfigGet(DRV_HANDLE hMac, void* configBuff, size_t buffSize, size_t* pConfigSize);
```

Returns

Number of bytes copied to the buffer

Description

ENCX24J600 Get Configuration

Gets the current configuration.

Remarks

This function does nothing in the first release.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
configBuff	location to copy the configuration too
buffSize	buffer size
pConfigSize	configuration size needed

DRV_ENCX24J600_LinkCheck Function

This function returns the status of the link.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
bool DRV_ENCX24J600_LinkCheck(DRV_HANDLE hMac);
```

Returns

- true - if the link is active
- false - all other times

Description

ENCX24J600 Link Check

This function checks the status of the link and returns it to the caller.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

DRV_ENCX24J600_Open Function

This function is called by the client to open a handle to a driver instance.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
DRV_HANDLE DRV_ENCX24J600_Open(SYS_MODULE_INDEX index, DRV_IO_INTENT intent);
```

Returns

Returns a valid handle - If successful INVALID_HANDLE - If unsuccessful

Description

ENCX24J600 Open

The client will call this function to open a handle to the driver. When the first instance is opened than the driver will send the RX enabled command to the ENC hardware.

Preconditions

The driver had to be successfully initialized with DRV_ENCX24J600_Initialize.

Parameters

Parameters	Description
index	This is the index of the driver instance to be initialized. The definition DRV_ENCX24J600_NUM_DRV_INSTANCES controls how many instances are available.
intent	The intent to use when opening the driver. Only exclusive is supported

DRV_ENCX24J600_ParametersGet Function

Get the parameters of the device.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_RES DRV_ENCX24J600_ParametersGet(DRV_HANDLE hMac, TCPIP_MAC_PARAMETERS* pMacParams);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OK - if the hMac is valid

Description

ENCX24J600 Get Parameters

Get the parameters of the device, which includes that it is an Ethernet device and what it's MAC address is.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
pMacParams	pointer to put the parameters

DRV_ENCX24J600_PowerMode Function

This function sets the power mode of the device.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
bool DRV_ENCX24J600_PowerMode(DRV_HANDLE hMac, TCPIP_MAC_POWER_MODE pwrMode);
```

Returns

- false - This functionality is not supported in this version of the driver

Description

ENCX24J600 Power Mode

This function sets the power mode of the ENCX24J600.

Remarks

This functionality is not implemented in the first release.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
pwrMode	the power mode to set

DRV_ENCX24J600_RegisterStatisticsGet Function

Get the register statistics.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_RES DRV_ENCX24J600_RegisterStatisticsGet(DRV_HANDLE hMac, TCPIP_MAC_STATISTICS_REG_ENTRY*
pRegEntries, int nEntries, int* pHwEntries);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OP_ERR - if the hMac is valid

Description

ENCX24J600 Get Register Statistics

Get the device specific statistics.

Remarks

Statistics are not planned for the first release

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
pRegEntries	
nEntries	
pHwEntries	

DRV_ENCX24J600_StatisticsGet Function

Retrieve the devices statistics.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_RES DRV_ENCX24J600_StatisticsGet(DRV_HANDLE hMac, TCPIP_MAC_RX_STATISTICS* pRxStatistics,
TCPIP_MAC_TX_STATISTICS* pTxStatistics);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OP_ERR - if the hMac is valid

Description

ENCX24J600 Get Statistics

Get the current statistics stored in the driver.

Remarks

Statistics are not planned for the first release.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

DRV_ENCX24J600_Status Function

Gets the current status of the driver.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
SYS_STATUS DRV_ENCX24J600_Status(SYS_MODULE_OBJ obect);
```

Returns

- SYS_STATUS_ERROR - if an invalid handle has been passed in
- SYS_STATUS_UNINITIALIZED - if the driver has not completed initialization
- SYS_STATUS_BUSY - if the driver is closing and moving to the closed state
- SYS_STATUS_READY - if the driver is ready for client commands

Description

ENCX24J600 Status

This function will get the status of the driver instance.

Preconditions

The driver had to be successfully initialized with DRV_ENCX24J600_Initialize().

Parameters

Parameters	Description
object	The object valid passed back to DRV_ENCX24J600_Initialize()

c) Receive Functions

DRV_ENCX24J600_PacketRx Function

Receive a packet from the driver.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_PACKET* DRV_ENCX24J600_PacketRx(DRV_HANDLE hMac, TCPIP_MAC_RES* pRes, const
TCPIP_MAC_PACKET_RX_STAT** ppPktStat);
```

Returns

- Pointer to a valid packet - if successful
- NULL - if unsuccessful

Description

ENCX24J600 Receive Packet

This function retrieves a packet from the driver. The packet needs to be acknowledged with the linked acknowledge function so it can be reused.

Remarks

ppPktStat is ignored in the first release.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
pRes	the result of the operation
ppPktStat	pointer to the receive statistics

DRV_ENCX24J600_RxFilterHashTableEntrySet Function

This function adds an entry to the hash table.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_RES DRV_ENCX24J600_RxFilterHashTableEntrySet(DRV_HANDLE hMac, const TCPIP_MAC_ADDR* DestMACAddr);
```

Returns

- TCPIP_MAC_RES_TYPE_ERR - if the hMac is invalid
- TCPIP_MAC_RES_OP_ERR - if the hMac is valid

Description

ENCX24J600 Receive Filter Hash Table Entry Set

This function adds to the MAC's hash table for hash table matching.

Remarks

This functionality is not implemented in the first release.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
DestMACAddr	MAC address to add to the hash table

d) Transmit Functions

DRV_ENCX24J600_PacketTx Function

This function queues a packet for transmission.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_RES DRV_ENCX24J600_PacketTx(DRV_HANDLE hMac, TCPIP_MAC_PACKET * ptrPacket);
```

Returns

- TCPIP_MAC_RES_OP_ERR - if the client handle is invalid
- TCPIP_MAC_RES_IS_BUSY - if the driver is not in the run state
- TCPIP_MAC_RES_QUEUE_TX_FULL - if there are no free descriptors
- TCPIP_MAC_RES_OK - on successful queuing of the packet

Description

ENCX24J600 Packet Transmit

This function will take a packet and add it to the queue for transmission. When the packet has finished transmitting the driver will call the packets acknowledge function. When that acknowledge function is complete the driver will forget about the packet.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
ptrPacket	pointer to the packet

e) Event Functions

DRV_ENCX24J600_EventAcknowledge Function

Acknowledges an event.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
bool DRV_ENCX24J600_EventAcknowledge(DRV_HANDLE hMac, TCPIP_MAC_EVENT macEvents);
```

Returns

- true - if successful
- false - if not successful

Description

ENCX24J600 Acknowledge Event

This function acknowledges an event.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
macEvents	the events to acknowledge

DRV_ENCX24J600_EventMaskSet Function

Sets the event mask.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
bool DRV_ENCX24J600_EventMaskSet(DRV_HANDLE hMac, TCPIP_MAC_EVENT macEvents, bool enable);
```

Returns

- true - if the mask could be set
- false - if the mast could not be set

Description

ENCX24J600 Set Event Mask

Sets the event mask to what is passed in.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle
macEvents	the mask to enable or disable
enable	to enable or disable events

DRV_ENCX24J600_EventPendingGet Function

Gets the current events.

Implementation: Dynamic

File

drv_encx24j600.h

C

```
TCPIP_MAC_EVENT DRV_ENCX24J600_EventPendingGet(DRV_HANDLE hMac);
```

Returns

- TCPIP_MAC_EV_NONE - Returned on an error
- List of events - Returned on event other than an error

Description

ENCX24J600 Get Events

This function gets the current events.

Preconditions

The client had to be successfully opened with DRV_ENCX24J600_Open.

Parameters

Parameters	Description
hMac	the successfully opened handle

f) Data Types and Constants

DRV_ENCX24J600_Configuration Structure

Defines the data required to initialize or reinitialize the ENCX24J600 Driver.

File

drv_encx24j600.h

C

```
typedef struct _DRV_ENCX24J600_Configuration {
  uint16_t txDescriptors;
  uint16_t rxDescriptors;
  uint16_t rxDescBufferSize;
  SYS_MODULE_INDEX spiDrvIndex;
  uint32_t spiBps;
  uint16_t rxBufferSize;
  uint16_t maxFrameSize;
  PORTS_MODULE_ID spiSSPortModule;
  PORTS_CHANNEL spiSSPortChannel;
  PORTS_BIT_POS spiSSPortPin;
  bool intEnable;
  PORTS_MODULE_ID intPortModule;
  PORTS_CHANNEL intPortChannel;
  PORTS_BIT_POS intPortPin;
  DRV_ENCX24J600_MDIX_TYPE mdixControl;
  PORTS_MODULE_ID mdixPortModule;
  PORTS_CHANNEL mdixPortChannel;
  PORTS_BIT_POS mdixPortPin;
  TCPIP_ETH_OPEN_FLAGS ethType;
  TCPIP_ETH_OPEN_FLAGS dupMode;
} DRV_ENCX24J600_Configuration;
```

Members

Members	Description
uint16_t txDescriptors;	Number of TX Descriptors to Allocate
uint16_t rxDescriptors;	Number of RX Descriptors to Allocate
uint16_t rxDescBufferSize;	Size of the buffer each RX Descriptor will use. Make sure its not smaller that maxFrameSize
SYS_MODULE_INDEX spiDrvIndex;	Index of the SPI driver to use
uint32_t spiBps;	Bus speed to use for the SPI interface. Section 1.0 of the ENCX24J600 data sheets says the maximum is 14000000 Hz. It is not recommended to go above this value.
uint16_t rxBufferSize;	The ENCX24J600 hardware has a 22 k dram. rxBufferSize defines how much of that memory is used by the rxBuffer
uint16_t maxFrameSize;	The maximum frame size to be supported by the hardware. 1536 is the default
PORTS_MODULE_ID spiSSPortModule;	Port Module of the GPIO pin hooked up to the CS/SS pin of the ENCX24J600
PORTS_CHANNEL spiSSPortChannel;	Port Channel of the GPIO pin hooked up to the CS/SS pin of the ENCX24J600
PORTS_BIT_POS spiSSPortPin;	Pin position of the GPIO pin hooked up to the CS/SS pin of the ENCX24J600
bool intEnable;	Use Interrupts or not.
PORTS_MODULE_ID intPortModule;	Port Module of the GPIO pin hooked up to the INT pin of the ENCX24J600
PORTS_CHANNEL intPortChannel;	Port Channel of the GPIO pin hooked up to the INT pin of the ENCX24J600
PORTS_BIT_POS intPortPin;	Pin Position of the GPIO pin hooked up to the INT pin of the ENCX24J600
DRV_ENCX24J600_MDIX_TYPE mdixControl;	To select the control type of the MDIX. This is only needed for hooking up to switches that don't have auto-mdix.
PORTS_MODULE_ID mdixPortModule;	Port Module of the GPIO pin hooked up to the MDIX select pin
PORTS_CHANNEL mdixPortChannel;	Port Channel of the GPIO pin hooked up to the MDIX select pin
PORTS_BIT_POS mdixPortPin;	Pin Position of the GPIO pin hooked up to the MDIX select pin
TCPIP_ETH_OPEN_FLAGS ethType;	Ethernet type
TCPIP_ETH_OPEN_FLAGS dupMode;	Duplex Mode

Description

ENCX24J600 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the ENCX24J600 driver. If the driver is built statically, the members of this data structure are statically over-ridden by static override definitions in the system_config.h file.

Remarks

None.

DRV_ENCX24J600_MDIX_TYPE Enumeration

Defines the enumeration for controlling the MDIX select.

File

drv_encx24j600.h

C

```
typedef enum {
  DRV_ENCX24J600_NO_CONTROL = 0,
  DRV_ENCX24J600_NORMAL,
  DRV_ENCX24J600_REVERSE = 0
} DRV_ENCX24J600_MDIX_TYPE;
```

Members

Members	Description
DRV_ENCX24J600_NO_CONTROL = 0	No Control
DRV_ENCX24J600_NORMAL	Normal MDIX
DRV_ENCX24J600_REVERSE = 0	Reverse MDIX

Description

ENCX24J600 Driver MDIX Control type

This type defines the enumeration for controlling the MDIX select.

Remarks

None.

Files

Files

Name	Description
drv_encx24j600.h	ENCx24J600 Driver interface definition.

Description

drv_encx24j600.h

ENCx24J600 Driver interface definition.

Enumerations

	Name	Description
	DRV_ENCX24J600_MDIX_TYPE	Defines the enumeration for controlling the MDIX select.

Functions

	Name	Description
	DRV_ENCX24J600_Close	Closes a client handle to the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_ConfigGet	Gets the current configuration. **Implementation:** Dynamic
	DRV_ENCX24J600_Deinitialize	Deinitializes the ENCx24J600 Driver Instance. **Implementation:** Dynamic
	DRV_ENCX24J600_EventAcknowledge	Acknowledges an event. **Implementation:** Dynamic

	DRV_ENCX24J600_EventMaskSet	Sets the event mask. **Implementation:** Dynamic
	DRV_ENCX24J600_EventPendingGet	Gets the current events. **Implementation:** Dynamic
	DRV_ENCX24J600_Initialize	Initializes the ENCx24J600 Driver Instance, with the configuration data. **Implementation:** Dynamic
	DRV_ENCX24J600_LinkCheck	This function returns the status of the link. **Implementation:** Dynamic
	DRV_ENCX24J600_Open	This function is called by the client to open a handle to a driver instance. **Implementation:** Dynamic
	DRV_ENCX24J600_PacketRx	Receive a packet from the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_PacketTx	This function queues a packet for transmission. **Implementation:** Dynamic
	DRV_ENCX24J600_ParametersGet	Get the parameters of the device. **Implementation:** Dynamic
	DRV_ENCX24J600_PowerMode	This function sets the power mode of the device. **Implementation:** Dynamic
	DRV_ENCX24J600_Process	Additional processing that happens outside the tasks function. **Implementation:** Dynamic
	DRV_ENCX24J600_RegisterStatisticsGet	Get the register statistics. **Implementation:** Dynamic
	DRV_ENCX24J600_Reinitialize	Reinitializes the instance of the ENCX24J600 driver. **Implementation:** Dynamic
	DRV_ENCX24J600_RxFilterHashTableEntrySet	This function adds an entry to the hash table. **Implementation:** Dynamic
	DRV_ENCX24J600_SetMacCtrlInfo	This function sets the MAC control information for the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_StackInitialize	This function initializes the driver with a TCPIP_MAC_INIT object. **Implementation:** Dynamic
	DRV_ENCX24J600_StatisticsGet	Retrieve the devices statistics. **Implementation:** Dynamic
	DRV_ENCX24J600_Status	Gets the current status of the driver. **Implementation:** Dynamic
	DRV_ENCX24J600_Tasks	Main task function for the driver. **Implementation:** Dynamic

Structures

	Name	Description
	_DRV_ENCX24J600_Configuration	Defines the data required to initialize or reinitialize the ENCX24J600 Driver.
	DRV_ENCX24J600_Configuration	Defines the data required to initialize or reinitialize the ENCX24J600 Driver.

Description

ENCx24J600 Driver Public Interface

This file defines the interface definition for the ENCx24J600 Driver.

File Name

drv_encx24j600.h

Company

Microchip Technology Inc.

Ethernet MAC Driver Library

This section describes the Ethernet MAC Driver Library.

Introduction

This library provides a driver-level abstraction of the on-chip Ethernet Controller found on many PIC32 devices. The driver implements the virtual MAC driver model that the MPLAB Harmony TCP/IP Stack requires. Please see the TCP/IP Stack Library MAC Driver Module help for details.

The "Host-To-Network" layer of a TCP/IP stack organization covers the Data Link and Physical Layers of the standard OSI stack. The Ethernet Controller provides the Data Link or Media Access Control Layer, in addition to other functions discussed in this section. An external Ethernet "PHY" provides the Physical layer, providing conversion between the digital and analog.

Description

The PIC32 Ethernet Controller is a bus master module that interfaces with an off-chip PHY to implement a complete Ethernet node in a system. The following are some of the key features of this module:

- Supports 10/100 Ethernet
  - Full-Duplex and Half-Duplex operation
  - Broadcast, Multicast and Unicast packets
  - Manual and automatic flow control
  - Supports Auto-MDIX enabled PHYs
  - Reduced Media Independent Interface (RMII) and Media Independent Interface (MII) PHY data interfaces
  - Performance statistics metrics in hardware.
- RAM descriptor based DMA operation for both receive and transmit path
- Fully configurable interrupts
- Configurable receive packet filtering using:
  - 64-bit Hash Table
  - 64-byte Pattern Match
  - Magic Packet™ Filtering
  - Runt Packet Detection and Filtering
- Supports Packet Payload Checksum calculation
- CRC Check

Support for the Serial Management Interface (SMI) (also known as the MIIM interface) is provided by the Ethernet PHY Driver Library.

Using the Library

The user of this driver is the MPLAB Harmony TCP/IP stack. This Ethernet driver is not intended as a system wide driver that the application or other system modules may use. It is intended for the sole use of the MPLAB Harmony TCP/IP stack and implements the virtual MAC model required by the stack.

This topic describes the basic architecture and functionality of the Ethernet MAC driver and is meant for advanced users or TCP/IP stack driver developers.

Interface Header File: `drv_ethmac.h`

The interface to the Ethernet MAC library is defined in the `drv_ethmac.h` header file, which is included by the MPLAB Harmony TCP/IP stack.

Please refer to the What is MPLAB Harmony? section for how the library interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the Ethernet MAC Driver Library on Microchip's microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

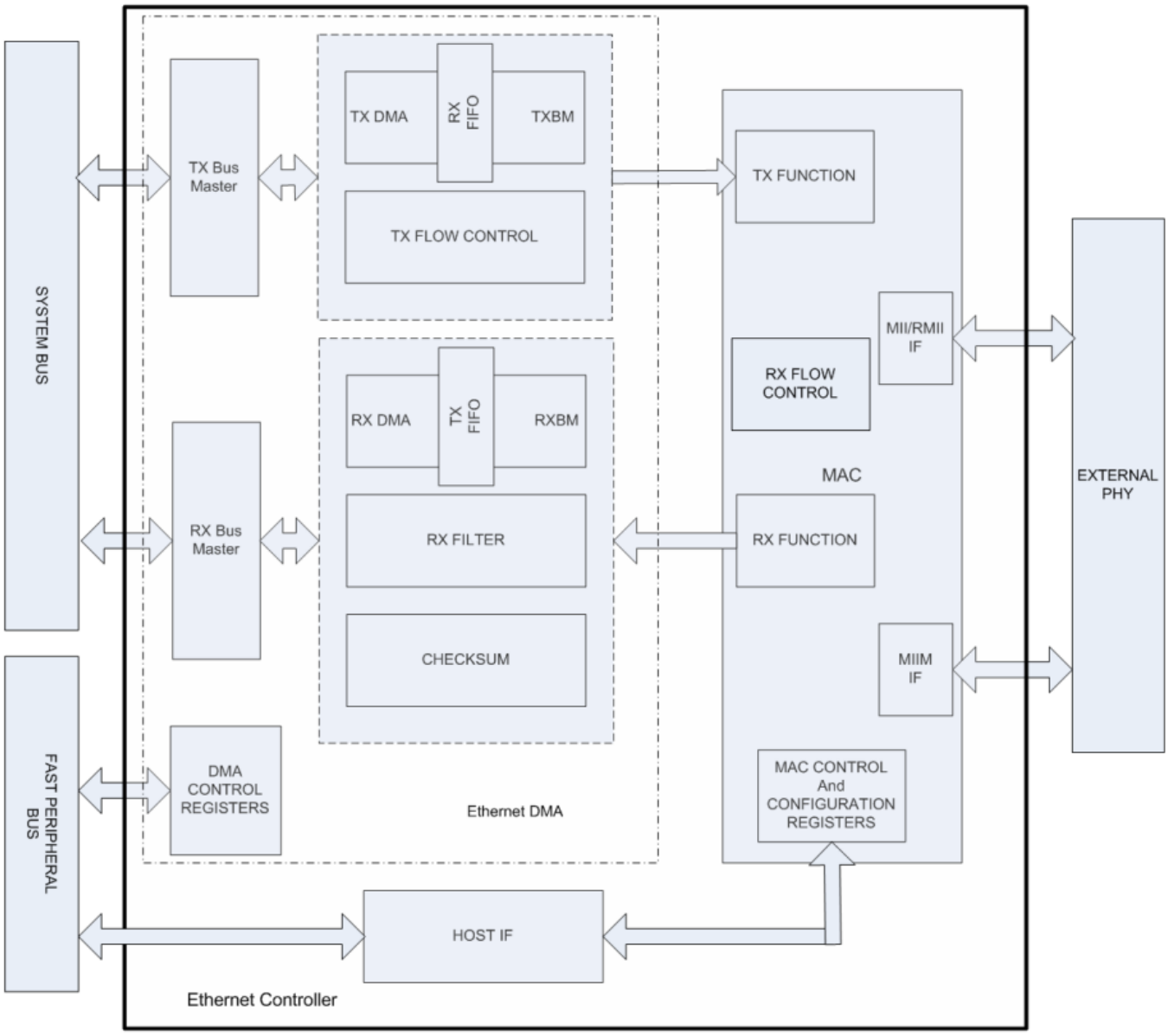
Description

The Ethernet Controller provides the modules needed to implement a 10/100 Mbps Ethernet node using an external Ethernet PHY chip. The PHY chip provides a digital-analog interface as part of the Physical Layer and the controller provides the Media Access Controller (MAC) layer above the PHY.

As shown in Figure 1, the Ethernet Controller consists of the following modules:

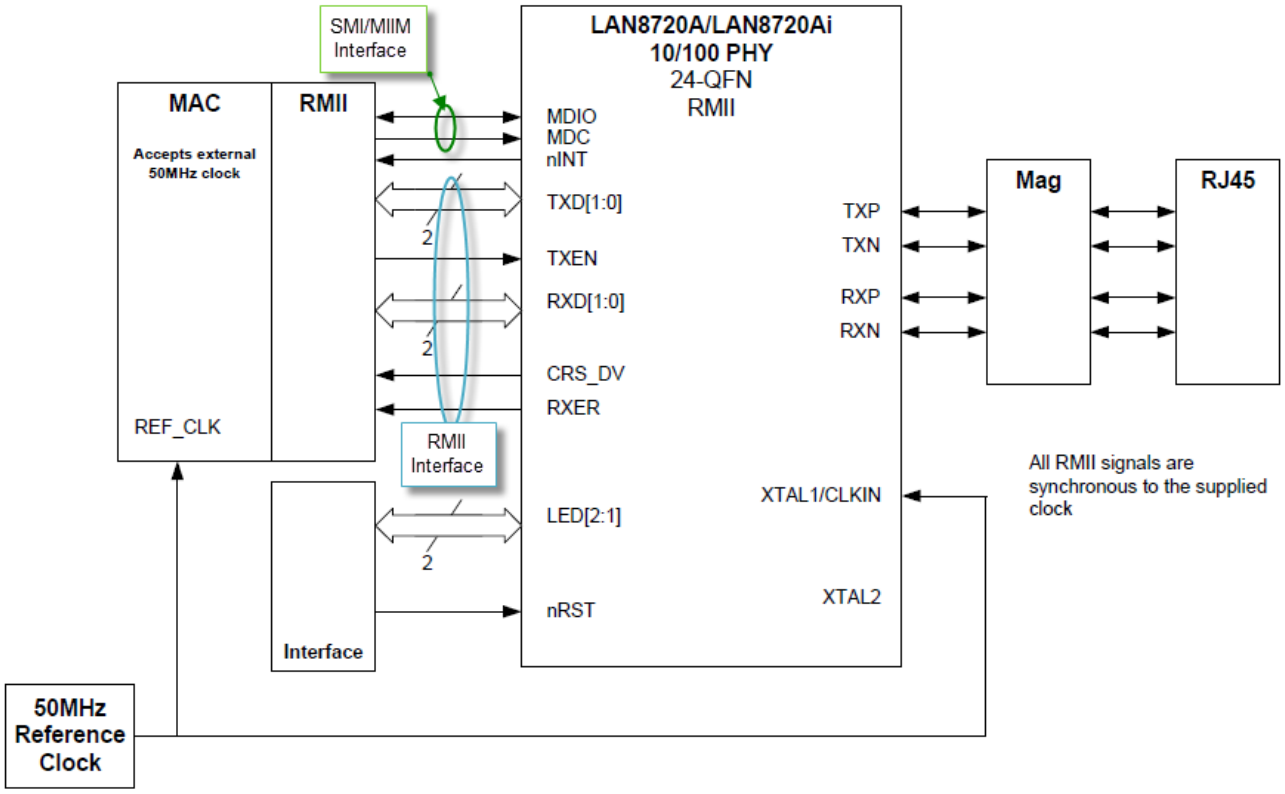
- Media Access Control (MAC) block: Responsible for implementing the MAC functions of the Ethernet IEEE 802.3 Specification
- Flow Control (FC) block: Responsible for control of the transmission of PAUSE frames. (Reception of PAUSE frames is handled within the MAC.)
- RX Filter (RXF) block: This module performs filtering on every receive packet to determine whether each packet should be accepted or rejected
- TX DMA/TX Buffer Management Engine: The TX DMA and TX Buffer Management engines perform data transfers from the memory (using descriptor tables) to the MAC Transmit Interface
- RX DMA/RX Buffer Management Engine: The RX DMA and RX Buffer Management engines transfer receive packets from the MAC to the memory (using descriptor tables)

Figure 1: Ethernet Controller Block Diagram



For completeness, we also need to look at the interface diagram of a representative Ethernet PHY. As shown in Figure 2, the PHY has two interfaces, one for configuring and managing the PHY (SMI/MIIM) and another for transmit and receive data (RMII or MII). The SMI/MIIM interface is the responsibility of the Ethernet PHY Driver Library. When setting up the Ethernet PHY, this Ethernet driver calls primitives from the Ethernet PHY Driver library. The RMII/MII data interface is the responsibility of the Ethernet MAC Driver Library (this library).

Figure 2: Ethernet PHY Interfaces



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system. Refer to the TCP/IP Stack Library MAC Driver Module help for the interface that the Ethernet driver has to implement in a MPLAB Harmony system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Ethernet MAC Driver Library.

Library Interface Section	Description
Client Level Functions	DRV_ETHMAC_PIC32MACOpen, DRV_ETHMAC_PIC32MACClose, and DRV_ETHMAC_PIC32MACSetup to support the TCP/IP Stack. Plus link status and power options.
Receive Functions	Receive routines.
Transmit Functions	Transmit routines.
Event Functions	Ethernet event support routines.
Other Functions	Additional routines.
Data Types and Constants	Typedefs and #defines.

Configuring the Library

Macros

	Name	Description
	DRV_ETHMAC_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_ETHMAC_INDEX	Ethernet MAC static index selection.
	DRV_ETHMAC_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_ETHMAC_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_ETHMAC_INTERRUPT_SOURCE	Defines an override of the interrupt source in case of static driver.
	DRV_ETHMAC_PERIPHERAL_ID	Defines an override of the peripheral ID.
	DRV_ETHMAC_POWER_STATE	Defines an override of the power state of the Ethernet MAC driver.

Description

The configuration of the Ethernet MAC driver is done as part of the MPLAB Harmony TCP/IP Stack configuration and is based on the `system_config.h` file, which may include the `tcpip_mac_config.h`. See the TCP/IP Stack Library MAC Driver Module help file for configuration options.

This header file contains the configuration selection for the Ethernet MAC Driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_ETHMAC_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_CLIENTS_NUMBER 1
```

Description

Ethernet MAC Maximum Number of Clients

This definition select the maximum number of clients that the Ethernet MAC driver can support at run time.

Remarks

The MAC driver is not a true multi-client driver. Under normal usage, the only client of the MAC driver is the TCP/IP stack. After the MAC driver provided an DRV_HANDLE as a result of an Open operation, any other attempt to call Open will return a invalid handle. Default value should be 1.

However, for allowing other modules to interface directly with the MAC driver while the TCP/IP stack currently uses the the MAC driver this symbol can have a value greater than 1. But the returned handle is the same one as the TCP/IP stack uses.

DRV_ETHMAC_INDEX Macro

Ethernet MAC static index selection.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_INDEX DRV_ETHMAC_INDEX_1
```

Description

Ethernet MAC Static Index Selection

This definition selects the Ethernet MAC static index for the driver object reference

Remarks

This index is required to make a reference to the driver object.

DRV_ETHMAC_INSTANCES_NUMBER Macro

Selects the maximum number of hardware instances that can be supported by the dynamic driver.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_INSTANCES_NUMBER 1
```

Description

Ethernet MAC hardware instance configuration

This definition selects the maximum number of hardware instances that can be supported by the dynamic driver. Not defining it means using a static driver.

Remarks

None.

DRV_ETHMAC_INTERRUPT_MODE Macro

Controls operation of the driver in the interrupt or polled mode.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_INTERRUPT_MODE true
```

Description

Ethernet MAC Interrupt And Polled Mode Operation Control

This macro controls the operation of the driver in the interrupt mode of operation. The possible values of this macro are:

- true - Select if interrupt mode of timer operation is desired
- false - Select if polling mode of timer operation is desired

Not defining this option to true or false will result in a build error.

Remarks

None.

DRV_ETHMAC_INTERRUPT_SOURCE Macro

Defines an override of the interrupt source in case of static driver.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_INTERRUPT_SOURCE INT_SOURCE_ETH_1
```

Description

Ethernet MAC Interrupt Source

Defines an override of the interrupt source in case of static driver.

Remarks

Refer to the INT PLIB document for more information on INT_SOURCE enumeration.

DRV_ETHMAC_PERIPHERAL_ID Macro

Defines an override of the peripheral ID.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_PERIPHERAL_ID ETHMAC_ID_1
```

Description

Ethernet MAC Peripheral ID Selection

Defines an override of the peripheral ID, using macros.

Remarks

Some devices also support ETHMAC_ID_0

DRV_ETHMAC_POWER_STATE Macro

Defines an override of the power state of the Ethernet MAC driver.

File

drv_ethmac_config.h

C

```
#define DRV_ETHMAC_POWER_STATE SYS_MODULE_POWER_IDLE_STOP
```

Description

Ethernet MAC power state configuration

Defines an override of the power state of the Ethernet MAC driver.

Remarks

This feature may not be available in the device or the Ethernet MAC module selected.

Building the Library

This section lists the files that are available in the Ethernet MAC Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/ethmac`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_ethmac.h`	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_ethmac.c`	PIC32 internal Ethernet driver virtual MAC implementation file.
`/src/dynamic/drv_ethmac_lib.c`	PIC32 internal Ethernet driver controller implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The Ethernet MAC Driver Library depends on the following modules:

- Ethernet PHY Driver Library
- Interrupt System Service Library
- Timer System Service Library
- Ethernet Peripheral Library

Library Interface

a) Client Level Functions

	Name	Description
	DRV_ETHMAC_PIC32MACClose	Closes a client instance of the PIC32 MAC Driver. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACDeinitialize	Deinitializes the PIC32 Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACInitialize	Initializes the PIC32 Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACLinkCheck	Checks current link status. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACOpen	Opens a client instance of the PIC32 MAC Driver. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACParametersGet	MAC parameter get function. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACPowerMode	Selects the current power mode for the Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACProcess	MAC periodic processing function. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACStatisticsGet	Gets the current MAC statistics. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACStatus	Provides the current status of the MAC driver module. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACConfigGet	Gets the current MAC driver configuration. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACRegisterStatisticsGet	Gets the current MAC hardware statistics registers. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACReinitialize	Reinitializes the PIC32 Ethernet MAC. **Implementation:** Dynamic

b) Receive Functions

	Name	Description
	DRV_ETHMAC_PIC32MACPacketRx	This is the MAC receive function. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACRxFilterHashTableEntrySet	Sets the current MAC hash table receive filter. **Implementation:** Dynamic

c) Transmit Functions

	Name	Description
	DRV_ETHMAC_PIC32MACPacketTx	MAC driver transmit function. **Implementation:** Dynamic

d) Event Functions

	Name	Description
	DRV_ETHMAC_PIC32MACEventAcknowledge	Acknowledges and re-enables processed events. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACEventMaskSet	Enables/disables the MAC events. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACEventPendingGet	Returns the currently pending events. **Implementation:** Dynamic

e) Other Functions

	Name	Description
	DRV_ETHMAC_Tasks_ISR	Ethernet MAC driver interrupt function. **Implementation:** Dynamic

	DRV_ETHMAC_PIC32MACTasks	Maintains the EThernet MAC driver's state machine. **Implementation:** Dynamic

f) Data Types and Constants

	Name	Description
	DRV_ETHMAC_INDEX_1	This is macro DRV_ETHMAC_INDEX_1.
	DRV_ETHMAC_INDEX_0	Ethernet driver index definitions.
	DRV_ETHMAC_INDEX_COUNT	Number of valid Ethernet driver indices.

Description

This section lists the interface routines, data types, constants and macros for the library.

a) Client Level Functions

DRV_ETHMAC_PIC32MACClose Function

Closes a client instance of the PIC32 MAC Driver.

Implementation: Dynamic

File

drv_ethmac.h

C

```
void DRV_ETHMAC_PIC32MACClose(DRV_HANDLE hMac);
```

Returns

None

Description

This function closes a client instance of the PIC32 MAC Driver.

Remarks

None

Preconditions

DRV_ETHMAC_PIC32MACOpen() should have been called.

Example

Parameters

Parameters	Description
hMac	valid MAC handle, obtained by a call to DRV_ETHMAC_PIC32MACOpen

Function

void DRV_ETHMAC_PIC32MACClose(DRV_HANDLE hMac)

DRV_ETHMAC_PIC32MACDeinitialize Function

Deinitializes the PIC32 Ethernet MAC.

Implementation: Dynamic

File

drv_ethmac.h

C

```
void DRV_ETHMAC_PIC32MACDeinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function supports teardown of the PIC32 Ethernet MAC (opposite of set up). Used by tcpip_module_manager.

Remarks

This function deinitializes the Ethernet controller, the MAC and the associated PHY. It should be called to be release any resources allocated by the initialization and return the MAC and the PHY to the idle/power down state.

Preconditions

DRV_ETHMAC_PIC32MACInitialize must have been called to set up the driver.

Example

Function

void DRV_ETHMAC_PIC32MACDeinitialize(SYS_MODULE_OBJ object);

DRV_ETHMAC_PIC32MACInitialize Function

Initializes the PIC32 Ethernet MAC.

Implementation: Dynamic

File

drv_ethmac.h

C

```
SYS_MODULE_OBJ DRV_ETHMAC_PIC32MACInitialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const
init);
```

Returns

- a valid handle to a driver object, if successful.
- SYS_MODULE_OBJ_INVALID if initialization failed.

Description

This function supports the initialization of the PIC32 Ethernet MAC. Used by tcpip_module_manager.

Remarks

This function initializes the Ethernet controller, the MAC and the associated PHY. It should be called to be able to schedule any Ethernet transmit or receive operation.

Preconditions

None

Example

Function

SYS_MODULE_OBJ DRV_ETHMAC_PIC32MACInitialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);

DRV_ETHMAC_PIC32MACLinkCheck Function

Checks current link status.

Implementation: Dynamic

File

drv_ethmac.h

C

```
bool DRV_ETHMAC_PIC32MACLinkCheck(DRV_HANDLE hMac);
```

Returns

- true - If the link is up

- false - If the link is not up

Description

This function checks the link status of the associated network interface.

Remarks

The function will automatically perform a MAC reconfiguration if the link went up after being down and the PHY auto negotiation is enabled.

Preconditions

DRV_ETHMAC_PIC32MACInitialize must have been called to set up the driver. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Parameters

Parameters	Description
hMac	Ethernet MAC client handle

Function

bool DRV_ETHMAC_PIC32MACLinkCheck(DRV_HANDLE hMac)

DRV_ETHMAC_PIC32MACOpen Function

Opens a client instance of the PIC32 MAC Driver.

Implementation: Dynamic

File

drv_ethmac.h

C

```
DRV_HANDLE DRV_ETHMAC_PIC32MACOpen(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

- DRV_HANDLE - handle (pointer) to MAC client
- 0 if call failed

Description

This function opens a client instance of the PIC32 MAC Driver. Used by tcpip_module_manager.

Remarks

The intent parameter is not used in the current implementation and is maintained only for compatibility with the generic driver Open function signature.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called.

Example

Function

DRV_HANDLE DRV_ETHMAC_PIC32MACOpen(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);

DRV_ETHMAC_PIC32MACParametersGet Function

MAC parameter get function.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_RES DRV_ETHMAC_PIC32MACParametersGet(DRV_HANDLE hMac, TCPIP_MAC_PARAMETERS* pMacParams);
```

Returns

- TCPIP_MAC_RES_OK if pMacParams updated properly
- a TCPIP_MAC_RES error code if processing failed for some reason

Description

MAC Parameter Get function TCPIP_MAC_RES DRV_ETHMAC_PIC32MACParametersGet(DRV_HANDLE hMac, TCPIP_MAC_PARAMETERS* pMacParams);

This is a function that returns the run time parameters of the MAC driver.

Remarks

None.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

DRV_ETHMAC_PIC32MACPowerMode Function

Selects the current power mode for the Ethernet MAC.

Implementation: Dynamic

File

drv_ethmac.h

C

```
bool DRV_ETHMAC_PIC32MACPowerMode(DRV_HANDLE hMac, TCPIP_MAC_POWER_MODE pwrMode);
```

Returns

- true if the call succeeded.
- false if the call failed

Description

This function sets the power mode for the Ethernet MAC.

Remarks

This function is not currently supported by the Ethernet MAC and will always return true.

Preconditions

DRV_ETHMAC_PIC32MACInitialize must have been called to set up the driver. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

bool DRV_ETHMAC_PIC32MACPowerMode(DRV_HANDLE hMac, TCPIP_MAC_POWER_MODE pwrMode)

DRV_ETHMAC_PIC32MACProcess Function

MAC periodic processing function.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_RES DRV_ETHMAC_PIC32MACProcess(DRV_HANDLE hMac);
```

Returns

- TCPIP_MAC_RES_OK if all processing went on OK
- a TCPIP_MAC_RES error code if processing failed for some reason

Description

This is a function that allows for internal processing by the MAC driver. It is meant for processing that cannot be done from within ISR.

Normally this function will be called in response to an TX and/or RX event signaled by the driver. This is specified by the MAC driver at initialization time using TCPIP_MAC_MODULE_CTRL.

Remarks

- The MAC driver may use the DRV_ETHMAC_PIC32MACProcess() for:
  - Processing its pending TX queues
  - RX buffers replenishing functionality. If the number of packets in the RX queue falls below a specified limit, the MAC driver may use this function to allocate some extra RX packets. Similarly, if there are too many allocated RX packets, the MAC driver can free some of them.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Parameters

Parameters	Description
hMac	Ethernet MAC client handle

Function

TCPIP_MAC_RES DRV_ETHMAC_PIC32MACProcess(DRV_HANDLE hMac);

DRV_ETHMAC_PIC32MACStatisticsGet Function

Gets the current MAC statistics.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_RES DRV_ETHMAC_PIC32MACStatisticsGet(DRV_HANDLE hMac, TCPIP_MAC_RX_STATISTICS* pRxStatistics,
TCPIP_MAC_TX_STATISTICS* pTxStatistics);
```

Returns

- TCPIP_MAC_RES_OK if all processing went on OK.
- TCPIP_MAC_RES_OP_ERR error code if function not supported by the driver.

Description

This function will get the current value of the statistic counters maintained by the MAC driver.

Remarks

- The reported values are info only and change dynamically.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

TCPIP_MAC_RES DRV_ETHMAC_PIC32MACStatisticsGet(DRV_HANDLE hMac, TCPIP_MAC_RX_STATISTICS* pRxStatistics, TCPIP_MAC_TX_STATISTICS* pTxStatistics);

DRV_ETHMAC_PIC32MACStatus Function

Provides the current status of the MAC driver module.

Implementation: Dynamic

File

drv_ethmac.h

C

```
SYS_STATUS DRV_ETHMAC_PIC32MACStatus(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed
- SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed
- SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This function provides the current status of the MAC driver module.

Remarks

None.

Preconditions

The DRV_ETHMAC_PIC32MACInitialize function must have been called before calling this function.

Example

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_ETHMAC_PIC32MACInitialize

Function

SYS_STATUS DRV_ETHMAC_PIC32MACStatus (SYS_MODULE_OBJ object)

DRV_ETHMAC_PIC32MACConfigGet Function

Gets the current MAC driver configuration.

Implementation: Dynamic

File

drv_ethmac.h

C

```
size_t DRV_ETHMAC_PIC32MACConfigGet(DRV_HANDLE hMac, void* configBuff, size_t buffSize, size_t*
pConfigSize);
```

Returns

- number of bytes copied into the supplied storage buffer

Description

This function will get the current MAC driver configuration and store it into a supplied buffer.

Remarks

- None

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

size_t DRV_ETHMAC_PIC32MACConfigGet(DRV_HANDLE hMac, void* configBuff, size_t buffSize, size_t* pConfigSize);

DRV_ETHMAC_PIC32MACRegisterStatisticsGet Function

Gets the current MAC hardware statistics registers.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_RES DRV_ETHMAC_PIC32MACRegisterStatisticsGet(DRV_HANDLE hMac, TCPIP_MAC_STATISTICS_REG_ENTRY*
pRegEntries, int nEntries, int* pHwEntries);
```

Returns

- TCPIP_MAC_RES_OK if all processing went on OK.
- TCPIP_MAC_RES_OP_ERR error code if function not supported by the driver.

Description

This function will get the current value of the statistic registers of the associated MAC controller.

Remarks

- The reported values are info only and change dynamically.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

TCPIP_MAC_RES DRV_ETHMAC_PIC32MACRegisterStatisticsGet(DRV_HANDLE hMac, TCPIP_MAC_STATISTICS_REG_ENTRY* pRegEntries, int nEntries, int* pHwEntries);

DRV_ETHMAC_PIC32MACReinitialize Function

Reinitializes the PIC32 Ethernet MAC.

Implementation: Dynamic

File

drv_ethmac.h

C

```
void DRV_ETHMAC_PIC32MACReinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
```

Returns

None.

Description

This function supports re-initialization of the PIC32 Ethernet MAC (opposite of set up).

Remarks

This function is not supported yet.

Preconditions

DRV_ETHMAC_PIC32MACInitialize must have been called to set up the driver.

Example

Function

void DRV_ETHMAC_PIC32MACReinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);

b) Receive Functions

DRV_ETHMAC_PIC32MACPacketRx Function

This is the MAC receive function.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_PACKET* DRV_ETHMAC_PIC32MACPacketRx(DRV_HANDLE hMac, TCPIP_MAC_RES* pRes, const
TCPIP_MAC_PACKET_RX_STAT** ppPktStat);
```

Returns

- a valid pointer to an available RX packet
- 0 if no packet pending/available

Description

This function will return a packet if such a pending packet exists.

Additional information about the packet is available by providing the pRes and ppPktStat fields.

Remarks

- Once a pending packet is available in the MAC driver internal RX queues this function will dequeue the packet and hand it over to the MAC driver's client - i.e., the stack - for further processing.

- The flags for a RX packet are updated by the MAC driver:
  - TCPIP_MAC_PKT_FLAG_RX will be set
  - TCPIP_MAC_PKT_FLAG_UNICAST is set if that packet is a unicast packet
  - TCPIP_MAC_PKT_FLAG_BCAST is set if that packet is a broadcast packet
  - TCPIP_MAC_PKT_FLAG_MCAST is set if that packet is a multicast packet
  - TCPIP_MAC_PKT_FLAG_QUEUED is set
  - TCPIP_MAC_PKT_FLAG_SPLIT is set if the packet has multiple data segments

- The MAC driver dequeues and return to the caller just one single packet. That is the packets are not chained.

- The packet buffers are allocated by the Ethernet MAC driver itself, Once the higher level layers in the stack are done with processing the RX packet, they have to call the corresponding packet acknowledgment function that tells the MAC driver that it can resume control of that packet.

- Once the stack modules are done processing the RX packets and the acknowledge function is called the MAC driver will reuse the RX packets.

- The MAC driver may use the DRV_ETHMAC_PIC32MACProcess() for obtaining new RX packets if needed.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

TCPIP_MAC_PACKET* DRV_ETHMAC_PIC32MACPacketRx (DRV_HANDLE hMac, TCPIP_MAC_RES* pRes, const TCPIP_MAC_PACKET_RX_STAT** ppPktStat);

DRV_ETHMAC_PIC32MACRxFilterHashTableEntrySet Function

Sets the current MAC hash table receive filter.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_RES DRV_ETHMAC_PIC32MACRxFilterHashTableEntrySet(DRV_HANDLE hMac, const TCPIP_MAC_ADDR*
DestMACAddr);
```

Returns

- TCPIP_MAC_RES_OK if success
- a TCPIP_MAC_RES error value if failed

Description

This function sets the MAC hash table filtering to allow packets sent to DestMACAddr to be received. It calculates a CRC-32 using polynomial 0x4C11DB7 over the 6 byte MAC address and then, using bits 28:23 of the CRC, will set the appropriate bits in the hash table filter registers (ETHHT0-ETHHT1).

The function will enable/disable the Hash Table receive filter if needed.

Remarks

- Sets the appropriate bit in the ETHHT0/1 registers to allow packets sent to DestMACAddr to be received and enabled the Hash Table receive filter.
- There is no way to individually remove destination MAC addresses from the hash table since it is possible to have a hash collision and therefore multiple MAC addresses relying on the same hash table bit.
- A workaround is to have the stack store each enabled MAC address and to perform the comparison at run time.
- A call to DRV_ETHMAC_PIC32MACRxFilterHashTableEntrySet() using a 00-00-00-00-00-00 destination MAC address, which will clear the entire hash table and disable the hash table filter. This will allow the receive of all packets, regardless of their destination

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

TCPIP_MAC_RES DRV_ETHMAC_PIC32MACRxFilterHashTableEntrySet(DRV_HANDLE hMac, const TCPIP_MAC_ADDR* DestMACAddr)

c) Transmit Functions

DRV_ETHMAC_PIC32MACPacketTx Function

MAC driver transmit function.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_RES DRV_ETHMAC_PIC32MACPacketTx(DRV_HANDLE hMac, TCPIP_MAC_PACKET * ptrPacket);
```

Returns

- TCPIP_MAC_RES_OK if success
- a TCPIP_MAC_RES error value if failed

Description

This is the MAC transmit function. Using this function a packet is submitted to the MAC driver for transmission.

Remarks

- The MAC driver supports internal queuing. A packet is rejected only if it's not properly formatted. Otherwise it will be scheduled for transmission and queued internally if needed.
- Once the packet is scheduled for transmission the MAC driver will set the TCPIP_MAC_PKT_FLAG_QUEUED flag so that the stack is aware that this packet is under processing and cannot be modified.
- Once the packet is transmitted, the TCPIP_MAC_PKT_FLAG_QUEUED will be cleared, the proper packet acknowledgment result (ackRes) will

be set and the packet acknowledgment function (ackFunc) will be called.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

Function

TCPIP_MAC_RES DRV_ETHMAC_PIC32MACPacketTx(DRV_HANDLE hMac, TCPIP_MAC_PACKET * ptrPacket);

d) Event Functions

DRV_ETHMAC_PIC32MACEventAcknowledge Function

Acknowledges and re-enables processed events.

Implementation: Dynamic

File

drv_ethmac.h

C

```
bool DRV_ETHMAC_PIC32MACEventAcknowledge(DRV_HANDLE hMac, TCPIP_MAC_EVENT tcpAckEv);
```

Returns

- true if events acknowledged
- false if no events to be acknowledged

Description

This function acknowledges and re-enables processed events. Multiple events can be ORed together as they are processed together. The events acknowledged by this function should be the events that have been retrieved from the stack by calling DRV_ETHMAC_PIC32MACEventPendingGet() or have been passed to the stack by the driver using the registered notification handler and have been processed and have to be re-enabled.

Remarks

- All events should be acknowledged, in order to be re-enabled.
- Some events are fatal errors and should not be acknowledged (TCPIP_MAC_EV_RX_BUSERR, TCPIP_MAC_EV_TX_BUSERR). Driver/stack re-initialization is needed under such circumstances.
- Some events are just system/application behavior and they are intended only as simple info (TCPIP_MAC_EV_RX_OVFLOW, TCPIP_MAC_EV_RX_BUFNA, TCPIP_MAC_EV_TX_ABORT, TCPIP_MAC_EV_RX_ACT).
- The TCPIP_MAC_EV_RX_FWMARK and TCPIP_MAC_EV_RX_EWMARK events are part of the normal flow control operation (if auto flow control was enabled). They should be enabled alternatively, if needed.
- The events are persistent. They shouldn't be re-enabled unless they have been processed and the condition that generated them was removed. Re-enabling them immediately without proper processing will have dramatic effects on system performance.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

```
DRV_ETHMAC_PIC32MACEventAcknowledge( hMac, stackNewEvents );
```

Function

bool DRV_ETHMAC_PIC32MACEventAcknowledge(DRV_HANDLE hMac, TCPIP_MAC_EVENT tcpAckEv);

DRV_ETHMAC_PIC32MACEventMaskSet Function

Enables/disables the MAC events.

Implementation: Dynamic

File

drv_ethmac.h

C

```
bool DRV_ETHMAC_PIC32MACEventMaskSet(DRV_HANDLE hMac, TCPIP_MAC_EVENT macEvents, bool enable);
```

Returns

always true, operation succeeded.

Description

This is a function that enables or disables the events to be reported to the Ethernet MAC client (TCP/IP stack).

All events that are to be enabled will be added to the notification process. All events that are to be disabled will be removed from the notification process. The stack has to catch the events that are notified and process them. After that the stack should call DRV_ETHMAC_PIC32MACEventAcknowledge() so that the events can be re-enable

The stack should process at least the following transfer events:

- TCPIP_MAC_EV_RX_PKTPEND
- TCPIP_MAC_EV_RX_DONE
- TCPIP_MAC_EV_TX_DONE

Remarks

- The event notification system enables the user of the TCP/IP stack to call into the stack for processing only when there are relevant events rather than being forced to periodically call from within a loop.
- If the notification events are nil, the interrupt processing will be disabled. Otherwise, the event notification will be enabled and the interrupts relating to the requested events will be enabled.
- Note that once an event has been caught by the stack ISR (and reported if a notification handler is in place) it will be disabled until the DRV_ETHMAC_PIC32MACEventAcknowledge() is called.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

```
DRV_ETHMAC_PIC32MACEventMaskSet( hMac, TCPIP_MAC_EV_RX_OVFLOW | TCPIP_MAC_EV_RX_BUFNA, true );
```

Function

bool DRV_ETHMAC_PIC32MACEventMaskSet(DRV_HANDLE hMac, TCPIP_MAC_EVENT macEvents, bool enable);

DRV_ETHMAC_PIC32MACEventPendingGet Function

Returns the currently pending events.

Implementation: Dynamic

File

drv_ethmac.h

C

```
TCPIP_MAC_EVENT DRV_ETHMAC_PIC32MACEventPendingGet(DRV_HANDLE hMac);
```

Returns

The currently stack pending events.

Description

This function returns the currently pending Ethernet MAC events. Multiple events will be ORed together as they accumulate. The stack should perform processing whenever a transmission related event (TCPIP_MAC_EV_RX_PKTPEND, TCPIP_MAC_EV_TX_DONE) is present. The other, non critical events, may not be managed by the stack and passed to an user. They will have to be eventually acknowledged if re-enabling is needed.

Remarks

- This is the preferred method to get the current pending MAC events. The stack maintains a proper image of the events from their occurrence to their acknowledgment.
- Even with a notification handler in place it's better to use this function to get the current pending events rather than using the events passed by the notification handler which could be stale.
- The events are persistent. They shouldn't be re-enabled unless they have been processed and the condition that generated them was removed. Re-enabling them immediately without proper processing will have dramatic effects on system performance.
- The returned value is just a momentary value. The pending events can change any time.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. DRV_ETHMAC_PIC32MACOpen() should have been called to obtain a valid handle.

Example

```
TCPIP_MAC_EVENT currEvents = DRV_ETHMAC_PIC32MACEventPendingGet( hMac);
```

Function

TCPIP_MAC_EVENT DRV_ETHMAC_PIC32MACEventPendingGet(DRV_HANDLE hMac)

e) Other Functions

DRV_ETHMAC_Tasks_ISR Function

Ethernet MAC driver interrupt function.

Implementation: Dynamic

File

drv_ethmac.h

C

```
void DRV_ETHMAC_Tasks_ISR(SYS_MODULE_OBJ macIndex);
```

Returns

None.

Description

This is the Ethernet MAC driver interrupt service routine. It processes the Ethernet related interrupts and notifies the events to the driver user (the TCP/IP stack).

Remarks

None.

Preconditions

DRV_ETHMAC_PIC32MACInitialize() should have been called. The TCP/IP stack event notification should be enabled.

Function

void DRV_ETHMAC_Tasks_ISR(SYS_MODULE_OBJ macIndex)

DRV_ETHMAC_PIC32MACTasks Function

Maintains the EThernet MAC driver's state machine.

Implementation: Dynamic

File

drv_ethmac.h

C

```
void DRV_ETHMAC_PIC32MACTasks(SYS_MODULE_OBJ object);
```

Returns

None

Description

This function is used to maintain the driver's internal state machine

Remarks

None.

Preconditions

The DRV_ETHMAC_PIC32MACInitialize routine must have been called for the specified MAC driver instance.

Example

Function

void DRV_ETHMAC_PIC32MACTasks(SYS_MODULE_OBJ object)

f) Data Types and Constants

DRV_ETHMAC_INDEX_1 Macro

File

drv_ethmac.h

C

```
#define DRV_ETHMAC_INDEX_1 1
```

Description

This is macro DRV_ETHMAC_INDEX_1.

DRV_ETHMAC_INDEX_0 Macro

Ethernet driver index definitions.

File

drv_ethmac.h

C

```
#define DRV_ETHMAC_INDEX_0 0
```

Description

Ethernet Driver Module Index Numbers

These constants provide Ethernet driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the MAC initialization routines to identify the driver instance in use.

DRV_ETHMAC_INDEX_COUNT Macro

Number of valid Ethernet driver indices.

File

drv_ethmac.h

C

```
#define DRV_ETHMAC_INDEX_COUNT ETH_NUMBER_OF_MODULES
```

Description

Ethernet Driver Module Index Count

This constant identifies number of valid Ethernet driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from part-specific header files defined as part of the peripheral libraries.

Files

Files

Name	Description
drv_ethmac.h	Ethernet MAC device driver interface file
drv_ethmac_config.h	Ethernet MAC driver configuration definitions template.

Description

This section lists the source and header files used by the Ethernet MAC Driver Library.

drv_ethmac.h

Ethernet MAC device driver interface file

Functions

	Name	Description
	DRV_ETHMAC_PIC32MACClose	Closes a client instance of the PIC32 MAC Driver. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACConfigGet	Gets the current MAC driver configuration. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACDeinitialize	Deinitializes the PIC32 Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACEventAcknowledge	Acknowledges and re-enables processed events. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACEventMaskSet	Enables/disables the MAC events. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACEventPendingGet	Returns the currently pending events. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACInitialize	Initializes the PIC32 Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACLinkCheck	Checks current link status. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACOpen	Opens a client instance of the PIC32 MAC Driver. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACPacketRx	This is the MAC receive function. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACPacketTx	MAC driver transmit function. **Implementation:** Dynamic

	DRV_ETHMAC_PIC32MACParametersGet	MAC parameter get function. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACPowerMode	Selects the current power mode for the Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACProcess	MAC periodic processing function. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACRegisterStatisticsGet	Gets the current MAC hardware statistics registers. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACReinitialize	Reinitializes the PIC32 Ethernet MAC. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACRxFilterHashTableEntrySet	Sets the current MAC hash table receive filter. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACStatisticsGet	Gets the current MAC statistics. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACStatus	Provides the current status of the MAC driver module. **Implementation:** Dynamic
	DRV_ETHMAC_PIC32MACTasks	Maintains the EThernet MAC driver's state machine. **Implementation:** Dynamic
	DRV_ETHMAC_Tasks_ISR	Ethernet MAC driver interrupt function. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_ETHMAC_INDEX_0	Ethernet driver index definitions.
	DRV_ETHMAC_INDEX_1	This is macro DRV_ETHMAC_INDEX_1.
	DRV_ETHMAC_INDEX_COUNT	Number of valid Ethernet driver indices.

Description

Ethernet MAC Device Driver Interface

The Ethernet MAC device driver provides a simple interface to manage the Ethernet peripheral. This file defines the interface definitions and prototypes for the Ethernet MAC driver.

File Name

drv_ethmac.h

Company

Microchip Technology Inc.

drv_ethmac_config.h

Ethernet MAC driver configuration definitions template.

Macros

	Name	Description
	DRV_ETHMAC_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_ETHMAC_INDEX	Ethernet MAC static index selection.
	DRV_ETHMAC_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_ETHMAC_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_ETHMAC_INTERRUPT_SOURCE	Defines an override of the interrupt source in case of static driver.
	DRV_ETHMAC_PERIPHERAL_ID	Defines an override of the peripheral ID.
	DRV_ETHMAC_POWER_STATE	Defines an override of the power state of the Ethernet MAC driver.

Description

ETHMAC Driver Configuration Definitions for the template version

These definitions statically define the driver's mode of operation.

File Name

drv_ethmac_config.h

Company

Microchip Technology Inc.

Ethernet PHY Driver Library

This section describes the Ethernet PHY Driver Library.

Introduction

This library provides a low-level abstraction of the Ethernet PHY Driver Library that is available on the Microchip family of microcontrollers with a convenient C language interface. It can be used to simplify low-level access to the module without the necessity of interacting directly with the module's registers, there by hiding differences from one microcontroller variant to another.

Description

This library provides a software abstraction for configuring external Ethernet PHY devices for use with the on-chip PIC32 Ethernet Controller.

Using the Library

The user of this driver is the MPLAB Harmony TCP/IP Stack through its Ethernet MAC driver. This Ethernet PHY driver is not intended as a system wide driver that the application or other system modules may use. It is intended for the sole use of the MPLAB Harmony TCP/IP stack and implements the PHY driver required by the Ethernet MAC.

This topic describes the basic architecture and functionality of the Ethernet PHY driver and is meant for advanced users or TCP/IP Stack driver developers.

Interface Header File: `drv_ethphy.h`

The interface to the Ethernet PHY library is defined in the `drv_ethphy.h` header file, which is included by the MPLAB Harmony TCP/IP stack.

Please refer to the What is MPLAB Harmony? section for how the library interacts with the framework.

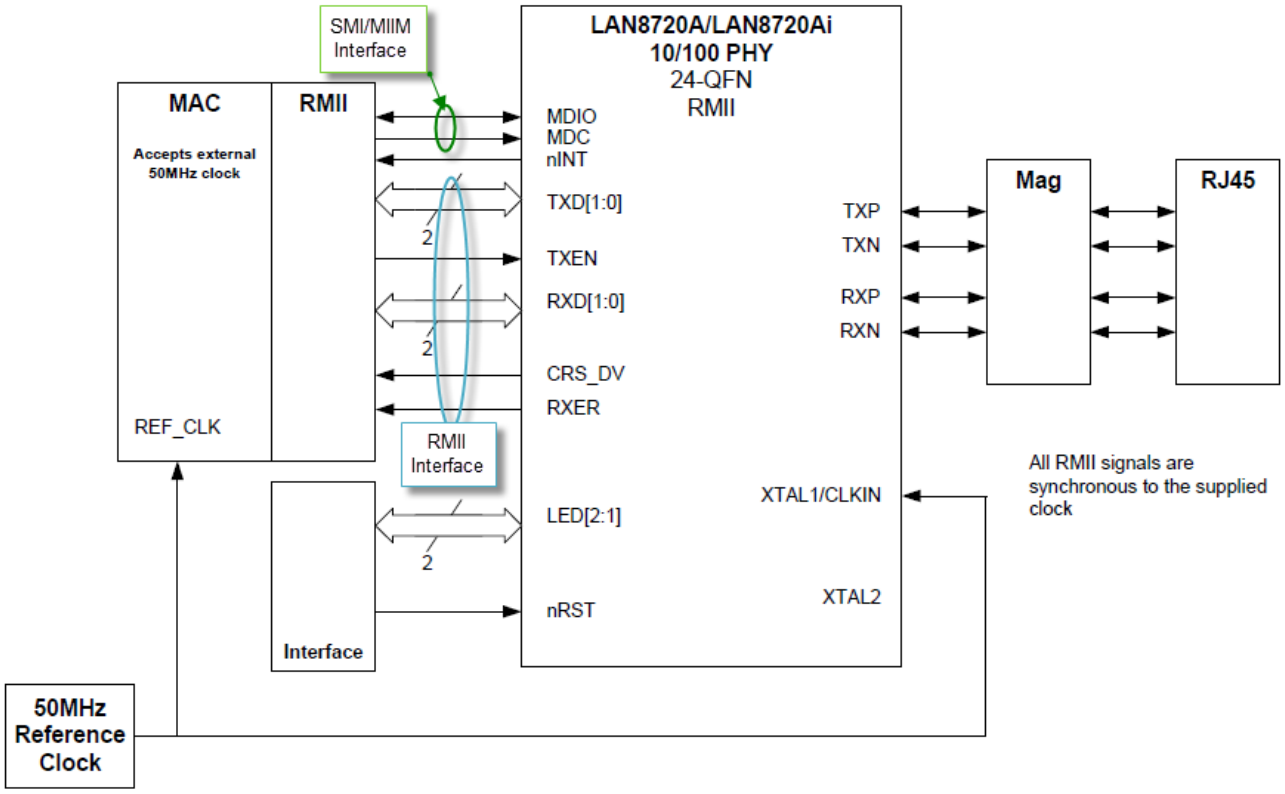
Abstraction Model

This library provides a low-level abstraction of the Ethernet PHY Driver Library on Microchip's microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

To understand how this library works you must first understand how an external Ethernet PHY interfaces with the Ethernet Controller. As shown in Figure 1, the PHY has two interfaces, one for managing the PHY, known as the Serial Management Interface (SMI), for configuring the device and a second, known as the Reduced Media Independent Interface (RMII), for transmit and receive data.

Figure 1: Typical External PHY Interface



The block diagram also shows an interrupt signal (nINT) going to a external interrupt pin on the host device and signals going to on-board LEDs to show link state and link activity.

The SMI interface is also known as the MII Management (MIIM) interface. This control interface is standardized for all PHYs by Clause 22 of the 802.3 standard. It provides up to 32 16-bit registers on the PHY. The following table provides a summary of all 32 registers. Consult the data sheet for the PHY device for the specific bit fields in each register.

Register Address	Register Name	Register Type
0	Control	Basic
1	Status	Basic
2, 3	PHY Identifier	Extended
4	Auto-Negotiation Advertisement	Extended
5	Auto-Negotiation Link Partner Base Page Ability	Extended
6	Auto-Negotiation Expansion	Extended
7	Auto-Negotiation Next Page Transmit	Extended
8	Auto-Negotiation Link Partner Received Next Page	Extended
9	MASTER-SLAVE Control Register	Extended
10	MASTER-SLAVE Status Register	Extended
11-14	Reserved	Extended
15	Extended Status	Reserved
16-31	Vendor Specific	Extended

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Ethernet PHY Driver Library

Library Interface Section	Description
System Level Functions	Routines that integrate the driver into the MPLAB Harmony framework.
Client Level Functions	Open, Close, Link Status, Auto Negotiation.
SMI/MIIM Functions	SMI/MIIM Management Interface.
External PHY Support Functions	Provides the API for PHY support routines that the driver will call when setting up the PHY. The driver library provides support for four PHYs.
Other Functions	Functions that provide software version information.
Data Types and Constants	C language typedefs and enums used by this library.

Configuring the Library

Macros

	Name	Description
	DRV_ETHPHY_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_ETHPHY_INDEX	Ethernet PHY static index selection.
	DRV_ETHPHY_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_ETHPHY_PERIPHERAL_ID	Defines an override of the peripheral ID.
	DRV_ETHPHY_NEG_DONE_TMO	Value of the PHY negotiation complete time out as per IEEE 802.3 spec.
	DRV_ETHPHY_NEG_INIT_TMO	Value of the PHY negotiation initiation time out as per IEEE 802.3 spec.
	DRV_ETHPHY_RESET_CLR_TMO	Value of the PHY Reset self clear time out as per IEEE 802.3 spec.

Description

The configuration of the Ethernet PHY Driver Library is based on the file `system_config.h`.

This header file contains the configuration selection for the Ethernet PHY Driver Library. Based on the selections made, the Ethernet PHY Driver Library may support the selected features.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_ETHPHY_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_CLIENTS_NUMBER 1
```

Description

Ethernet PHY Maximum Number of Clients This definition select the maximum number of clients that the Ethernet PHY driver can support at run time. Not defining it means using a single client.

Remarks

The MAC driver is the client of the PHY driver. Multiple clients may be needed when access to MIIM bus (for PHY vendor specific functionality) is needed through the PHY driver.

However MIIM operations are not supported when the PHY driver uses the MIIM driver for MIIM bus accesses. In this case the number of clients should be 1 and the DRV_MIIM should be used for accessing the MIIM bus.

DRV_ETHPHY_INDEX Macro

Ethernet PHY static index selection.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_INDEX DRV_ETHPHY_INDEX_1
```

Description

Ethernet PHY Static Index Selection

This definition selects the Ethernet PHY static index for the driver object reference.

Remarks

This index is required to make a reference to the driver object.

DRV_ETHPHY_INSTANCES_NUMBER Macro

Selects the maximum number of hardware instances that can be supported by the dynamic driver.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_INSTANCES_NUMBER 1
```

Description

Ethernet PHY hardware instance configuration

This definition selects the maximum number of hardware instances that can be supported by the dynamic driver. Not defining it means using a static driver.

Remarks

None.

DRV_ETHPHY_PERIPHERAL_ID Macro

Defines an override of the peripheral ID.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_PERIPHERAL_ID ETHPHY_ID_1
```

Description

Ethernet PHY Peripheral ID Selection

Defines an override of the peripheral ID, using macros.

Remarks

Some devices also support ETHPHY_ID_0

DRV_ETHPHY_NEG_DONE_TMO Macro

Value of the PHY negotiation complete time out as per IEEE 802.3 spec.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_NEG_DONE_TMO (2000)
```

Description

Ethernet PHY Negotiation Complete time out

This definition sets the time out of the PHY negotiation complete, in ms.

Remarks

See IEEE 802.3 Clause 28 Table 28-9 autoneg_wait_timer value (max 1s).

DRV_ETHPHY_NEG_INIT_TMO Macro

Value of the PHY negotiation initiation time out as per IEEE 802.3 spec.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_NEG_INIT_TMO (1)
```

Description

Ethernet PHY Negotiation Initiation time out

This definition sets the time out of the PHY negotiation initiation, in ms.

Remarks

None.

DRV_ETHPHY_RESET_CLR_TMO Macro

Value of the PHY Reset self clear time out as per IEEE 802.3 spec.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_RESET_CLR_TMO (500)
```

Description

Ethernet PHY Reset self clear time out

This definition sets the time out of the PHY Reset self clear, in ms.

Remarks

See IEEE 802.3 Clause 22 Table 22-7 and paragraph "22.2.4.1.1 Reset" (max 0.5s)

Building the Library

This section lists the files that are available in the Ethernet PHY Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/ethphy`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_ethphy.h`	Header file that exports the driver API.

Required File(s)

MHC *All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.*

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_extphy.c	Basic PHY driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
/src/dynamic/drv_extphy_smsc8700.c	SMSC 8700 PHY implementation file.
/src/dynamic/drv_extphy_smsc8720.c	SMSC 8720 PHY implementation file.
/src/dynamic/drv_extphy_smsc8720.c	SMSC 8740 PHY implementation file.
/src/dynamic/drv_extphy_ip101gr.c	IP101GR PHY implementation file.
/src/dynamic/drv_extphy_dp83640.c	National DP83640 PHY implementation file.
/src/dynamic/drv_extphy_dp83848.c	National DP83848 PHY implementation file.

Module Dependencies

The Ethernet MAC Driver Library depends on the following modules:

- Ethernet MAC Driver Library
- Clock System Service Library
- Ports System Service Library
- Timer System Service Library
- Ethernet Peripheral Library

Library Interface

a) System Level Functions

	Name	Description
	DRV_ETHPHY_Initialize	Initializes the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_Deinitialize	Deinitializes the specified instance of the Ethernet PHY driver module. **Implementation:** Dynamic
	DRV_ETHPHY_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings. **Implementation:** Dynamic
	DRV_ETHPHY_Status	Provides the current status of the Ethernet PHY driver module. **Implementation:** Dynamic
	DRV_ETHPHY_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Dynamic
	DRV_ETHPHY_HWConfigFlagsGet	Returns the current Ethernet PHY hardware MII/RMII and ALTERNATE/DEFAULT configuration flags. **Implementation:** Dynamic
	DRV_ETHPHY_Setup	Initializes Ethernet PHY configuration and set up procedure. **Implementation:** Dynamic

b) Client Level Functions

	Name	Description
	DRV_ETHPHY_ClientStatus	Gets the current client-specific status the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_Close	Closes an opened instance of the Ethernet PHY driver. **Implementation:** Dynamic

	DRV_ETHPHY_Open	Opens the specified Ethernet PHY driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_ETHPHY_Reset	Immediately resets the Ethernet PHY. **Implementation:** Dynamic
	DRV_ETHPHY_ClientOperationAbort	Aborts a current client operation initiated by the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_ClientOperationResult	Gets the result of a client operation initiated by the Ethernet PHY driver. **Implementation:** Dynamic

c) SMI/MIIM Functions

	Name	Description
	DRV_ETHPHY_SMIScanStatusGet	Gets the status of the SMI/MIIM scan data. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanStop	Stops the scan of a previously requested SMI/MIIM register. **Implementation:** Dynamic
	DRV_ETHPHY_SMIClockSet	Sets the SMI/MIIM interface clock. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanStart	Starts the scan of a requested SMI/MIIM register. **Implementation:** Dynamic
	DRV_ETHPHY_SMIRead	Initiates a SMI/MIIM read transaction. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanDataGet	Gets the latest SMI/MIIM scan data result. **Implementation:** Dynamic
	DRV_ETHPHY_SMIStatus	Returns the current status of the SMI/MIIM interface. **Implementation:** Dynamic
	DRV_ETHPHY_SMIWrite	Initiates a SMI/MIIM write transaction. **Implementation:** Dynamic

d) Vendor Functions

	Name	Description
	DRV_ETHPHY_VendorDataGet	Returns the current value of the vendor data. **Implementation:** Dynamic
	DRV_ETHPHY_VendorDataSet	Returns the current value of the vendor data. **Implementation:** Dynamic
	DRV_ETHPHY_VendorSMIReadResultGet	Reads the result of a previous vendor initiated SMI read transfer with DRV_ETHPHY_VendorSMIReadStart. **Implementation:** Dynamic
	DRV_ETHPHY_VendorSMIReadStart	Starts a vendor SMI read transfer. Data will be available with DRV_ETHPHY_VendorSMIReadResultGet. **Implementation:** Dynamic
	DRV_ETHPHY_VendorSMIWriteStart	Starts a vendor SMI write transfer. **Implementation:** Dynamic

e) Other Functions

	Name	Description
	DRV_ETHPHY_LinkStatusGet	Returns the current link status. **Implementation:** Dynamic
	DRV_ETHPHY_NegotiationIsComplete	Returns the results of a previously initiated Ethernet PHY negotiation. **Implementation:** Dynamic
	DRV_ETHPHY_NegotiationResultGet	Returns the result of a completed negotiation. **Implementation:** Dynamic
	DRV_ETHPHY_PhyAddressGet	Returns the PHY address. **Implementation:** Dynamic
	DRV_ETHPHY_RestartNegotiation	Restarts auto-negotiation of the Ethernet PHY link. **Implementation:** Dynamic

f) Data Types and Constants

	Name	Description
	DRV_ETHPHY_CLIENT_STATUS	Identifies the client-specific status of the Ethernet PHY driver.
	DRV_ETHPHY_INIT	Contains all the data necessary to initialize the Ethernet PHY device.
	DRV_ETHPHY_NEGOTIATION_RESULT	Contains all the data necessary to get the Ethernet PHY negotiation result
	DRV_ETHPHY_SETUP	Contains all the data necessary to set up the Ethernet PHY device.
	DRV_ETHPHY_VENDOR_MDIX_CONFIGURE	Pointer to function that configures the MDIX mode for the Ethernet PHY.
	DRV_ETHPHY_VENDOR_MII_CONFIGURE	Pointer to function to configure the Ethernet PHY in one of the MII/RMII operation modes.
	DRV_ETHPHY_VENDOR_SMI_CLOCK_GET	Pointer to a function to return the SMI/MIIM maximum clock speed in Hz of the Ethernet PHY.
	DRV_ETHPHY_INDEX_0	Ethernet PHY driver index definitions.
	DRV_ETHPHY_INDEX_1	This is macro DRV_ETHPHY_INDEX_1.
	DRV_ETHPHY_INDEX_COUNT	Number of valid Ethernet PHY driver indices.
	DRV_ETHPHY_LINK_STATUS	Defines the possible status flags of PHY Ethernet link.
	DRV_ETHPHY_CONFIG_FLAGS	Defines configuration options for the Ethernet PHY.
	DRV_ETHPHY_OBJECT	Identifies the interface of a Ethernet PHY vendor driver.
	DRV_ETHPHY_VENDOR_WOL_CONFIGURE	Pointer to a function to configure the PHY WOL functionality
	DRV_ETHPHY_OBJECT_BASE_TYPE	Identifies the base interface of a Ethernet PHY driver.
	DRV_ETHPHY_OBJECT_BASE	Identifies the base interface of a Ethernet PHY driver.
	DRV_ETHPHY_RESET_FUNCTION	Pointer to a function to perform an additional PHY reset
	DRV_ETHPHY_RESULT	Defines the possible results of Ethernet operations that can succeed or fail
	DRV_ETHPHY_USE_DRV_MIIM	Defines the way the PHY driver accesses the MIIM bus to communicate with the PHY.
	DRV_ETHPHY_INTERFACE_INDEX	Defines the index type for a PHY interface.
	DRV_ETHPHY_INTERFACE_TYPE	Defines the type of interface a PHY supports.

Description

This section describes the Application Programming Interface (API) functions of the Ethernet PHY Driver Library.

Refer to each section for a detailed description.

a) System Level Functions

DRV_ETHPHY_Initialize Function

Initializes the Ethernet PHY driver.

Implementation: Dynamic

File

drv_ethphy.h

C

```
SYS_MODULE_OBJ DRV_ETHPHY_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

- a valid handle to a driver object, if successful.
- SYS_MODULE_OBJ_INVALID if initialization failed.

Description

This function initializes the Ethernet PHY driver, making it ready for clients to open and use it.

Remarks

- This function must be called before any other Ethernet PHY routine is called.
- This function should only be called once during system initialization unless DRV_ETHPHY_Deinitialize is called to deinitialize the driver instance.
- The returned object must be passed as argument to DRV_ETHPHY_Reinitialize, DRV_ETHPHY_Deinitialize, DRV_ETHPHY_Tasks and DRV_ETHPHY_Status routines.

Preconditions

None.

Example

```
DRV_ETHPHY_INIT    init;
SYS_MODULE_OBJ  objectHandle;

// Populate the Ethernet PHY initialization structure
init.phyId  = ETHPHY_ID_0;

// Populate the Ethernet PHY initialization structure
init.phyId  = ETHPHY_ID_2;
init.pPhyObject  = &DRV_ETHPHY_OBJECT_SMSC_LAN8720;

// Do something

objectHandle = DRV_ETHPHY_Initialize(DRV_ETHPHY_INDEX_0, (SYS_MODULE_INIT*)&init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Function

SYS_MODULE_OBJ DRV_ETHPHY_Initialize(const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init)

DRV_ETHPHY_Deinitialize Function

Deinitializes the specified instance of the Ethernet PHY driver module.

Implementation: Dynamic

File

drv_ethphy.h

C

```
void DRV_ETHPHY_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the specified instance of the Ethernet PHY driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

- Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

Preconditions

The DRV_ETHPHY_Initialize function must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_ETHPHY_Initialize
SYS_STATUS          status;

DRV_ETHPHY_Deinitialize(object);

status = DRV_ETHPHY_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Function

void DRV_ETHPHY_Deinitialize (SYS_MODULE_OBJ object)

DRV_ETHPHY_Reinitialize Function

Reinitializes the driver and refreshes any associated hardware settings.

Implementation: Dynamic

File

drv_ethphy.h

C

```
void DRV_ETHPHY_Reinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
```

Returns

None.

Description

This function reinitializes the driver and refreshes any associated hardware settings using the initialization data given, but it will not interrupt any ongoing operations.

Remarks

- This function can be called multiple times to reinitialize the module.
- This operation can be used to refresh any supported hardware registers as specified by the initialization data or to change the power state of the module.

Preconditions

The DRV_ETHPHY_Initialize function must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
DRV_ETHPHY_INIT    init;
SYS_MODULE_OBJ  objectHandle;

// Populate the Ethernet PHY initialization structure
init.phyId  = ETHPHY_ID_2;
init.pPhyObject  = &DRV_ETHPHY_OBJECT_SMSC_LAN8720;

DRV_ETHPHY_Reinitialize(objectHandle, (SYS_MODULE_INIT*)&init);

phyStatus = DRV_ETHPHY_Status(objectHandle);
if (SYS_STATUS_BUSY == phyStatus)
{
    // Check again later to ensure the driver is ready
}
else if (SYS_STATUS_ERROR >= phyStatus)
{
    // Handle error
}
```

Function

void DRV_ETHPHY_Reinitialize(SYS_MODULE_OBJ object,

const SYS_MODULE_INIT * const init)

DRV_ETHPHY_Status Function

Provides the current status of the Ethernet PHY driver module.

Implementation: Dynamic

File

drv_ethphy.h

C

```
SYS_STATUS DRV_ETHPHY_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed
- SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed
- SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This function provides the current status of the Ethernet PHY driver module.

Remarks

- Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.
- SYS_STATUS_BUSY - Indicates that the driver is busy with a previous system level operation and cannot start another
- SYS_STATUS_ERROR - Indicates that the driver is in an error state
- Any value less than SYS_STATUS_ERROR is also an error state.
- SYS_MODULE_DEINITIALIZED - Indicates that the driver has been deinitialized
- The this operation can be used to determine when any of the driver's module level operations has completed.
- If the status operation returns SYS_STATUS_BUSY, the a previous operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.
- The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.
- This function will NEVER block waiting for hardware.
- If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The DRV_ETHPHY_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_ETHPHY_Initialize
SYS_STATUS          status;

status = DRV_ETHPHY_Status(object);
if (SYS_STATUS_ERROR >= status)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_ETHPHY_Initialize

Function

SYS_STATUS DRV_ETHPHY_Status (SYS_MODULE_OBJ object)

DRV_ETHPHY_Tasks Function

Maintains the driver's state machine and implements its ISR.

Implementation: Dynamic

File

drv_ethphy.h

C

```
void DRV_ETHPHY_Tasks(SYS_MODULE_OBJ object);
```

Returns

None

Description

This function is used to maintain the driver's internal state machine and implement its ISR for interrupt-driven implementations.

Remarks

- This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks)
- This function will never block or access any resources that may cause it to block.

Preconditions

The DRV_ETHPHY_Initialize routine must have been called for the specified Ethernet PHY driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_ETHPHY_Initialize

while (true)
{
    DRV_ETHPHY_Tasks (object);

    // Do other tasks
}
```

Function

void DRV_ETHPHY_Tasks(SYS_MODULE_OBJ object)

DRV_ETHPHY_HWConfigFlagsGet Function

Returns the current Ethernet PHY hardware MII/RMII and ALTERNATE/DEFAULT configuration flags.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_HWConfigFlagsGet(DRV_HANDLE handle, DRV_ETHPHY_CONFIG_FLAGS* pFlags);
```

Returns

DRV_ETHPHY_RES_OK - if the configuration flags successfully stored at pFlags DRV_ETHPHY_RESULT error code otherwise

Description

This function returns the current Ethernet PHY hardware MII/RMII and ALTERNATE/DEFAULT configuration flags from the Device Configuration Fuse bits.

Remarks

None.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_ETHPHY_Open)
pFlags	address to store the hardware configuration

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_HWConfigFlagsGet(DRV_HANDLE handle, DRV_ETHPHY_CONFIG_FLAGS* pFlags)

DRV_ETHPHY_Setup Function

Initializes Ethernet PHY configuration and set up procedure.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_Setup(DRV_HANDLE handle, DRV_ETHPHY_SETUP* pSetUp, TCPIP_ETH_OPEN_FLAGS*
pSetupFlags);
```

Returns

- DRV_ETHPHY_RES_PENDING operation has been scheduled successfully
- an DRV_ETHPHY_RESULT error code if the set up procedure failed.

Description

This function initializes the Ethernet PHY communication. It tries to detect the external Ethernet PHY, to read the capabilities and find a match with the requested features. Then, it programs the Ethernet PHY accordingly.

Remarks

PHY configuration may be a lengthy operation due to active negotiation that the PHY has to perform with the link party. The DRV_ETHPHY_ClientStatus will repeatedly return DRV_ETHPHY_CLIENT_STATUS_BUSY until the set up procedure is complete (unless an error detected at which an error code will be returned immediately).

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_Setup(DRV_HANDLE handle, DRV_ETHPHY_SETUP* pSetUp, TCPIP_ETH_OPEN_FLAGS* pSetupFlags)

b) Client Level Functions

DRV_ETHPHY_ClientStatus Function

Gets the current client-specific status the Ethernet PHY driver.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_CLIENT_STATUS DRV_ETHPHY_ClientStatus(DRV_HANDLE handle);
```

Returns

- DRV_ETHPHY_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the Ethernet PHY driver associated with the given handle.

Remarks

This function will not block for hardware access and will immediately return the current status.

This function has to be used to check that a driver operation has completed. It will return DRV_ETHPHY_CLIENT_STATUS_BUSY when an operation is in progress. It will return DRV_ETHPHY_CLIENT_STATUS_READY when the operation has completed.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE phyHandle;  // Returned from DRV_ETHPHY_Open
```

```
DRV_ETHPHY_CLIENT_STATUS phyClientStatus;

phyClientStatus = DRV_ETHPHY_ClientStatus(phyHandle);
if(DRV_ETHPHY_CLIENT_STATUS_ERROR >= phyClientStatus)
{
    // Handle the error
}
```

Function

DRV_ETHPHY_CLIENT_STATUS DRV_ETHPHY_ClientStatus(DRV_HANDLE handle)

DRV_ETHPHY_Close Function

Closes an opened instance of the Ethernet PHY driver.

Implementation: Dynamic

File

drv_ethphy.h

C

```
void DRV_ETHPHY_Close(DRV_HANDLE handle);
```

Returns

None

Description

This function closes an opened instance of the Ethernet PHY driver, invalidating the handle.

Remarks

- After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_ETHPHY_Open before the caller may use the driver again.
- Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_ETHPHY_Initialize routine must have been called for the specified Ethernet PHY driver instance.

DRV_ETHPHY_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_ETHPHY_Open

DRV_ETHPHY_Close(handle);
```

Function

void DRV_ETHPHY_Close(DRV_HANDLE handle)

DRV_ETHPHY_Open Function

Opens the specified Ethernet PHY driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_HANDLE DRV_ETHPHY_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

- valid open-instance handle if successful (a number identifying both the caller and the module instance).
- DRV_HANDLE_INVALID if an error occurs

Description

This function opens the specified Ethernet PHY driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_ETHPHY_Close routine is called.

This function will NEVER block waiting for hardware.

The intent parameter is not used. The PHY driver implements a non-blocking behavior.

Preconditions

The DRV_ETHPHY_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_ETHPHY_Open(DRV_ETHPHY_INDEX_0, 0);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Function

DRV_HANDLE DRV_ETHPHY_Open(const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT intent)

DRV_ETHPHY_Reset Function

Immediately resets the Ethernet PHY.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_Reset(DRV_HANDLE handle, bool waitComplete);
```

Returns

- DRV_ETHPHY_RES_PENDING for ongoing, in progress operation
- DRV_ETHPHY_RES_OPERATION_ERR - invalid parameter or operation in the current context

Description

This function immediately resets the Ethernet PHY, optionally waiting for a reset to complete.

Remarks

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

When operation is completed but failed, DRV_ETHPHY_ClientOperationResult will return:

- DRV_ETHPHY_RES_DTCT_ERR if the PHY failed to respond

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_Reset(DRV_HANDLE handle, bool waitComplete)

DRV_ETHPHY_ClientOperationAbort Function

Aborts a current client operation initiated by the Ethernet PHY driver.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_ClientOperationAbort(DRV_HANDLE handle);
```

Returns

- DRV_ETHPHY_RESULT value describing the current operation result: DRV_ETHPHY_RES_OK for success; operation has been aborted an DRV_ETHPHY_RESULT error code if the operation failed.

Description

Aborts a current client operation initiated by the Ethernet PHY driver.

Remarks

None

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid opened device handle.
- A driver operation was started

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_ClientOperationAbort(DRV_HANDLE handle)

DRV_ETHPHY_ClientOperationResult Function

Gets the result of a client operation initiated by the Ethernet PHY driver.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_ClientOperationResult(DRV_HANDLE handle);
```

Returns

- DRV_ETHPHY_RESULT value describing the current operation result: DRV_ETHPHY_RES_OK for success; operation has been completed successfully DRV_ETHPHY_RES_PENDING operation is in progress an DRV_ETHPHY_RESULT error code if the operation failed.

Description

Returns the result of a client operation initiated by the Ethernet PHY driver.

Remarks

This function will not block for hardware access and will immediately return the current status.

This function returns the result of the last driver operation. It will return DRV_ETHPHY_RES_PENDING if an operation is still in progress. Otherwise a DRV_ETHPHY_RESULT describing the operation outcome.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid opened device handle.
- A driver operation was started and completed

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_ClientOperationResult(DRV_HANDLE handle)

c) SMI/MIIM Functions

DRV_ETHPHY_SMIScanStatusGet Function

Gets the status of the SMI/MIIM scan data.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanStatusGet(DRV_HANDLE handle);
```

Returns

DRV_ETHPHY_RES_OPERATION_ERR - no scan operation currently in progress

DRV_ETHPHY_RES_OK - scan data is available

DRV_ETHPHY_RES_PENDING - scan data is not yet available

< 0 - an error has occurred and the operation could not be completed

Description

This function gets the status of the SMI/MIIM scan data.

Remarks

This operation is not supported when the PHY driver uses the MIIM driver for MIIM bus accesses. Use the DRV_MIIM for accessing the MIIM bus.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY
- DRV_ETHPHY_SMIScanStart() has been called.

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanStatusGet(DRV_HANDLE handle)

DRV_ETHPHY_SMIScanStop Function

Stops the scan of a previously requested SMI/MIIM register.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanStop(DRV_HANDLE handle);
```

Returns

DRV_ETHPHY_RES_OPERATION_ERR - no scan operation currently in progress

DRV_ETHPHY_RES_OK - the scan transaction has been stopped successfully < 0 - an error has occurred and the operation could not be completed

Description

This function stops the current scan of a SMI/MIIM register.

Remarks

This operation is not supported when the PHY driver uses the MIIM driver for MIIM bus accesses. Use the DRV_MIIM for accessing the MIIM bus.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY
- DRV_ETHPHY_SMIScanStart was called to start a scan

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanStop(DRV_HANDLE handle)

DRV_ETHPHY_SMIClockSet Function

Sets the SMI/MIIM interface clock.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIClockSet(DRV_HANDLE handle, uint32_t hostClock, uint32_t maxSMIClock);
```

Returns

DRV_ETHPHY_RES_HANDLE_ERR - passed in handle was invalid

DRV_ETHPHY_RES_OK - operation successful

< 0 - an error has occurred and the operation could not be completed

Description

This function sets SMI/MIIM interface clock base on host clock and maximum supported SMI/MIIM interface clock speed.

Remarks

This operation is not supported when the PHY driver uses the MIIM driver for MIIM bus accesses. Use the DRV_MIIM for accessing the MIIM bus.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIClockSet(DRV_HANDLE handle,

uint32_t hostClock,

uint32_t maxSMIClock)

DRV_ETHPHY_SMIScanStart Function

Starts the scan of a requested SMI/MIIM register.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanStart(DRV_HANDLE handle, unsigned int rIx);
```

Returns

DRV_ETHPHY_RES_PENDING - the scan transaction was initiated and is ongoing < 0 - an error has occurred and the operation could not be completed

Description

This function starts the scan of a requested SMI/MIIM register.

Remarks

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

However, the client status will always be DRV_ETHPHY_CLIENT_STATUS_BUSY and the client result will always show DRV_ETHPHY_RES_PENDING for as long as the scan is active. Use DRV_ETHPHY_SMIScanStop() to stop a scan in progress. Use DRV_ETHPHY_SMIScanStatusGet() to check is there is scan data available. Use DRV_ETHPHY_SMIScanDataGet() to retrieve the scan data.

This operation is not supported when the PHY driver uses the MIIM driver for MIIM bus accesses. Use the DRV_MIIM for accessing the MIIM bus.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanStart(DRV_HANDLE handle,
unsigned int rIx)

DRV_ETHPHY_SMIRead Function

Initiates a SMI/MIIM read transaction.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIRead(DRV_HANDLE handle, unsigned int rIx, uint16_t* pSmiRes, int phyAdd);
```

Returns

DRV_ETHPHY_RES_PENDING - the transaction was initiated and is ongoing < 0 - an error has occurred and the operation could not be completed

Description

This function initiates a SMI/MIIM read transaction for a given PHY register.

Remarks

In most situations the PHY address to be used for this function should be the one returned by DRV_ETHPHY_PhyAddressGet(). However this function allows using a different PHY address for advanced operation.

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

This operation is not supported when the PHY driver uses the MIIM driver for MIIM bus accesses. Use the DRV_MIIM for accessing the MIIM bus.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid opened device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIRead(DRV_HANDLE handle, unsigned int rIx, uint16_t* pSmiRes, int phyAdd)

DRV_ETHPHY_SMIScanDataGet Function

Gets the latest SMI/MIIM scan data result.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanDataGet(DRV_HANDLE handle, uint16_t* pScanRes);
```

Returns

DRV_ETHPHY_RES_OPERATION_ERR - no scan operation currently in progress

DRV_ETHPHY_RES_OK - scan data is available and stored at pScanRes DRV_ETHPHY_RES_PENDING - scan data is not yet available

< 0 - an error has occurred and the operation could not be completed

Description

This function gets the latest SMI/MIIM scan data result.

Remarks

This operation is not supported when the PHY driver uses the MIIM driver for MIIM bus accesses. Use the DRV_MIIM for accessing the MIIM bus.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY
- DRV_ETHPHY_SMIScanStart() has been called
- Data is available if DRV_ETHPHY_SMIScanStatusGet() previously returned DRV_ETHPHY_RES_OK

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIScanDataGet(DRV_HANDLE handle, uint16_t* pScanRes)

DRV_ETHPHY_SMIStatus Function

Returns the current status of the SMI/MIIM interface.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIStatus(DRV_HANDLE handle);
```

Returns

- DRV_ETHPHY_RES_BUSY - if the SMI/MIIM interface is busy

- DRV_ETHPHY_RES_OK - if the SMI/MIIM is not busy

< 0 - an error has occurred and the operation could not be completed

Description

This function checks if the SMI/MIIM interface is busy with a transaction.

Remarks

This function is info only and returns the momentary status of the SMI bus. Even if the bus is free there is no guarantee it will be free later on especially if the driver is on going some operation.

This operation is not supported when the PHY driver uses the MIIM driver for MIIM bus accesses. Use the DRV_MIIM for accessing the MIIM bus.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIStatus(DRV_HANDLE handle)

DRV_ETHPHY_SMIWrite Function

Initiates a SMI/MIIM write transaction.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_SMIWrite(DRV_HANDLE handle, unsigned int rIx, uint16_t wData, int phyAdd, bool waitComplete);
```

Returns

DRV_ETHPHY_RES_OK - the write transaction has been scheduled/completed successfully DRV_ETHPHY_RES_PENDING - the transaction was initiated and is ongoing < 0 - an error has occurred and the operation could not be completed

Description

This function initiates a SMI/MIIM write transaction for a given PHY register.

Remarks

In most situations the PHY address to be used for this function should be the one returned by DRV_ETHPHY_PhyAddressGet(). However this function allows using a different PHY address for advanced operation.

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

This operation is not supported when the PHY driver uses the MIIM driver for MIIM bus accesses. Use the DRV_MIIM for accessing the MIIM bus.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_SMIWrite(DRV_HANDLE handle, unsigned int rIx, uint16_t wData, int phyAdd, bool waitComplete)

d) Vendor Functions

DRV_ETHPHY_VendorDataGet Function

Returns the current value of the vendor data.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_VendorDataGet(DRV_HANDLE handle, uint32_t* pVendorData);
```

Returns

DRV_ETHPHY_RES_OK - if the vendor data is stored at the pVendorData address

DRV_ETHPHY_RES_HANDLE_ERR - handle error

Description

This function returns the current value of the vendor data. Each DRV_ETHPHY client object maintains data that could be used for vendor specific operations. This routine allows retrieving of the vendor specific data.

Remarks

The PHY driver will clear the vendor specific data before any call to a vendor specific routine. Otherwise the PHY driver functions do not touch this value.

The DRV_ETHPHY_VendorDataSet can be used for writing data into this field.

Currently only a 32 bit value is supported.

The function is intended for implementing vendor specific functions, like DRV_EXTPHY_MIIConfigure and DRV_EXTPHY_MDIXConfigure, that need a way of maintaining their own data and state machine.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_VendorDataGet(DRV_HANDLE handle, uint32_t* pVendorData)

DRV_ETHPHY_VendorDataSet Function

Returns the current value of the vendor data.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_VendorDataSet(DRV_HANDLE handle, uint32_t vendorData);
```

Returns

DRV_ETHPHY_RES_OK - if the vendor data is stored in the client object

DRV_ETHPHY_RES_HANDLE_ERR - handle error

Description

This function returns the current value of the vendor data. Each DRV_ETHPHY client object maintains data that could be used for vendor specific operations. This routine allows retrieving of the vendor specific data.

Remarks

The PHY driver will clear the vendor specific data before any call to a vendor specific routine. Otherwise the PHY driver functions do not touch this value.

The DRV_ETHPHY_VendorDataGet can be used for reading data into this field.

Currently only a 32 bit value is supported.

The function is intended for implementing vendor specific functions, like DRV_EXTPHY_MIIConfigure and DRV_EXTPHY_MDIXConfigure, that need a way of maintaining their own data and state machine.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_VendorDataSet(DRV_HANDLE handle, uint32_t vendorData)

DRV_ETHPHY_VendorSMIReadResultGet Function

Reads the result of a previous vendor initiated SMI read transfer with DRV_ETHPHY_VendorSMIReadStart.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_VendorSMIReadResultGet(DRV_HANDLE handle, uint16_t* pSmiRes);
```

Returns

DRV_ETHPHY_RES_OK - transaction complete and result deposited at pSmiRes.

DRV_ETHPHY_RES_PENDING - if the vendor transaction is still ongoing The call needs to be retried.

< 0 - some error and the DRV_EXTPHY_MIIConfigure/DRV_EXTPHY_MDIXConfigure has to return error to be aborted by the DRV_ETHPHY_Setup

Description

This function will return the data of a SMI read transfer.

Remarks

The function is intended for implementing vendor SMI transfers within DRV_EXTPHY_MIIConfigure and DRV_EXTPHY_MDIXConfigure.

It has to be called from within the DRV_EXTPHY_MIIConfigure or DRV_EXTPHY_MDIXConfigure functions (which are called, in turn, by the DRV_ETHPHY_Setup procedure) otherwise the call will fail.

The DRV_ETHPHY_RES_OK and DRV_ETHPHY_RES_PENDING significance is changed from the general driver API.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup is in progress and configures the PHY
- The vendor implementation of the DRV_EXTPHY_MIIConfigure/DRV_EXTPHY_MDIXConfigure is running and a SMI transfer is needed
- DRV_ETHPHY_VendorSMIReadStart should have been called to initiate a transfer

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_VendorSMIReadResultGet(DRV_HANDLE handle, uint16_t* pSmiRes)

DRV_ETHPHY_VendorSMIReadStart Function

Starts a vendor SMI read transfer. Data will be available with DRV_ETHPHY_VendorSMIReadResultGet.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_VendorSMIReadStart(DRV_HANDLE handle, uint16_t rIx, int phyAddress);
```

Returns

DRV_ETHPHY_RES_OK - the vendor transaction is started DRV_ETHPHY_VendorSMIReadResultGet() needs to be called for the transaction to complete and to retrieve the result

DRV_ETHPHY_RES_PENDING - the SMI bus is busy and the call needs to be retried

< 0 - some error and the DRV_EXTPHY_MIIConfigure/DRV_EXTPHY_MDIXConfigure has to return error to be aborted by the DRV_ETHPHY_Setup

Description

This function will start a SMI read transfer.

Remarks

The function is intended for implementing vendor SMI transfers within DRV_EXTPHY_MIIConfigure and DRV_EXTPHY_MDIXConfigure.

It has to be called from within the DRV_EXTPHY_MIIConfigure or DRV_EXTPHY_MDIXConfigure functions (which are called, in turn, by the DRV_ETHPHY_Setup procedure) otherwise the call will fail.

The DRV_ETHPHY_RES_OK and DRV_ETHPHY_RES_PENDING significance is changed from the general driver API.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup is in progress and configures the PHY
- The vendor implementation of the DRV_EXTPHY_MIIConfigure/DRV_EXTPHY_MDIXConfigure is running and a SMI transfer is needed

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_VendorSMIReadStart(DRV_HANDLE handle, uint16_t rIx, int phyAddress)

DRV_ETHPHY_VendorSMIWriteStart Function

Starts a vendor SMI write transfer.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_VendorSMIWriteStart(DRV_HANDLE handle, uint16_t rIx, uint16_t wData, int
phyAddress);
```

Returns

DRV_ETHPHY_RES_OK - if the vendor SMI write transfer is started

DRV_ETHPHY_RES_PENDING - the SMI bus was busy and the call needs to be retried

< 0 - some error and the DRV_EXTPHY_MIIConfigure/DRV_EXTPHY_MDIXConfigure has to return error to be aborted by the DRV_ETHPHY_Setup

Description

This function will start a SMI write transfer.

Remarks

The function is intended for implementing vendor SMI transfers within DRV_EXTPHY_MIIConfigure and DRV_EXTPHY_MDIXConfigure.

It has to be called from within the DRV_EXTPHY_MIIConfigure or DRV_EXTPHY_MDIXConfigure functions (which are called, in turn, by the DRV_ETHPHY_Setup procedure) otherwise the call will fail.

The DRV_ETHPHY_RES_OK and DRV_ETHPHY_RES_PENDING significance is changed from the general driver API.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup is in progress and configures the PHY
- The vendor implementation of the DRV_EXTPHY_MIIConfigure/DRV_EXTPHY_MDIXConfigure is running and a SMI transfer is needed

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_VendorSMIWriteStart(DRV_HANDLE handle, uint16_t rIx, uint16_t wData, int phyAddress)

e) Other Functions

DRV_ETHPHY_LinkStatusGet Function

Returns the current link status.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_LinkStatusGet(DRV_HANDLE handle, DRV_ETHPHY_INTERFACE_INDEX portIndex,
DRV_ETHPHY_LINK_STATUS* pLinkStat, bool refresh);
```

Returns

- DRV_ETHPHY_RES_PENDING for ongoing, in progress operation
- an DRV_ETHPHY_RESULT error code if the link status get procedure failed.

Description

This function returns the current link status.

Remarks

This function reads the Ethernet PHY to get current link status. If refresh is specified then, if the link is down a second read will be performed to return the current link status.

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_LinkStatusGet(DRV_HANDLE handle, DRV_ETHPHY_LINK_STATUS* pLinkStat, bool refresh)

DRV_ETHPHY_NegotiationIsComplete Function

Returns the results of a previously initiated Ethernet PHY negotiation.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_NegotiationIsComplete(DRV_HANDLE handle, DRV_ETHPHY_INTERFACE_INDEX portIndex,
bool waitComplete);
```

Returns

- DRV_ETHPHY_RES_PENDING operation is ongoing
- an DRV_ETHPHY_RESULT error code if the procedure failed.

Description

This function returns the results of a previously initiated Ethernet PHY negotiation.

Remarks

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

When operation is completed but negotiation has failed, DRV_ETHPHY_ClientOperationResult will return:

- DRV_ETHPHY_RES_NEGOTIATION_INACTIVE if no negotiation in progress
- DRV_ETHPHY_RES_NEGOTIATION_NOT_STARTED if negotiation not yet started yet (means time out if waitComplete was requested)
- DRV_ETHPHY_RES_NEGOTIATION_ACTIVE if negotiation ongoing

(means time out if waitComplete was requested).

See also DRV_ETHPHY_NegotiationResultGet.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY
- DRV_ETHPHY_RestartNegotiation should have been called.

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_NegotiationIsComplete(DRV_HANDLE handle, bool waitComplete)

DRV_ETHPHY_NegotiationResultGet Function

Returns the result of a completed negotiation.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_NegotiationResultGet(DRV_HANDLE handle, DRV_ETHPHY_INTERFACE_INDEX portIndex,
DRV_ETHPHY_NEGOTIATION_RESULT* pNegResult);
```

Returns

- DRV_ETHPHY_RES_PENDING operation is ongoing
- an DRV_ETHPHY_RESULT error code if the procedure failed.

Description

This function returns the PHY negotiation data gathered after a completed negotiation.

Remarks

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

When operation is completed but negotiation has failed, DRV_ETHPHY_ClientOperationResult will return:

- DRV_ETHPHY_RES_NEGOTIATION_INACTIVE if no negotiation in progress
- DRV_ETHPHY_RES_NEGOTIATION_NOT_STARTED if negotiation not yet started yet (means time out if waitComplete was requested)
- DRV_ETHPHY_RES_NEGOTIATION_ACTIVE if negotiation ongoing

The returned value for the negotiation flags is valid only if the negotiation was completed successfully.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY
- DRV_ETHPHY_RestartNegotiation, and DRV_ETHPHY_NegotiationIsComplete should have been called.

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_NegotiationResultGet(DRV_HANDLE handle, DRV_ETHPHY_NEGOTIATION_RESULT* pNegResult)

DRV_ETHPHY_PhyAddressGet Function

Returns the PHY address.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_PhyAddressGet(DRV_HANDLE handle, DRV_ETHPHY_INTERFACE_INDEX portIndex, int* pPhyAddress);
```

Returns

DRV_ETHPHY_RES_OK - operation successful and the PHY address stored at

DRV_ETHPHY_RES_HANDLE_ERR - passed in handle was invalid pPhyAddress

Description

This function returns the current PHY address as set by the DRV_ETHPHY_Setup procedure.

Remarks

None.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_PhyAddressGet(DRV_HANDLE handle, int* pPhyAddress);

DRV_ETHPHY_RestartNegotiation Function

Restarts auto-negotiation of the Ethernet PHY link.

Implementation: Dynamic

File

drv_ethphy.h

C

```
DRV_ETHPHY_RESULT DRV_ETHPHY_RestartNegotiation(DRV_HANDLE handle, DRV_ETHPHY_INTERFACE_INDEX portIndex);
```

Returns

- DRV_ETHPHY_RES_PENDING operation has been scheduled successfully
- an DRV_ETHPHY_RESULT error code if the procedure failed.

Description

This function restarts auto-negotiation of the Ethernet PHY link.

Remarks

Use DRV_ETHPHY_ClientStatus() and DRV_ETHPHY_ClientOperationResult() to check when the operation was completed and its outcome.

Preconditions

- The DRV_ETHPHY_Initialize routine must have been called.
- DRV_ETHPHY_Open must have been called to obtain a valid device handle.
- DRV_ETHPHY_Setup must have been called to properly configure the PHY

Example

Function

DRV_ETHPHY_RESULT DRV_ETHPHY_RestartNegotiation(DRV_HANDLE handle)

f) Data Types and Constants

DRV_ETHPHY_CLIENT_STATUS Enumeration

Identifies the client-specific status of the Ethernet PHY driver.

File

drv_ethphy.h

C

```
typedef enum {
  DRV_ETHPHY_CLIENT_STATUS_ERROR,
  DRV_ETHPHY_CLIENT_STATUS_CLOSED,
  DRV_ETHPHY_CLIENT_STATUS_BUSY,
  DRV_ETHPHY_CLIENT_STATUS_READY
} DRV_ETHPHY_CLIENT_STATUS;
```

Members

Members	Description
DRV_ETHPHY_CLIENT_STATUS_ERROR	Unspecified error condition
DRV_ETHPHY_CLIENT_STATUS_CLOSED	Client is not open
DRV_ETHPHY_CLIENT_STATUS_BUSY	An operation is currently in progress
DRV_ETHPHY_CLIENT_STATUS_READY	Up and running, no operations running

Description

Ethernet PHY Driver Client Status

This enumeration identifies the client-specific status of the Ethernet PHY driver.

Remarks

None.

DRV_ETHPHY_INIT Structure

Contains all the data necessary to initialize the Ethernet PHY device.

File

drv_ethphy.h

C

```
struct DRV_ETHPHY_INIT {
  SYS_MODULE_INIT moduleInit;
  uintptr_t ethphyId;
  uint16_t phyAddress;
  DRV_ETHPHY_CONFIG_FLAGS phyFlags;
  const DRV_ETHPHY_OBJECT* pPhyObject;
  DRV_ETHPHY_RESET_FUNCTION resetFunction;
  const struct DRV_MIIM_OBJECT_BASE* pMiimObject;
  const struct DRV_MIIM_INIT* pMiimInit;
  SYS_MODULE_INDEX miimIndex;
};
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
uintptr_t ethphyId;	Identifies peripheral (PLIB-level) ID
uint16_t phyAddress;	PHY address, as configured on the board. All PHYs respond to address 0
DRV_ETHPHY_CONFIG_FLAGS phyFlags;	PHY configuration
const DRV_ETHPHY_OBJECT* pPhyObject;	Non-volatile pointer to the PHY object providing vendor functions for this PHY
DRV_ETHPHY_RESET_FUNCTION resetFunction;	Function to be called when the PHY is reset/initialized. Could be NULL if no special reset functionality needed - default
const struct DRV_MIIM_OBJECT_BASE* pMiimObject;	Non-volatile pointer to the DRV_MIIM object providing MIIM access for this PHY Could be NULL if the MIIM driver is not used
const struct DRV_MIIM_INIT* pMiimInit;	Non-volatile pointer to the DRV_MIIM initialization data Could be NULL if the MIIM driver is not used
SYS_MODULE_INDEX miimIndex;	MIIM module index to be used Not needed if the MIIM driver is not used

Description

Ethernet PHY Device Driver Initialization Data

This data structure contains all the data necessary to initialize the Ethernet PHY device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_ETHPHY_Initialize routine.

DRV_ETHPHY_NEGOTIATION_RESULT Structure

Contains all the data necessary to get the Ethernet PHY negotiation result

File

drv_ethphy.h

C

```
typedef struct {
  DRV_ETHPHY_LINK_STATUS linkStatus;
  TCPIP_ETH_OPEN_FLAGS linkFlags;
  TCPIP_ETH_PAUSE_TYPE pauseType;
} DRV_ETHPHY_NEGOTIATION_RESULT;
```

Members

Members	Description
DRV_ETHPHY_LINK_STATUS linkStatus;	link status after a completed negotiation

TCPIP_ETH_OPEN_FLAGS linkFlags;	the negotiation result flags
TCPIP_ETH_PAUSE_TYPE pauseType;	pause type supported by the link partner

Description

Ethernet PHY Device Driver Negotiation result Data

Contains all the data necessary to get the Ethernet PHY negotiation result

Remarks

A pointer to a structure of this format must be passed into the DRV_ETHPHY_NegotiationResultGet routine.

DRV_ETHPHY_SETUP Structure

Contains all the data necessary to set up the Ethernet PHY device.

File

drv_ethphy.h

C

```
typedef struct {
  int phyAddress;
  TCPIP_ETH_OPEN_FLAGS openFlags;
  DRV_ETHPHY_CONFIG_FLAGS configFlags;
  TCPIP_ETH_PAUSE_TYPE macPauseType;
  DRV_ETHPHY_RESET_FUNCTION resetFunction;
} DRV_ETHPHY_SETUP;
```

Members

Members	Description
int phyAddress;	the address the PHY is configured for
TCPIP_ETH_OPEN_FLAGS openFlags;	the capability flags: FD/HD, 100/100Mbps, etc.
DRV_ETHPHY_CONFIG_FLAGS configFlags;	configuration flags: MII/RMII, I/O setup
TCPIP_ETH_PAUSE_TYPE macPauseType;	MAC requested pause type
DRV_ETHPHY_RESET_FUNCTION resetFunction;	If ! NULL, function to be called when the PHY is reset/initialized

Description

Ethernet PHY Device Driver Set up Data

This data structure contains all the data necessary to configure the Ethernet PHY device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_ETHPHY_Setup routine.

DRV_ETHPHY_VENDOR_MDIX_CONFIGURE Type

Pointer to function that configures the MDIX mode for the Ethernet PHY.

File

drv_ethphy.h

C

```
typedef DRV_ETHPHY_RESULT (* DRV_ETHPHY_VENDOR_MDIX_CONFIGURE)(const struct DRV_ETHPHY_OBJECT_BASE_TYPE*
pBaseObj, DRV_HANDLE handle, TCPIP_ETH_OPEN_FLAGS oFlags);
```

Returns

- DRV_ETHPHY_RES_OK - if success, operation complete
- DRV_ETHPHY_RES_PENDING - if function needs to be called again

< 0 - on failure: configuration not supported or some other error

Description

Pointer To Function: typedef DRV_ETHPHY_RESULT (* DRV_ETHPHY_VENDOR_MDIX_CONFIGURE) (const struct DRV_ETHPHY_OBJECT_BASE_TYPE* pBaseObj, DRV_HANDLE handle, TCPIP_ETH_OPEN_FLAGS oFlags);

This type describes a pointer to a function that configures the MDIX mode for the Ethernet PHY. This configuration function is PHY specific and

every PHY driver has to provide their own implementation.

Remarks

The PHY driver consists of 2 modules:

- the main/base PHY driver which uses standard IEEE PHY registers
- the vendor specific functionality

This function provides vendor specific functionality. Every PHY driver has to expose this vendor specific function as part of its interface.

Traditionally the name used for this function is DRV_EXTPHY_MDIXConfigure but any name can be used.

The function can use all the vendor specific functions to store/retrieve specific data or start SMI transactions (see Vendor Interface Routines).

The function should not block but return DRV_ETHPHY_RES_PENDING if waiting for SMI transactions.

Preconditions

Communication to the PHY should have been established.

DRV_ETHPHY_VENDOR_MII_CONFIGURE Type

Pointer to function to configure the Ethernet PHY in one of the MII/RMII operation modes.

File

drv_ethphy.h

C

```
typedef DRV_ETHPHY_RESULT (* DRV_ETHPHY_VENDOR_MII_CONFIGURE)(const struct DRV_ETHPHY_OBJECT_BASE_TYPE*
pBaseObj, DRV_HANDLE handle, DRV_ETHPHY_CONFIG_FLAGS cFlags);
```

Returns

- DRV_ETHPHY_RES_OK - if success, operation complete
- DRV_ETHPHY_RES_PENDING - if function needs to be called again

< 0 - on failure: configuration not supported or some other error

Description

Pointer To Function: typedef DRV_ETHPHY_RESULT (* DRV_ETHPHY_VENDOR_MII_CONFIGURE) (const struct DRV_ETHPHY_OBJECT_BASE_TYPE* pBaseObj, DRV_HANDLE handle, DRV_ETHPHY_CONFIG_FLAGS cFlags);

This type describes a pointer to a function that configures the Ethernet PHY in one of the MII/RMII operation modes. This configuration function is PHY specific and every PHY driver has to provide their own implementation.

Remarks

The PHY driver consists of 2 modules:

- the main/base PHY driver which uses standard IEEE PHY registers
- the vendor specific functionality

This function provides vendor specific functionality. Every PHY driver has to expose this vendor specific function as part of its interface.

Traditionally the name used for this function is DRV_EXTPHY_MIIConfigure but any name can be used.

The PHY driver will call the vendor set up functions after the communication to the PHY has been established.

The function can use all the vendor specific functions to store/retrieve specific data or start SMI transactions (see Vendor Interface Routines).

The function should not block but return DRV_ETHPHY_RES_PENDING if waiting for SMI transactions.

Preconditions

Communication to the PHY should have been established.

DRV_ETHPHY_VENDOR_SMI_CLOCK_GET Type

Pointer to a function to return the SMI/MIIM maximum clock speed in Hz of the Ethernet PHY.

File

drv_ethphy.h

C

```
typedef unsigned int (* DRV_ETHPHY_VENDOR_SMI_CLOCK_GET)(const struct DRV_ETHPHY_OBJECT_BASE_TYPE*
pBaseObj, DRV_HANDLE handle);
```

Returns

The maximum SMI/MIIM clock speed as an unsigned integer.

Description

Pointer to Function: typedef unsigned int (* DRV_ETHPHY_VENDOR_SMI_CLOCK_GET) (const struct DRV_ETHPHY_OBJECT_BASE_TYPE* pBaseObj, DRV_HANDLE handle);

This type describes a pointer to a function that returns the SMI/MIIM maximum clock speed in Hz of the Ethernet PHY. This configuration function is PHY specific and every PHY driver has to provide their own implementation.

Remarks

The PHY driver consists of 2 modules:

- the main/base PHY driver which uses standard IEEE PHY registers
- the vendor specific functionality

This function provides vendor specific functionality. Every PHY driver has to expose this vendor specific function as part of its interface.

This value is PHY specific. All PHYs are requested to support 2.5 MHz.

Traditionally the name used for this function is DRV_EXTPHY_SMIClockGet but any name can be used.

The PHY driver will call the vendor set up functions after the communication to the PHY has been established.

The function should not block but return immediately. The function cannot start SMI transactions and cannot use the vendor specific functions to store/retrieve specific data (see Vendor Interface Routines).

Preconditions

Communication to the PHY should have been established.

DRV_ETHPHY_INDEX_0 Macro

Ethernet PHY driver index definitions.

File

drv_ethphy.h

C

```
#define DRV_ETHPHY_INDEX_0 0
```

Description

Ethernet PHY Driver Module Index Numbers

These constants provide the Ethernet PHY driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_ETHPHY_Initialize and DRV_ETHPHY_Open routines to identify the driver instance in use.

DRV_ETHPHY_INDEX_1 Macro

File

drv_ethphy.h

C

```
#define DRV_ETHPHY_INDEX_1 1
```

Description

This is macro DRV_ETHPHY_INDEX_1.

DRV_ETHPHY_INDEX_COUNT Macro

Number of valid Ethernet PHY driver indices.

File

drv_ethphy.h

C

```
#define DRV_ETHPHY_INDEX_COUNT 1
```

Description

Ethernet PHY Driver Module Index Count

This constant identifies the number of valid Ethernet PHY driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from part-specific header files defined as part of the peripheral libraries.

DRV_ETHPHY_LINK_STATUS Enumeration

Defines the possible status flags of PHY Ethernet link.

File

drv_ethphy.h

C

```
typedef enum {
  DRV_ETHPHY_LINK_ST_DOWN,
  DRV_ETHPHY_LINK_ST_UP,
  DRV_ETHPHY_LINK_ST_LP_NEG_UNABLE,
  DRV_ETHPHY_LINK_ST_REMOTE_FAULT,
  DRV_ETHPHY_LINK_ST_PDF,
  DRV_ETHPHY_LINK_ST_LP_PAUSE,
  DRV_ETHPHY_LINK_ST_LP_ASM_DIR,
  DRV_ETHPHY_LINK_ST_NEG_TMO,
  DRV_ETHPHY_LINK_ST_NEG_FATAL_ERR
} DRV_ETHPHY_LINK_STATUS;
```

Members

Members	Description
DRV_ETHPHY_LINK_ST_DOWN	No connection to the LinkPartner
DRV_ETHPHY_LINK_ST_UP	Link is up
DRV_ETHPHY_LINK_ST_LP_NEG_UNABLE	LP non negotiation able
DRV_ETHPHY_LINK_ST_REMOTE_FAULT	LP fault during negotiation
DRV_ETHPHY_LINK_ST_PDF	Parallel Detection Fault encountered (when DRV_ETHPHY_LINK_ST_LP_NEG_UNABLE)
DRV_ETHPHY_LINK_ST_LP_PAUSE	LP supports symmetric pause
DRV_ETHPHY_LINK_ST_LP_ASM_DIR	LP supports asymmetric TX/RX pause operation
DRV_ETHPHY_LINK_ST_NEG_TMO	LP not there
DRV_ETHPHY_LINK_ST_NEG_FATAL_ERR	An unexpected fatal error occurred during the negotiation

Description

Ethernet PHY Device Link Status Codes

This enumeration defines the flags describing the status of the PHY Ethernet link.

Remarks

Multiple flags can be set.

DRV_ETHPHY_CONFIG_FLAGS Enumeration

Defines configuration options for the Ethernet PHY.

File

drv_ethphy.h

C

```
typedef enum {
  DRV_ETHPHY_CFG_RMII,
  DRV_ETHPHY_CFG_MII,
```

```
  DRV_ETHPHY_CFG_ALTERNATE,
  DRV_ETHPHY_CFG_DEFAULT,
  DRV_ETHPHY_CFG_AUTO
} DRV_ETHPHY_CONFIG_FLAGS;
```

Members

Members	Description
DRV_ETHPHY_CFG_RMII	RMII data interface in configuration fuses.
DRV_ETHPHY_CFG_MII	MII data interface in configuration fuses.
DRV_ETHPHY_CFG_ALTERNATE	Configuration fuses is ALT
DRV_ETHPHY_CFG_DEFAULT	Configuration fuses is DEFAULT
DRV_ETHPHY_CFG_AUTO	Use the fuses configuration to detect if you are RMII/MII and ALT/DEFAULT configuration

Description

Ethernet PHY Configuration Flags

This enumeration defines configuration options for the Ethernet PHY. Used by: DRV_ETHPHY_MIIConfigure, DRV_ETHPHY_INIT structure, DRV_ETHPHY_Setup, Returned by: DRV_ETHPHY_HWConfigFlagsGet

DRV_ETHPHY_OBJECT Structure

Identifies the interface of a Ethernet PHY vendor driver.

File

drv_ethphy.h

C

```
typedef struct {
  DRV_ETHPHY_VENDOR_MII_CONFIGURE miiConfigure;
  DRV_ETHPHY_VENDOR_MDIX_CONFIGURE mdixConfigure;
  DRV_ETHPHY_VENDOR_SMI_CLOCK_GET smiClockGet;
  DRV_ETHPHY_VENDOR_WOL_CONFIGURE wolConfigure;
} DRV_ETHPHY_OBJECT;
```

Members

Members	Description
DRV_ETHPHY_VENDOR_MII_CONFIGURE miiConfigure;	PHY driver function to configure the operation mode: MII/RMII
DRV_ETHPHY_VENDOR_MDIX_CONFIGURE mdixConfigure;	PHY driver function to configure the MDIX mode
DRV_ETHPHY_VENDOR_SMI_CLOCK_GET smiClockGet;	PHY driver function to get the SMI clock rate
DRV_ETHPHY_VENDOR_WOL_CONFIGURE wolConfigure;	PHY driver function to configure the WOL functionality

Description

Ethernet PHY Driver Vendor Object

This data structure identifies the required interface of the Ethernet PHY driver. Any PHY vendor driver has to export this interface.

Remarks

The PHY driver consists of 2 modules:

- the main/base PHY driver which uses standard IEEE PHY registers
- the vendor specific functionality

This object provides vendor specific functionality. Every PHY driver has to expose this vendor specific functionality as part of its interface.

DRV_ETHPHY_VENDOR_WOL_CONFIGURE Type

Pointer to a function to configure the PHY WOL functionality

File

drv_ethphy.h

C

```
typedef void (* DRV_ETHPHY_VENDOR_WOL_CONFIGURE)(const struct DRV_ETHPHY_OBJECT_BASE_TYPE* pBaseObj,
DRV_HANDLE handle, unsigned char bAddr[]);
```

Returns

None

Description

Pointer to Function: typedef void (* DRV_ETHPHY_VENDOR_WOL_CONFIGURE) (const struct DRV_ETHPHY_OBJECT_BASE_TYPE* pBaseObj, DRV_HANDLE handle, unsigned char bAddr[]);

This type describes a pointer to a function that configures the PHY WOL functionality of the Ethernet PHY. Configures the WOL of the PHY with a Source MAC address or a 6 byte magic packet mac address.

This configuration function is PHY specific and every PHY driver has to provide their own implementation.

Remarks

The PHY driver consists of 2 modules:

- the main/base PHY driver which uses standard IEEE PHY registers
- the vendor specific functionality

This function provides vendor specific functionality. Every PHY driver has to expose this vendor specific function as part of its interface.

Traditionally the name used for this function is DRV_EXTPHY_WOLConfiguration but any name can be used.

The PHY driver will call the vendor set up functions after the communication to the PHY has been established.

The function can use all the vendor specific functions to store/retrieve specific data or start SMI transactions (see Vendor Interface Routines).

The function should not block but return DRV_ETHPHY_RES_PENDING if waiting for SMI transactions.

This feature is not currently supported for all PHYs.

Preconditions

Communication to the PHY should have been established.

DRV_ETHPHY_OBJECT_BASE Structure

Identifies the base interface of a Ethernet PHY driver.

File

drv_ethphy.h

C

```
typedef struct DRV_ETHPHY_OBJECT_BASE_TYPE {
  SYS_MODULE_OBJ (* DRV_ETHPHY_Initialize)(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const
init);
  void (* DRV_ETHPHY_Reinitialize)(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
  void (* DRV_ETHPHY_Deinitialize)(SYS_MODULE_OBJ object);
  SYS_STATUS (* DRV_ETHPHY_Status)(SYS_MODULE_OBJ object);
  void (* DRV_ETHPHY_Tasks)(SYS_MODULE_OBJ object);
  DRV_HANDLE (* DRV_ETHPHY_Open)(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
  void (* DRV_ETHPHY_Close)(DRV_HANDLE handle);
  DRV_ETHPHY_CLIENT_STATUS (* DRV_ETHPHY_ClientStatus)(DRV_HANDLE handle);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_ClientOperationResult)(DRV_HANDLE handle);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_ClientOperationAbort)(DRV_HANDLE handle);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_SMIRead)(DRV_HANDLE handle, unsigned int rIx, uint16_t* pSmiRes, int
phyAdd);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_SMIWrite)(DRV_HANDLE handle, unsigned int rIx, uint16_t wData, int
phyAdd, bool waitComplete);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_SMIScanStart)(DRV_HANDLE handle, unsigned int rIx);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_SMIScanStop)(DRV_HANDLE handle);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_SMIScanStatusGet)(DRV_HANDLE handle);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_SMIScanDataGet)(DRV_HANDLE handle, uint16_t* pScanRes);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_SMIStatus)(DRV_HANDLE handle);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_SMIClockSet)(DRV_HANDLE handle, uint32_t hostClock, uint32_t maxSMIClock);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_PhyAddressGet)(DRV_HANDLE handle, DRV_ETHPHY_INTERFACE_INDEX portIndex,
int* pPhyAddress);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_Setup)(DRV_HANDLE handle, DRV_ETHPHY_SETUP* pSetUp, TCPIP_ETH_OPEN_FLAGS*
pSetupFlags);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_RestartNegotiation)(DRV_HANDLE handle, DRV_ETHPHY_INTERFACE_INDEX
portIndex);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_HWConfigFlagsGet)(DRV_HANDLE handle, DRV_ETHPHY_CONFIG_FLAGS* pFlags);
```

```
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_NegotiationIsComplete)(DRV_HANDLE handle, DRV_ETHPHY_INTERFACE_INDEX
portIndex, bool waitComplete);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_NegotiationResultGet)(DRV_HANDLE handle, DRV_ETHPHY_INTERFACE_INDEX
portIndex, DRV_ETHPHY_NEGOTIATION_RESULT* pNegResult);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_LinkStatusGet)(DRV_HANDLE handle, DRV_ETHPHY_INTERFACE_INDEX portIndex,
DRV_ETHPHY_LINK_STATUS* pLinkStat, bool refresh);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_Reset)(DRV_HANDLE handle, bool waitComplete);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_VendorDataGet)(DRV_HANDLE handle, uint32_t* pVendorData);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_VendorDataSet)(DRV_HANDLE handle, uint32_t vendorData);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_VendorSMIReadStart)(DRV_HANDLE handle, uint16_t rIx, int phyAddress);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_VendorSMIReadResultGet)(DRV_HANDLE handle, uint16_t* pSmiRes);
  DRV_ETHPHY_RESULT (* DRV_ETHPHY_VendorSMIWriteStart)(DRV_HANDLE handle, uint16_t rIx, uint16_t wData, int
phyAddress);
} DRV_ETHPHY_OBJECT_BASE;
```

Description

Ethernet PHY Driver Base Object

This data structure identifies the required interface of the Ethernet PHY driver. Any dynamic PHY driver has to export this interface.

Remarks

The PHY driver consists of 2 modules:

- the main/base PHY driver which uses standard IEEE PHY registers
- the vendor specific functionality

This object provides the base functionality. Every dynamic PHY driver has to expose this basic functionality as part of its interface.

See above the description of each function that's part of the base PHY driver.

DRV_ETHPHY_RESET_FUNCTION Type

Pointer to a function to perform an additional PHY reset

File

drv_ethphy.h

C

```
typedef void (* DRV_ETHPHY_RESET_FUNCTION)(const struct DRV_ETHPHY_OBJECT_BASE_TYPE* pBaseObj);
```

Returns

None

Description

Pointer to Function: typedef void (* DRV_ETHPHY_RESET_FUNCTION) (const struct DRV_ETHPHY_OBJECT_BASE_TYPE* pBaseObj);

This type describes a pointer to a function that is called by the driver before starting the detection and initialization process to the PHY - as a result of the DRV_ETHPHY_Setup call.

Remarks

The PHY driver will call this function as part of its detection and initialization procedure. It can be used for implementing extra steps that the user needs, before the driver starts talking to the PHY. For example, if a hard reset needs to be applied to the PHY.

The function should be short and not block. It is meant just for short I/O operations, not for lengthy processing.

Preconditions

None

DRV_ETHPHY_RESULT Enumeration

Defines the possible results of Ethernet operations that can succeed or fail

File

drv_ethphy.h

C

```
typedef enum {
} DRV_ETHPHY_RESULT;
```

Description

Ethernet PHY Driver Operation Result *

PHY Driver Operation Result Codes

This enumeration defines the possible results of any of the PHY driver operations that have the possibility of failing. This result should be checked to ensure that the operation achieved the desired result.

DRV_ETHPHY_USE_DRV_MIIM Macro

Defines the way the PHY driver accesses the MIIM bus to communicate with the PHY.

File

drv_ethphy_config.h

C

```
#define DRV_ETHPHY_USE_DRV_MIIM true
```

Description

Ethernet MIIM access configuration

Defines the way the PHY driver accesses the MIIM bus to communicate with the PHY:

- either using direct access to the ETH plibs
- using the MIIM driver - preferred way

Remarks

Using the MIIM driver to perform MIIM bus operations is more versatile and preferred.

DRV_ETHPHY_INTERFACE_INDEX Enumeration

Defines the index type for a PHY interface.

File

drv_ethphy.h

C

```
typedef enum {
  DRV_ETHPHY_INF_IDX_ALL_EXTERNAL,
  DRV_ETHPHY_INF_IDX_PORT_0,
  DRV_ETHPHY_INF_IDX_PORT_1,
  DRV_ETHPHY_INF_IDX_PORT_2,
  DRV_ETHPHY_INF_IDX_PORT_3,
  DRV_ETHPHY_INF_IDX_PORT_4,
  DRV_ETHPHY_INF_IDX_PORT_5
} DRV_ETHPHY_INTERFACE_INDEX;
```

Members

Members	Description
DRV_ETHPHY_INF_IDX_ALL_EXTERNAL	All External Interfaces
DRV_ETHPHY_INF_IDX_PORT_0	Port 0 interface
DRV_ETHPHY_INF_IDX_PORT_1	Port 1 interface
DRV_ETHPHY_INF_IDX_PORT_2	Port 2 interface
DRV_ETHPHY_INF_IDX_PORT_3	Port 3 interface
DRV_ETHPHY_INF_IDX_PORT_4	Port 4 interface
DRV_ETHPHY_INF_IDX_PORT_5	Port 5 interface

Description

Ethernet PHY Interface Index

This enumeration defines the index type supported by the PHY Used by: DRV_ETHPHY_PhyAddressGet, DRV_ETHPHY_RestartNegotiation, DRV_ETHPHY_NegotiationIsComplete, DRV_ETHPHY_LinkStatusGet

DRV_ETHPHY_INTERFACE_TYPE Enumeration

Defines the type of interface a PHY supports.

File

drv_ethphy.h

C

```
typedef enum {
  DRV_ETHPHY_INF_TYPE_EXTERNAL,
  DRV_ETHPHY_INF_TYPE_INTERNAL,
  DRV_ETHPHY_INF_TYPE_NOT_SUPPORTED
} DRV_ETHPHY_INTERFACE_TYPE;
```

Members

Members	Description
DRV_ETHPHY_INF_TYPE_EXTERNAL	External Interface
DRV_ETHPHY_INF_TYPE_INTERNAL	Internal Interface
DRV_ETHPHY_INF_TYPE_NOT_SUPPORTED	Not Supported

Description

Ethernet PHY Interface Type

This enumeration defines the type of interface supported by the PHY Returned by: DRV_ETHPHY_GetInterfaceType

Files

Files

Name	Description
drv_ethphy.h	Ethernet ETHPHY Device Driver Interface File
drv_ethphy_config.h	Ethernet PHY driver configuration definitions template.

Description

This section lists the source and header files used by the Ethernet PHY Driver Library.

drv_ethphy.h

Ethernet ETHPHY Device Driver Interface File

Enumerations

	Name	Description
	DRV_ETHPHY_CLIENT_STATUS	Identifies the client-specific status of the Ethernet PHY driver.
	DRV_ETHPHY_CONFIG_FLAGS	Defines configuration options for the Ethernet PHY.
	DRV_ETHPHY_INTERFACE_INDEX	Defines the index type for a PHY interface.
	DRV_ETHPHY_INTERFACE_TYPE	Defines the type of interface a PHY supports.
	DRV_ETHPHY_LINK_STATUS	Defines the possible status flags of PHY Ethernet link.
	DRV_ETHPHY_RESULT	Defines the possible results of Ethernet operations that can succeed or fail

Functions

	Name	Description
	DRV_ETHPHY_ClientOperationAbort	Aborts a current client operation initiated by the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_ClientOperationResult	Gets the result of a client operation initiated by the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_ClientStatus	Gets the current client-specific status the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_Close	Closes an opened instance of the Ethernet PHY driver. **Implementation:** Dynamic

	DRV_ETHPHY_Deinitialize	Deinitializes the specified instance of the Ethernet PHY driver module. **Implementation:** Dynamic
	DRV_ETHPHY_HWConfigFlagsGet	Returns the current Ethernet PHY hardware MII/RMII and ALTERNATE/DEFAULT configuration flags. **Implementation:** Dynamic
	DRV_ETHPHY_Initialize	Initializes the Ethernet PHY driver. **Implementation:** Dynamic
	DRV_ETHPHY_LinkStatusGet	Returns the current link status. **Implementation:** Dynamic
	DRV_ETHPHY_NegotiationIsComplete	Returns the results of a previously initiated Ethernet PHY negotiation. **Implementation:** Dynamic
	DRV_ETHPHY_NegotiationResultGet	Returns the result of a completed negotiation. **Implementation:** Dynamic
	DRV_ETHPHY_Open	Opens the specified Ethernet PHY driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_ETHPHY_PhyAddressGet	Returns the PHY address. **Implementation:** Dynamic
	DRV_ETHPHY_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings. **Implementation:** Dynamic
	DRV_ETHPHY_Reset	Immediately resets the Ethernet PHY. **Implementation:** Dynamic
	DRV_ETHPHY_RestartNegotiation	Restarts auto-negotiation of the Ethernet PHY link. **Implementation:** Dynamic
	DRV_ETHPHY_Setup	Initializes Ethernet PHY configuration and set up procedure. **Implementation:** Dynamic
	DRV_ETHPHY_SMIClockSet	Sets the SMI/MIIM interface clock. **Implementation:** Dynamic
	DRV_ETHPHY_SMIRead	Initiates a SMI/MIIM read transaction. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanDataGet	Gets the latest SMI/MIIM scan data result. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanStart	Starts the scan of a requested SMI/MIIM register. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanStatusGet	Gets the status of the SMI/MIIM scan data. **Implementation:** Dynamic
	DRV_ETHPHY_SMIScanStop	Stops the scan of a previously requested SMI/MIIM register. **Implementation:** Dynamic
	DRV_ETHPHY_SMIStatus	Returns the current status of the SMI/MIIM interface. **Implementation:** Dynamic
	DRV_ETHPHY_SMIWrite	Initiates a SMI/MIIM write transaction. **Implementation:** Dynamic
	DRV_ETHPHY_Status	Provides the current status of the Ethernet PHY driver module. **Implementation:** Dynamic
	DRV_ETHPHY_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Dynamic
	DRV_ETHPHY_VendorDataGet	Returns the current value of the vendor data. **Implementation:** Dynamic
	DRV_ETHPHY_VendorDataSet	Returns the current value of the vendor data. **Implementation:** Dynamic
	DRV_ETHPHY_VendorSMIReadResultGet	Reads the result of a previous vendor initiated SMI read transfer with DRV_ETHPHY_VendorSMIReadStart. **Implementation:** Dynamic
	DRV_ETHPHY_VendorSMIReadStart	Starts a vendor SMI read transfer. Data will be available with DRV_ETHPHY_VendorSMIReadResultGet. **Implementation:** Dynamic
	DRV_ETHPHY_VendorSMIWriteStart	Starts a vendor SMI write transfer. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_ETHPHY_INDEX_0	Ethernet PHY driver index definitions.
	DRV_ETHPHY_INDEX_1	This is macro DRV_ETHPHY_INDEX_1.
	DRV_ETHPHY_INDEX_COUNT	Number of valid Ethernet PHY driver indices.

Structures

	Name	Description
	DRV_ETHPHY_INIT	Contains all the data necessary to initialize the Ethernet PHY device.
	DRV_ETHPHY_OBJECT_BASE_TYPE	Identifies the base interface of a Ethernet PHY driver.
	DRV_ETHPHY_NEGOTIATION_RESULT	Contains all the data necessary to get the Ethernet PHY negotiation result
	DRV_ETHPHY_OBJECT	Identifies the interface of a Ethernet PHY vendor driver.
	DRV_ETHPHY_OBJECT_BASE	Identifies the base interface of a Ethernet PHY driver.
	DRV_ETHPHY_SETUP	Contains all the data necessary to set up the Ethernet PHY device.

Types

	Name	Description
	DRV_ETHPHY_RESET_FUNCTION	Pointer to a function to perform an additional PHY reset
	DRV_ETHPHY_VENDOR_MDIX_CONFIGURE	Pointer to function that configures the MDIX mode for the Ethernet PHY.
	DRV_ETHPHY_VENDOR_MII_CONFIGURE	Pointer to function to configure the Ethernet PHY in one of the MII/RMII operation modes.
	DRV_ETHPHY_VENDOR_SMI_CLOCK_GET	Pointer to a function to return the SMI/MIIM maximum clock speed in Hz of the Ethernet PHY.
	DRV_ETHPHY_VENDOR_WOL_CONFIGURE	Pointer to a function to configure the PHY WOL functionality

Description

Ethernet ETHPHY Device Driver Interface

The Ethernet ETHPHY device driver provides a simple interface to manage an Ethernet ETHPHY peripheral using MIIM (or SMI) interface. This file defines the interface definitions and prototypes for the Ethernet ETHPHY driver.

File Name

drv_ethphy.h

Company

Microchip Technology Inc.

drv_ethphy_config.h

Ethernet PHY driver configuration definitions template.

Macros

	Name	Description
	DRV_ETHPHY_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_ETHPHY_INDEX	Ethernet PHY static index selection.
	DRV_ETHPHY_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_ETHPHY_NEG_DONE_TMO	Value of the PHY negotiation complete time out as per IEEE 802.3 spec.
	DRV_ETHPHY_NEG_INIT_TMO	Value of the PHY negotiation initiation time out as per IEEE 802.3 spec.
	DRV_ETHPHY_PERIPHERAL_ID	Defines an override of the peripheral ID.
	DRV_ETHPHY_RESET_CLR_TMO	Value of the PHY Reset self clear time out as per IEEE 802.3 spec.
	DRV_ETHPHY_USE_DRV_MIIM	Defines the way the PHY driver accesses the MIIM bus to communicate with the PHY.

Description

Ethernet PHY Driver Configuration Definitions for the Template Version

These definitions statically define the driver's mode of operation.

File Name

drv_ethphy_config.h

Company

Microchip Technology Inc.

Flash Driver Library

This section describes the Flash Driver Library.

Introduction

The Flash Driver Library provides functions that allow low-level interface with the on-chip Flash.

Description

Through MHC, this driver provides low-level functions for writing and erasing sections of the Flash memory.

Flash Program Memory

The Flash Program Memory is readable, writeable, and erasable during normal operation over the entire operating voltage range.

A read from program memory is executed at one byte/word at a time depending on the width of the data bus.

A write to the program memory is executed in either blocks of specific sizes or a single word depending on the type of processor used.

An erase is performed in blocks. A bulk erase may be performed from user code depending on the type of processor supporting the operation.

Writing or erasing program memory will cease instruction fetches until the operation is complete, restricting memory access, and therefore preventing code execution. This is controlled by an internal programming timer.

Library Interface

Functions

	Name	Description
	DRV_FLASH_ErasePage	Erases a page of Flash. **Implementation:** Static
	DRV_FLASH_GetPageSize	Returns the size in bytes of a single "Page" which can be erased in the flash. **Implementation:** Static
	DRV_FLASH_GetRowSize	Returns the size in bytes of a single "Row" which can be written to the flash. **Implementation:** Static
	DRV_FLASH_Initialize	Initializes the Flash instance for the specified driver index. **Implementation:** Static
	DRV_FLASH_IsBusy	Returns true if the Flash device is still busy writing or is erasing. **Implementation:** Static
	DRV_FLASH_Open	Initializes a channel to the appropriate flash device.
	DRV_FLASH_WriteQuadWord	Writes four 4-byte words to the Flash at the (word-aligned) flashAddr. **Implementation:** Static
	DRV_FLASH_WriteRow	Writes an DRV_FLASH_ROW_SIZE bytes to the Flash at the (word-aligned) flashAddr. **Implementation:** Static
	DRV_FLASH_WriteWord	Writes a 4-byte Word to the Flash at the (word-aligned) flashAddr. **Implementation:** Static

Data Types and Constants

	Name	Description
	DRV_FLASH_INDEX_0	FLASH driver index definitions
	DRV_FLASH_PAGE_SIZE	Specifies the FLASH Driver Program Page Size in bytes.
	DRV_FLASH_ROW_SIZE	Specifies the FLASH Driver Program Row Size in bytes.

Description

This section describes the Application Programming Interface (API) functions of the Flash Driver Library.

Refer to each section for a detailed description.

Functions

DRV_FLASH_ErasePage Function

Erases a page of Flash.

Implementation: Static

File

drv_flash.h

C

```
void DRV_FLASH_ErasePage(const DRV_HANDLE handle, uint32_t flashAddr);
```

Returns

None.

Description

This function starts the process of erasing a page of Flash. It does not wait for the erase operation to be done. That is left to the user. It does not verify that the erase was successful. That is left to the user. It always erases a single page. The size of a page in bytes will vary by device. It will be available in the DRV_FLASH_PAGE_SIZE parameter.

Remarks

Most devices will be running for code stored in the Flash. This means that any erases of the Flash will necessarily be writes to program space. As such, they will prevent the CPU from reading further instructions until the write is done. However, some devices may have more than one Flash such that it can run from one while writing to another. Additionally, if the application is small enough, it may run out of a cache. In any case, it is up to the user to wait for an operation to complete and or to decide that such a wait is unnecessary.

Preconditions

The flashAddr is taken as a valid Flash address. No range checking occurs. Any previous Flash operations (write or erase) must be completed or this will fail silently. The Flash must be correctly erased at flashAddr.

Example

```
flashAddr = 0x9d008000;
DRV_FLASH_Erase_Page(handle, flashAddr);
```

Function

void DRV_FLASH_Erase_Page(uint32_t flashAddr);

DRV_FLASH_GetPageSize Function

Returns the size in bytes of a single "Page" which can be erased in the flash.

Implementation: Static

File

drv_flash.h

C

```
uint32_t DRV_FLASH_GetPageSize(const DRV_HANDLE handle);
```

Returns

None.

Description

This function allows the user to get the size of a flash Page.

Remarks

None.

Preconditions

None

Function

uint32_t DRV_FLASH_GetPageSize(const DRV_HANDLE handle)

DRV_FLASH_GetRowSize Function

Returns the size in bytes of a single "Row" which can be written to the flash.

Implementation: Static

File

drv_flash.h

C

```
uint32_t DRV_FLASH_GetRowSize(const DRV_HANDLE handle);
```

Returns

None.

Description

This function allows the user to get the size of a flash Row.

Remarks

None.

Preconditions

None

Function

uint32_t DRV_FLASH_GetRowSize(const DRV_HANDLE handle)

DRV_FLASH_Initialize Function

Initializes the Flash instance for the specified driver index.

Implementation: Static

File

drv_flash.h

C

```
SYS_MODULE_OBJ DRV_FLASH_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes the Flash Driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Remarks

This function must be called before any other Flash function is called. This function should only be called once during system initialization.

Preconditions

None.

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_FLASH_Initialize(
const SYS_MODULE_INDEX index,
const SYS_MODULE_INIT * const init
)

DRV_FLASH_IsBusy Function

Returns true if the Flash device is still busy writing or is erasing.

Implementation: Static

File

drv_flash.h

C

```
bool DRV_FLASH_IsBusy(const DRV_HANDLE handle);
```

Returns

- true - Indicates the Flash is busy
- false - Indicates the Flash is not busy

Description

This function checks whether the process of programming a Word into the Flash is still operating.

Remarks

Most devices will be running for code stored in the Flash. This means that any writes to the Flash will necessarily be writes to program space. As such, they will prevent the CPU from reading further instructions until the write is done. However, some devices may have more than one Flash such that it can run from one while writing to another. Additionally, if the application is small enough, it may run out of a cache. In any case, it is up to the user to wait for an operation to complete and or to decide that such a wait is unnecessary.

Preconditions

None.

Example

```
flashAddr = 0x9d008000;
sourceData = 0x12345678;
DRV_FLASH_Write_Word(flashAddr,  sourceData);
DRV_FLASH_IsBusy( void );
```

Function

bool DRV_FLASH_IsBusy(void)

DRV_FLASH_Open Function

Initializes a channel to the appropriate flash device.

File

drv_flash.h

C

```
DRV_HANDLE DRV_FLASH_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT ioIntent);
```

Returns

Handle for future calls to the driver's operations.

Preconditions

None

Function

DRV_HANDLE DRV_FLASH_Open(
const SYS_MODULE_INDEX index,
const DRV_IO_INTENT ioIntent
);

DRV_FLASH_WriteQuadWord Function

Writes four 4-byte words to the Flash at the (word-aligned) flashAddr.

Implementation: Static

File

drv_flash.h

C

```
void DRV_FLASH_WriteQuadWord(const DRV_HANDLE handle, uint32_t flashAddr, uint32_t * sourceData);
```

Returns

None.

Description

This function starts the process of programming a word into the Flash. It does not wait for the write operation to be done, which is left to the user. It does not verify that the write was successful, which is left to the user.

Remarks

Most devices will be running for code stored in the Flash. This means that any writes to the Flash will necessarily be writes to program space. As such, they will prevent the CPU from reading further instructions until the write is done. However, some devices may have more than one Flash such that it can run from one while writing to another. Additionally, if the application is small enough, it may run out of a cache. In any case, it is up to the user to wait for an operation to complete and or to decide that such a wait is unnecessary.

Preconditions

The flashAddr is taken as a valid Flash address. No range checking occurs. Any previous Flash operations (write or erase) must be completed or this will fail silently. The Flash must be correctly erased at flashAddr.

Example

```
flashAddr = 0x9d008000;
sourceData[4] = {0x12345678,0x9ABCDEF0,0x55AAAA55,0x11111111};
DRV_FLASH_WriteQuadWord(handle, flashAddr,  sourceData);
```

Function

void DRV_FLASH_WriteQuadWord(const DRV_HANDLE handle, uint32_t flashAddr, uint32_t sourceData)

DRV_FLASH_WriteRow Function

Writes an DRV_FLASH_ROW_SIZE bytes to the Flash at the (word-aligned) flashAddr.

Implementation: Static

File

drv_flash.h

C

```
void DRV_FLASH_WriteRow(const DRV_HANDLE handle, uint32_t flashAddr, uint32_t sourceData);
```

Returns

None.

Description

This function starts the process of programming a buffer into the Flash. It does not wait for the write operation to be done, which is left to the user. It does not verify that the write was successful, which is left to the user.

Remarks

Most devices will be running for code stored in the Flash. This means that any writes to the Flash will necessarily be writes to program space. As such, they will prevent the CPU from reading further instructions until the write is done. However, some devices may have more than one Flash such that it can run from one while writing to another. Additionally, if the application is small enough, it may run out of a cache. In any case, it is up to the user to wait for an operation to complete and or to decide that such a wait is unnecessary.

Preconditions

The flashAddr is taken as a valid Flash address. No range checking occurs. The memory pointed to by sourceData must be valid memory for at least DRV_FLASH_ROW_SIZE bytes. Any previous Flash operations (write or erase) must be completed or this will fail silently. The Flash must be correctly erased at flashAddr.

Example

```
flashAddr = 0x9d008000;
```

```
uint32_t dataStore[DRV_FLASH_ROW_SIZE] = {0,1,2,3,4,5};
DRV_FLASH_Write_Row( const DRV_HANDLE handle, flashAddr,  dataStore);
```

Function

void DRV_FLASH_WriteRow(const DRV_HANDLE handle, uint32_t flashAddr, uint32_t sourceData)

DRV_FLASH_WriteWord Function

Writes a 4-byte Word to the Flash at the (word-aligned) flashAddr.

Implementation: Static

File

drv_flash.h

C

```
void DRV_FLASH_WriteWord(const DRV_HANDLE handle, uint32_t flashAddr, uint32_t sourceData);
```

Returns

None.

Description

This function starts the process of programming a Word into the Flash. It does not wait for the write operation to be done, which is left to the user. It does not verify that the write was successful, which is left to the user.

Remarks

Most devices will be running for code stored in the Flash. This means that any writes to the Flash will necessarily be writes to program space. As such, they will prevent the CPU from reading further instructions until the write is done. However, some devices may have more than one Flash such that it can run from one while writing to another. Additionally, if the application is small enough, it may run out of a cache. In any case, it is up to the user to wait for an operation to complete and or to decide that such a wait is unnecessary.

Preconditions

The flashAddr is taken as a valid Flash address. No range checking occurs. Any previous Flash operations (write or erase) must be completed or this will fail silently. The Flash must be correctly erased at flashAddr.

Example

```
flashAddr = 0x9d008000;
sourceData = 0x12345678;
DRV_FLASH_WriteWord(handle, flashAddr,  sourceData);
```

Function

void DRV_FLASH_WriteWord(const DRV_HANDLE handle, uint32_t flashAddr, uint32_t sourceData)

Data Types and Constants

DRV_FLASH_INDEX_0 Macro

FLASH driver index definitions

File

drv_flash.h

C

```
#define DRV_FLASH_INDEX_0 0
```

Description

These constants provide FLASH driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_FLASH_Initialize and DRV_FLASH_Open routines to identify the driver instance in use.

Section

Constants

****

****

****

Driver FLASH Module Index

DRV_FLASH_PAGE_SIZE Macro

Specifies the FLASH Driver Program Page Size in bytes.

File

drv_flash.h

C

```
#define DRV_FLASH_PAGE_SIZE (NVM_PAGE_SIZE)
```

Description

FLASH Driver Program Page Size.

This definition specifies the FLASH Driver Program Page Size in bytes. This parameter is device specific and is obtained from the device specific processor header file.

Remarks

None

DRV_FLASH_ROW_SIZE Macro

Specifies the FLASH Driver Program Row Size in bytes.

File

drv_flash.h

C

```
#define DRV_FLASH_ROW_SIZE (NVM_ROW_SIZE)
```

Description

FLASH Driver Program Row Size.

This definition specifies the FLASH Driver Program Row Size in bytes. This parameter is device specific and is obtained from the device specific processor header file. The Program Row Size is the maximum block size that can be programmed in one program operation.

Remarks

None

Files

Files

Name	Description
drv_flash.h	Flash Driver interface declarations for the static single instance driver.

Description

drv_flash.h

Flash Driver interface declarations for the static single instance driver.

Functions

	Name	Description
	DRV_FLASH_ErasePage	Erases a page of Flash. **Implementation:** Static
	DRV_FLASH_GetPageSize	Returns the size in bytes of a single "Page" which can be erased in the flash. **Implementation:** Static
	DRV_FLASH_GetRowSize	Returns the size in bytes of a single "Row" which can be written to the flash. **Implementation:** Static
	DRV_FLASH_Initialize	Initializes the Flash instance for the specified driver index. **Implementation:** Static
	DRV_FLASH_IsBusy	Returns true if the Flash device is still busy writing or is erasing. **Implementation:** Static
	DRV_FLASH_Open	Initializes a channel to the appropriate flash device.
	DRV_FLASH_WriteQuadWord	Writes four 4-byte words to the Flash at the (word-aligned) flashAddr. **Implementation:** Static
	DRV_FLASH_WriteRow	Writes an DRV_FLASH_ROW_SIZE bytes to the Flash at the (word-aligned) flashAddr. **Implementation:** Static
	DRV_FLASH_WriteWord	Writes a 4-byte Word to the Flash at the (word-aligned) flashAddr. **Implementation:** Static

Macros

	Name	Description
	DRV_FLASH_INDEX_0	FLASH driver index definitions
	DRV_FLASH_PAGE_SIZE	Specifies the FLASH Driver Program Page Size in bytes.
	DRV_FLASH_ROW_SIZE	Specifies the FLASH Driver Program Row Size in bytes.

Description

Flash Driver Interface Declarations for Static Single Instance Driver

The Flash device driver provides a simple interface to manage the Flash Controller on Microchip microcontrollers. This file defines the interface Declarations for the Flash driver.

Remarks

Static interfaces incorporate the driver instance number within the names of the routines, eliminating the need for an object ID or object handle.

Static single-open interfaces also eliminate the need for the open handle.

File Name

drv_flash.h

Company

Microchip Technology Inc.

Ethernet GMAC Driver Library

This section describes the Ethernet MAC Driver Library.

Introduction

This library provides a driver-level abstraction of the on-chip Ethernet Controller found on many PIC32 devices. The driver implements the virtual MAC driver model that the MPLAB Harmony TCP/IP Stack requires. Please see the TCP/IP Stack Library MAC Driver Module help for details.

The "Host-To-Network" layer of a TCP/IP stack organization covers the Data Link and Physical Layers of the standard OSI stack. The Ethernet Controller provides the Data Link or Media Access Control Layer, in addition to other functions discussed in this section. An external Ethernet "PHY" provides the Physical layer, providing conversion between the digital and analog.

Description

The Ethernet Media Access Controller (GMAC) module implements a 10/100 Mbps Ethernet MAC, compatible with the IEEE 802.3 standard. The GMAC can operate in either half or full duplex mode at all supported speeds.

Embedded Characteristics

- Compatible with IEEE Standard 802.3
- 10, 100 Mbps operation
- Full and half duplex operation at all supported speeds of operation
- Statistics Counter Registers for RMON/MIB
- MII interface to the physical layer
- Integrated physical coding
- Direct memory access (DMA) interface to external memory
- Support for 6 priority queues in DMA
- 8 KB transmit RAM and 4 KB receive RAM
- Programmable burst length and endianism for DMA
- Interrupt generation to signal receive and transmit completion, errors or other events
- Automatic pad and cyclic redundancy check (CRC) generation on transmitted frames
- Automatic discard of frames received with errors
- Receive and transmit IP, TCP and UDP checksum offload. Both IPv4 and IPv6 packet types supported
- Address checking logic for four specific 48-bit addresses, four type IDs, promiscuous mode, hash matching of unicast and multicast destination addresses and Wake-on-LAN
- Management Data Input/Output (MDIO) interface for physical layer management
- Support for jumbo frames up to 10240 Bytes
- Full duplex flow control with recognition of incoming pause frames and hardware generation of transmitted pause frames
- Half duplex flow control by forcing collisions on incoming frames
- Support for 802.1Q VLAN tagging with recognition of incoming VLAN and priority tagged frames
- Support for 802.1Qbb priority-based flow control
- Programmable Inter Packet Gap (IPG) Stretch
- Recognition of IEEE 1588 PTP frames
- IEEE 1588 time stamp unit (TSU)
- Support for 802.1AS timing and synchronization
- Supports 802.1Qav traffic shaping on two highest priority queues

Using the Library

The user of this driver is the MPLAB Harmony TCP/IP stack. This Ethernet driver is not intended as a system wide driver that the application or other system modules may use. It is intended for the sole use of the MPLAB Harmony TCP/IP stack and implements the virtual MAC model required by the stack.

This topic describes the basic architecture and functionality of the Ethernet MAC driver and is meant for advanced users or TCP/IP stack driver developers.

Interface Header File: `drv_gmac.h`

The interface to the Ethernet MAC library is defined in the `drv_gmac.h` header file, which is included by the MPLAB Harmony TCP/IP stack.

Please refer to the What is MPLAB Harmony? section for how the library interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the Ethernet GMAC Driver Library on Microchip's microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

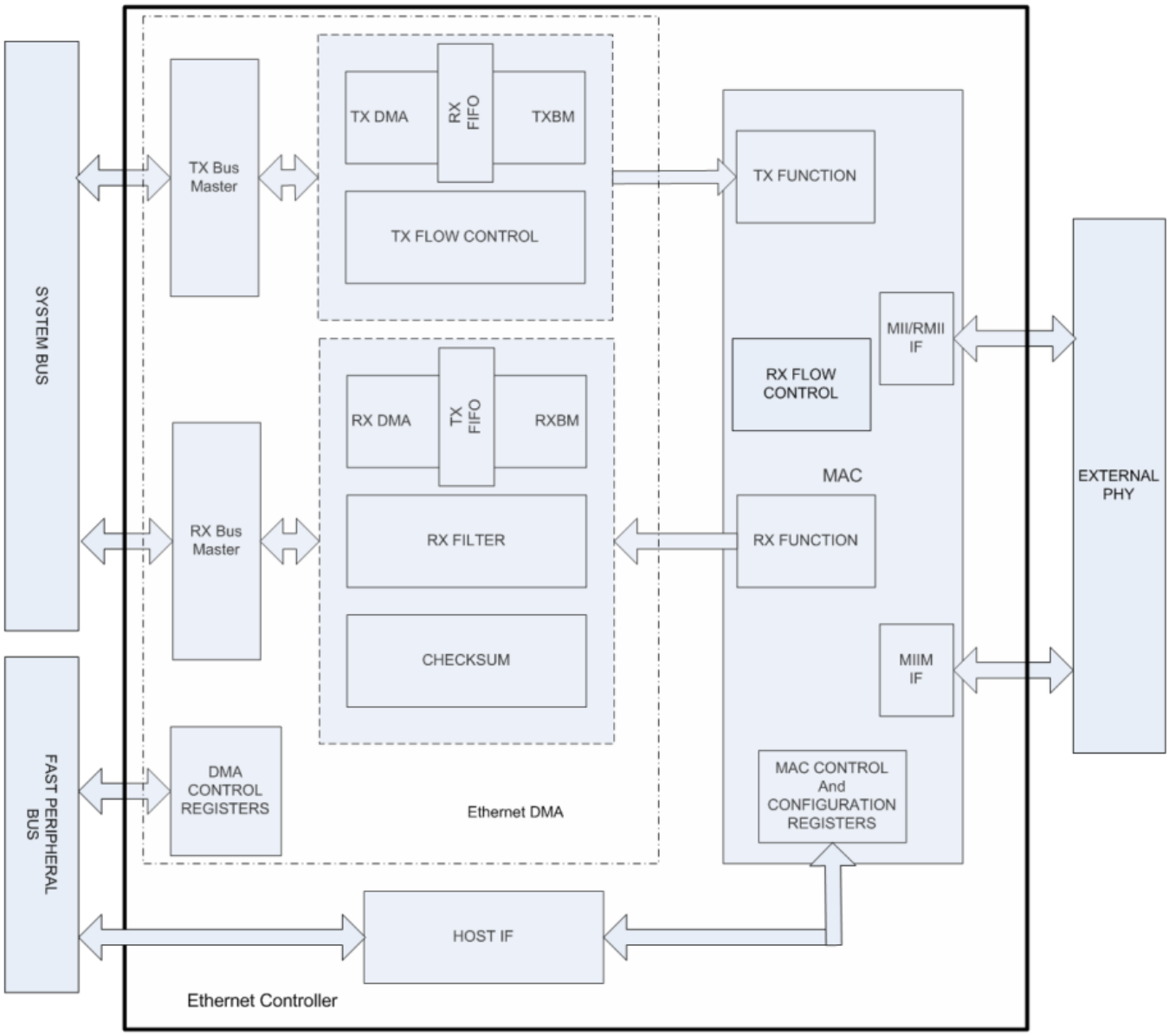
Description

The Ethernet Controller provides the modules needed to implement a 10/100 Mbps Ethernet node using an external Ethernet PHY chip. The PHY chip provides a digital-analog interface as part of the Physical Layer and the controller provides the Media Access Controller (MAC) layer above the PHY.

As shown in Figure 1, the Ethernet Controller consists of the following modules:

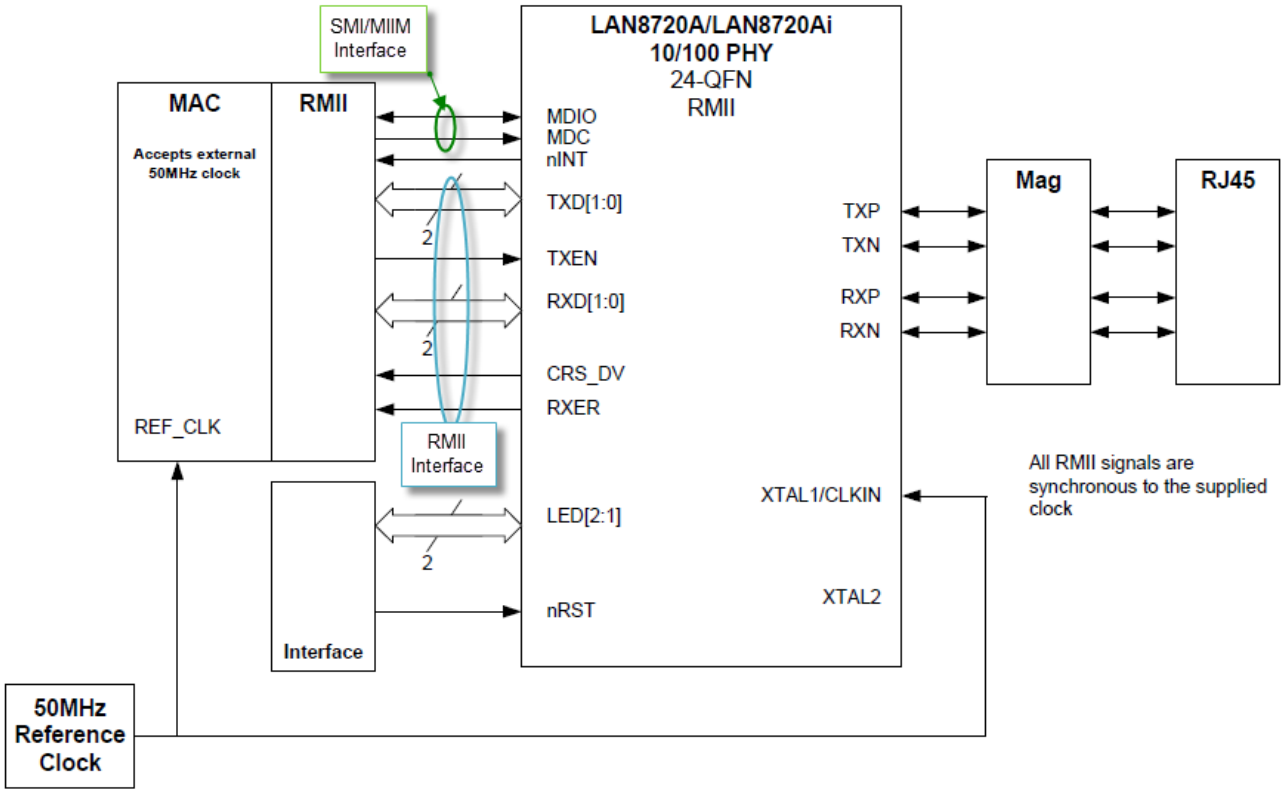
- Media Access Control (MAC) block: Responsible for implementing the MAC functions of the Ethernet IEEE 802.3 Specification
- Flow Control (FC) block: Responsible for control of the transmission of PAUSE frames. (Reception of PAUSE frames is handled within the MAC.)
- RX Filter (RXF) block: This module performs filtering on every receive packet to determine whether each packet should be accepted or rejected
- TX DMA/TX Buffer Management Engine: The TX DMA and TX Buffer Management engines perform data transfers from the memory (using descriptor tables) to the MAC Transmit Interface
- RX DMA/RX Buffer Management Engine: The RX DMA and RX Buffer Management engines transfer receive packets from the MAC to the memory (using descriptor tables)

Figure 1: Ethernet Controller Block Diagram



For completeness, we also need to look at the interface diagram of a representative Ethernet PHY. As shown in Figure 2, the PHY has two interfaces, one for configuring and managing the PHY (SMI/MIIM) and another for transmit and receive data (RMII or MII). The SMI/MIIM interface is the responsibility of the Ethernet PHY Driver Library. When setting up the Ethernet PHY, this Ethernet driver calls primitives from the Ethernet PHY Driver library. The RMII/MII data interface is the responsibility of the Ethernet MAC Driver Library (this library).

Figure 2: Ethernet PHY Interfaces



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system. Refer to the TCP/IP Stack Library MAC Driver Module help for the interface that the Ethernet driver has to implement in a MPLAB Harmony system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Ethernet GMAC Driver Library.

Library Interface Section	Description
Client Level Functions	Open, Close, Initialize, Reinitialize, and Deinitialize functions to support the TCP/IP Stack. Plus link status and power options.
Receive Functions	Receive routines.
Transmit Functions	Transmit routines.
Event Functions	Ethernet event support routines.
Other Functions	Additional routines.
Data Types and Constants	Typedefs and #defines.

Configuring the Library

The configuration of the Ethernet MAC driver is done as part of the MPLAB Harmony TCP/IP Stack configuration and is based on the `system_config.h` file, which may include the `tcpip_mac_config.h`. See the TCP/IP Stack Library MAC Driver Module help file for configuration options.

This header file contains the configuration selection for the Ethernet GMAC Driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Building the Library

This section lists the files that are available in the Ethernet GMAC Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/gmac`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_gmac.h`	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_gmac.c`	PIC32 internal Ethernet driver virtual GMAC implementation file.
`/src/dynamic/drv_gmac_lib.c`	PIC32 internal Ethernet driver controller implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The Ethernet MAC Driver Library depends on the following modules:

- Ethernet PHY Driver Library
- Interrupt System Service Library
- Timer System Service Library
- Ethernet Peripheral Library

Library Interface

This section lists the interface routines, data types, constants and macros for the library.

a) Client Level Functions

b) Receive Functions

c) Transmit Functions

d) Event Functions

e) Other Functions

f) Data Types and Constants

Files

Files

Name	Description
drv_gmac.h	This is file drv_gmac.h.

Description

This section lists the source and header files used by the Ethernet MAC Driver Library.

drv_gmac.h

This is file drv_gmac.h.

I2C Driver Library Help

This section describes the I2C Driver Library.

Introduction

This library provides an interface to manage the data transfer operations using the I2C module on the Microchip family of microcontrollers.

Description

The driver communicates using the concept of *transactions*. In instances where the I2C operates in Master mode, the driver sends the start signal, followed by a slave device address (including a Read/Write bit), followed by a number of bytes written to or read from the slave. The *transaction* is completed by sending the stop signal. When the driver operates in the Slave mode, it will either read data or write data to the master.

This driver library provides application ready routines to read and write data using the I2C protocol, thus minimizing developer's awareness of the working of the I2C protocol.

- Provides read/write and buffer data transfer models
- Supports interrupt and Polled modes of operation
- Support multi-client and multi-instance operation
- Provides data transfer events
- Supports blocking and non-blocking operation
- Supports baud rate setting
- Supports bit bang mode.

Using the Library

This topic describes the basic architecture of the I2C Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_i2c.h`

The interface to the I2C Driver Library is defined in the `drv_i2c.h` header file. Any C language source (`.c`) file that uses the I2C Driver Library should include `drv_i2c.h`.

Library File: The I2C Driver Library archive (`.a`) file is installed with MPLAB Harmony.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

The I2C Driver Library provides the low-level abstraction of the I2C module on the Microchip family of microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the I2C Driver Library interface.

Description

The I2C Driver Library features routines to perform two functions, driver maintenance and data transfer:

Driver Maintenance

The Driver initialization routines allow the application to initialize the driver. The initialization data configures the I2C module as a Master or a Slave and sets the necessary parameters required for operation in the particular mode. The driver must be initialized before it can be used by the application. After the end of operation, the driver can be deinitialized.

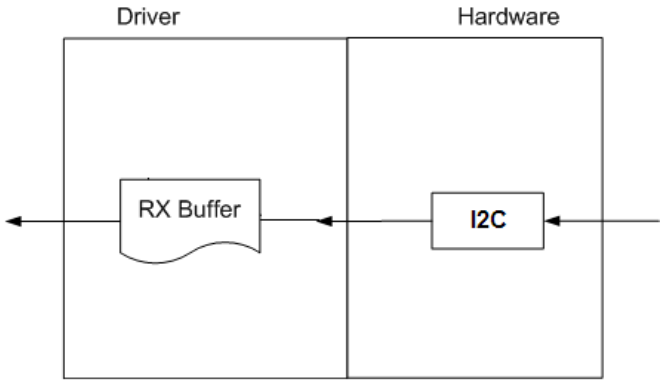
Data Transfer

Data transfer is accomplished by separate Write and Read functions through a data buffer. The read and write function makes the user transparent to the internal working of the I2C protocol. The user can use callback mechanisms or use polling to check status of transfer.

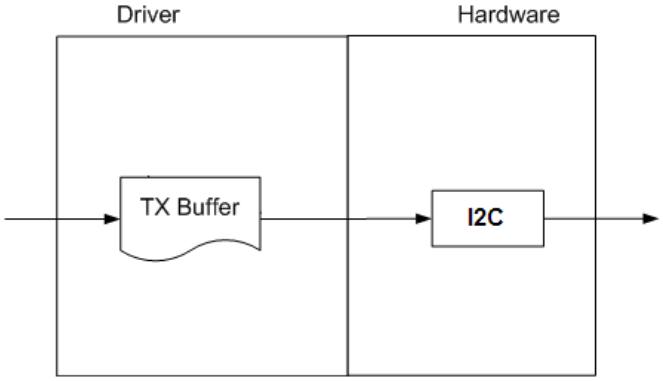
The following diagrams illustrate the model used by the I2C Driver for transmitter and receiver.

Note: The driver can be configured to use either the SOC peripheral, or in Bit Bang mode. Bit Bang mode uses CPU resources to process the data bits onto or off of the I2C I/O pins. Be aware that and I2C driver configured this way uses a large amount of CPU resources.

Receiver Abstraction Model



Transmitter Abstraction Model



Library Overview

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the I2C Driver Library.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open, close, status and other setup functions.
Data Transfer Functions	Provides data transfer functions available in the configuration.
Miscellaneous Functions	Provides miscellaneous driver functions.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

System Access

This section provides information on system access.

Description

System Access

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the I2C module would be initialized with the following configuration settings (either passed dynamically at run-time using DRV_I2C_INIT or by using initialization overrides) that are supported by the specific I2C device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- The actual peripheral ID enumerated as the PLIB level module ID (e.g., I2C_ID_2)
- Master or Slave mode of operation and their associated parameters
- Defining the respective interrupt sources for Master, Slave, and Error Interrupt

The DRV_I2C_Initialize API returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the Initialize interface would be used by the other system interfaces like DRV_I2C_Deinitialize, DRV_I2C_Status, and DRV_I2C_Tasks.

Note: The system initialization settings, only affect the instance of the peripheral that is being initialized.

Example:

```
DRV_I2C_INIT        i2c_init_data;
SYS_MODULE_OBJ      objectHandle;

i2c_init_data.i2cId = DRV_I2C_PERIPHERAL_ID_IDX0,
i2c_init_data.i2cMode = DRV_I2C_MODE_MASTER,
OR
i2c_init_data.i2cMode = DRV_I2C_MODE_SLAVE,

/*    Master mode parameters */
i2c_init_data.baudRate = 100000,
i2c_init_data.busspeed = DRV_I2C_SLEW_RATE_CONTROL_IDX0,
i2c_init_data.buslevel = DRV_I2C_SMBus_SPECIFICATION_IDX0,

/*    Master mode parameters */
i2c_init_data.addWidth = DRV_I2C_7BIT_SLAVE,
i2c_init_data.reservedaddenable  = false,
i2c_init_data.generalcalladdress = false,
i2c_init_data.slaveaddvalue = 0x0060,

//interrupt sources
i2c_init_data.mstrInterruptSource = INT_SOURCE_I2C_2_MASTER,
i2c_init_data.slaveInterruptSource = INT_SOURCE_I2C_2_ERROR,
i2c_init_data.errInterruptSource = INT_SOURCE_I2C_2_ERROR,
i2c_init_data.queueSize = 1,

/* callback for Master (Master mode can use callbacks if needed) */
i2c_init_data.operationStarting = NULL,

/* Slave mode callbacks needed */
i2c_init_data.operationStarting = APP_I2CSlaveFunction,

objectHandle = DRV_I2C_Initialize(DRV_I2C_INDEX_0, (SYS_MODULE_INIT *)&drvI2C0InitData)
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
   // Handle error
}
```

Since the I2C bus is controlled by the Master, the Slave should respond to a read or write request whenever the Master makes the request. Thus, the slave does not have driver states like the Master. The operation of the I2C Driver when used in Slave mode is handled using callbacks. The callback, OperationStarting, must be configured during system initialization when in Slave mode. This callback is provided so that the application can respond appropriately when a read or write request is received from the Master.

Client Access

This section provides information on client access.

Description

For the application to start using an instance of the module, it must call the DRV_I2C_Open function. This provides the configuration required to open the I2C instance for operation. If the driver is deinitialized using the function DRV_I2C_Deinitialize, the application must call the DRV_I2C_Open function again to set up the instance of the I2C.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

After a client instance is opened, DRV_I2C_ClientSetup can be called to set up client-specific parameters. In I2C Slave mode, this is used to set-up the IRQ logic so that the slave can toggle this line to request Master to send a Read command.

As during initialization, when the I2C module operates in the Slave mode, only the Master can terminate a transaction with the Slave. In this case, the driver provides a callback to the application after the reception of each byte from the Master or after transmission of a byte to the Master.

Example:

```
/* I2C Driver Handle  */
DRV_HANDLE  drvI2CHandle;

/* Open the I2C Driver */
appData.drvI2CHandle = DRV_I2C_Open( DRV_I2C_INDEX_0,DRV_IO_INTENT_WRITE );

if (drvI2CHandle != DRV_HANDLE_VALID)
{
     //Client cannot open instance
}
```

Client Transfer

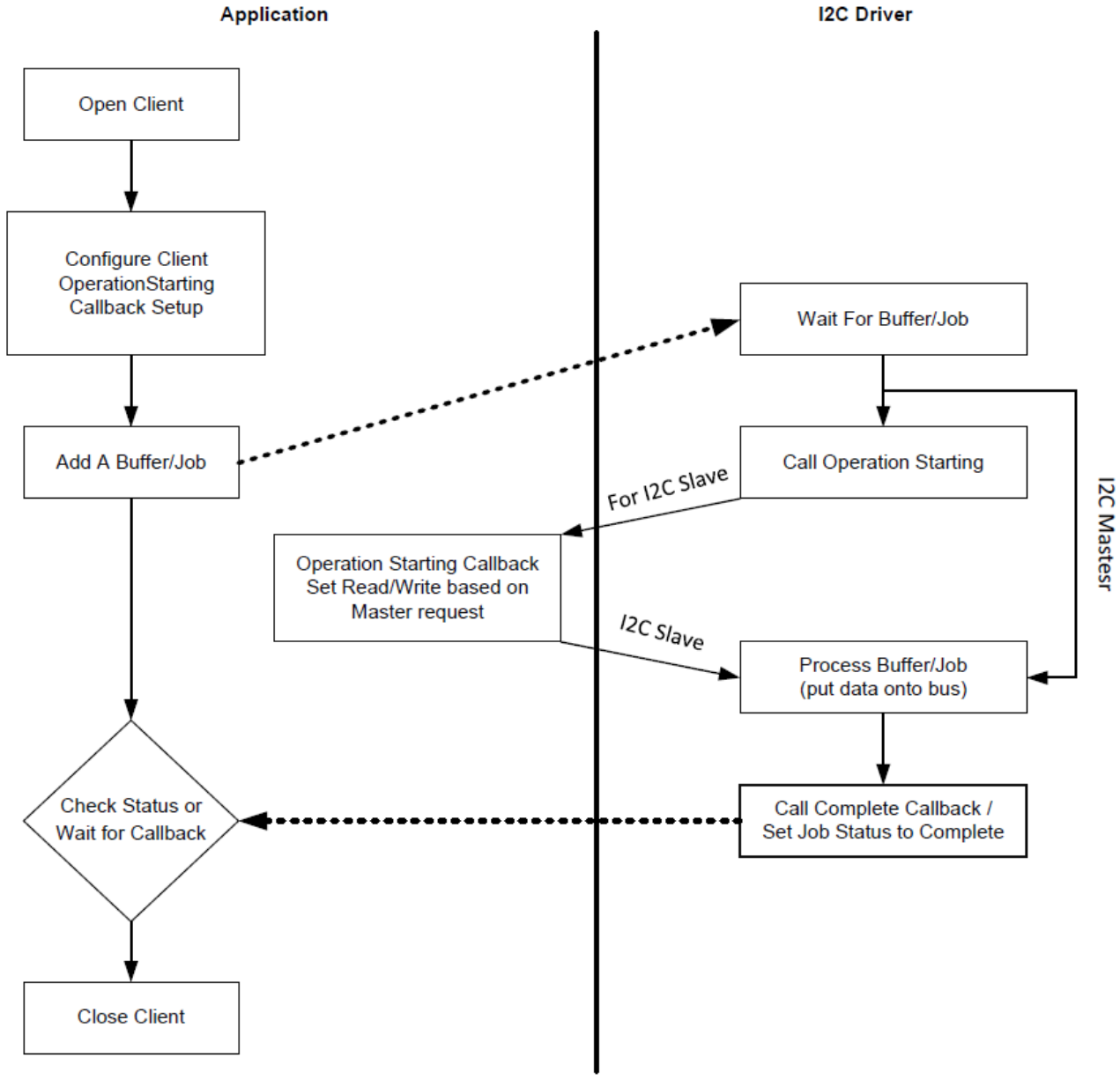
This section provides information on client transfer functionality.

Description

Core Functionality

Client basic functionality provides an extremely basic interface for the driver operation.

The following diagram illustrates the byte/word model used for the data transfer.



Client Data Transfer Functionality

Applications using the I2C driver need to perform the following:

1. The system should have completed necessary initialization and the DRV_I2C_Tasks should either be running in polled environment, or in an interrupt environment.
2. Open the driver using DRV_I2C_Open with the necessary intent.
3. Add a buffer using the DRV_I2C_Receive, DRV_I2C_Transmit, and DRV_I2C_TransmitThenReceive functions. An optional callback can be provided that will be called when the buffer/job is complete using DRV_I2C_BufferEventHandlerSet.
4. Check for the current transfer status using DRV_I2C_TransferStatusGet or wait for the callback to be called with buffer transfer status set to DRV_I2C_BUFFER_EVENT_COMPLETE or DRV_I2C_BUFFER_EVENT_ERROR.
5. Buffer status DRV_I2C_BUFFER_EVENT_COMPLETE implies that the I2C transaction has been completed without any errors. Buffer status DRV_I2C_BUFFER_EVENT_ERROR indicates that the I2C transaction was aborted and the entire contents of the buffer were not transferred.
6. In Master mode, common cases for DRV_I2C_BUFFER_EVENT_ERROR to be set are:
   - Slave is non-operational
   - Slave is performing an internal operation and cannot accept any more I2C messages from the Master until the operation completes. In such a case, if the Master tries to address the Slave and is attempting to transfer data, the Slave NACKs the transfer. This will result in the Master prematurely terminating the transaction and setting the DRV_I2C_BUFFER_EVENT_FLAG. In the application level, the Master can continuously attempt to send the transaction until transfer status changes from for DRV_I2C_BUFFER_EVENT_ERROR to DRV_I2C_BUFFER_EVENT_COMPLETE. This will in effect perform the so-called "Acknowledge Polling". An example of a Slave device that depicts this behavior is an EEPROM.

7. The client will be able to close the driver using DRV_I2C_Close when required.

Example:

```
/*  This example demonstrates the I2C driver setup of one instance of I2C acting
    as a Master to another instance of the I2C Driver acting as a Slave.
    In the Slave initialization data structure in system_init.c, the member
    operationStarting should be assigned a function pointer. This function will
    be called when the Slave receives an address match. Based on the R/W bit in
    the address, the transmit or receive function will be called by the Slave
    (e.g., .operationStarting = APP_SlaveDataforMaster) */

SYS_MODULE_OBJ i2cMasterObject;

SYS_MODULE_OBJ i2cSlaveObject;


/* function prototype of callback function */
void I2CMasterOpStatusCb ( DRV_I2C_BUFFER_EVENT event,
                           DRV_I2C_BUFFER_HANDLE bufferHandle,
                           uintptr_t context);

int main( void )
{
    while ( 1 )
    {
        appTask ();
    }
}

void appTask ()
{
    #define MY_BUFFER_SIZE             5
    #define RTCC_SLAVE_ADDRESS        0xDE

    /* initialize slave address value */
    unsigned char address             =    RTCC_SLAVE_ADDRESS;

    /*Initialize myBuffer with MY_BUFFER_SIZE bytes of valid data */
    char myBuffer[MY_BUFFER_SIZE] = { 11, 22, 33, 44, 55};
    unsigned int numBytes;

    DRV_HANDLE drvI2CMasterHandle;        //Returned from DRV_I2C_Open for I2C Master
    DRV_I2C_BUFFER_HANDLE bufHandle_M1;   //Returned from calling a Data Transfer function
    uintptr_t i2cOpStatus;                //Operation status of I2C operation returned from callback

    DRV_HANDLE drvI2CSlaveHandle;         //Returned from DRV_I2C_Open for I2C Slave
    DRV_I2C_BUFFER_HANDLE bufHandle_S1;   //Returned from calling a Data Transfer function
    DRV_I2C_BUFFER_HANDLE bufHandle_S2;   //Returned from calling a Data Transfer function



    while( 1 )
    {
        switch( state )
        {
            case APP_STATE_INIT:
            {
                /* Initialize the Master I2C Driver */
                i2cMasterObject = DRV_I2C_Initialize( DRV_I2C_INDEX_0, (SYS_MODULE_INIT *)&drvI2C0InitData
);

                /* Initialize the Slave I2C Driver */
                i2cSlaveObject  = DRV_I2C_Initialize(DRV_I2C_INDEX_1, (SYS_MODULE_INIT *)&drvI2C1InitData);

                /* Check for the System Status */
                if( SYS_STATUS_READY != DRV_I2C_Status( i2cObject ) )
                return 0;

                /* Open the Driver for I2C Master */
                drvI2CMasterHandle = DRV_I2C_Open( DRV_I2C_INDEX_0,DRV_IO_INTENT_WRITE );
```

```
                if ( drvI2CMasterHandle != (DRV_HANDLE)NULL )
                {
                    /* event-handler set up receive callback from DRV_I2C_Tasks */
                    /* Event handler need to be set up only if needed */
                    DRV_I2C_BufferEventHandlerSet(drvI2CMasterHandle, I2CMasterOpStatusCb, i2cOpStatus );

                    /* Update the state to transfer data */
                    state = APP_STATE_DATA_PUT;
                }
                else
                {
                    state = APP_STATE_ERROR;
                }

                /* Open the I2C Driver for Slave on the same device */
                drvI2CSlaveHandle = DRV_I2C_Open( DRV_I2C_INDEX_1,DRV_IO_INTENT_WRITE );

                if ( drvI2CMasterHandle != (DRV_HANDLE)NULL )
                {
                    /* event-handler set up receive callback from DRV_I2C_Tasks */
                    /* Event handler need to be set up only if needed */
                    DRV_I2C_BufferEventHandlerSet(drvI2CMasterHandle, I2CMasterOpStatusCb, i2cOpStatus );

                    /* Update the state to transfer data */
                    state = APP_STATE_DATA_PUT;
                }
                else
                {
                    state = APP_STATE_ERROR;
                }


                break;
            }
            case APP_STATE_DATA_PUT:
            {
                /* I2C master writes data onto I2C bus */
                bufHandle_M1 = DRV_I2C_Transmit ( drvI2CMasterHandle , address, &myBuffer[], 5, NULL );

                /* Update the state to status check */
                state = APP_STATE_DATA_CHECK;
                break;
            }
            case APP_STATE_DATA_CHECK:
            {
                /* Check for the successful data transfer */
                if( DRV_I2C_BUFFER_EVENT_COMPLETE == DRV_I2C_TransferStatusGet
                                                    (drvI2CMasterHandle, bufHandle_M1) )
                {
                    /* Do this repeatedly */
                    state = APP_STATE_DATA_PUT;
                }
                break;
            }
            case APP_STATE_ERROR:
            {

                //include any error handling routines here

                break;
            }
            default:
            {
                break;
            }
        }
    }
```

```
/*****************************************************************************/
//  Function: I2CMasterOpStatusCb
//
//  Callback called in Master mode from the DRV_I2C_Tasks function. This
//  callback is invoked when the Master has to indicate to the application
//  that the BUFFER event is COMPLETE or there was an error in transmission.
//*****************************************************************************/

void I2CMasterOpStatusCb (    DRV_I2C_BUFFER_EVENT event,
                           DRV_I2C_BUFFER_HANDLE bufferHandle,
                           uintptr_t context)
{
    switch(event)
    {
        case DRV_I2C_BUFFER_EVENT_COMPLETE:
            //this indicates that the I2C transaction has completed
            //DRV_I2C_BUFFER_EVENT_COMPLETE can be handled in the callback
            //or by checking for this event using the API DRV_I2C_BufferStatus
            /* include any callback event handling code here if needed */
        break;
        case DRV_I2C_BUFFER_EVENT_ERROR:
            //this indicates that the I2C transaction has completed
            //and a STOP condition has been asserted on the bus.
            //However the slave has NACKED either the address or data
            //byte.
            /* include any callback event handling code here if needed */
        default:
        break;
    }

}

//*****************************************************************************/
//  Function: APP_SlaveDataforMaster
//
//  Callback function from DRV_I2C_Tasks when operating as a Slave. When an
//  address match is received by the Slave, this callback is executed and
//  the buffer event depends on the R/W bit. If R/W = 0, DRV_I2C_Receive is
//  called implying the Slave is going to read data send from the Master.
//  If R/W = 1, DRV_I2C_Transmit is called implying the Slave is going to send
//  data to the Master.
//*****************************************************************************/

void APP_SlaveDataforMaster(DRV_I2C_BUFFER_EVENT event, void * context)
{
    switch (event)
    {
        case DRV_I2C_BUFFER_SLAVE_READ_REQUESTED:
            deviceAddressPIC32 = PIC32_SLAVE_ADDRESS;

            bufHandle_S1 = DRV_I2C_Receive( drvI2CSlaveHandle,
                                            deviceAddressPIC32,
                                            &SlaveRxbuffer[0],
                                            NUMBER_OF_UNKNOWN_BYTES_TO_SLAVE,
                                            NULL );

            break;
        case DRV_I2C_BUFFER_SLAVE_WRITE_REQUESTED:
            deviceAddressPIC32 = PIC32_SLAVE_ADDRESS;

            bufHandle_S2 = DRV_I2C_Transmit ( drvI2CSlaveHandle,
                                              deviceAddressPIC32,
                                              &SlaveTxbuffer[0],
                                              NUMBER_OF_UNKNOWN_BYTES_TO_SLAVE,
                                              NULL );
            break;
        default:
            break;
    }
```

```
}

void __ISR(_I2C_2_VECTOR, ipl4AUTO) _IntHandlerDrvI2CInstance0(void)
{
   DRV_I2C_Tasks(i2cMasterObject);
}

void __ISR(_I2C1_SLAVE_VECTOR, ipl6AUTO) _IntHandlerDrvI2CSlaveInstance1(void)
{
    DRV_I2C_Tasks(i2cSlaveObject);
}
```

Configuring the Library

Macros

	Name	Description
	DRV_DYNAMIC_BUILD	Dynamic driver build, dynamic device instance parameters.
	DRV_I2C_CONFIG_BUILD_TYPE	Selects static or dynamic driver build configuration.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BASIC	Enables the device driver to support basic transfer mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BLOCKING	Enables the device driver to support blocking operations.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_EXCLUSIVE	Enables the device driver to support operation in Exclusive mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_MASTER	Enables the device driver to support operation in Master mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_NON_BLOCKING	Enables the device driver to support non-blocking during operations
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_READ	Enables the device driver to support read operations.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_SLAVE	Enables the device driver to support operation in Slave mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE	Enables the device driver to support write operations.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE_READ	Enables the device driver to support write followed by read.
	DRV_STATIC_BUILD	Static driver build, static device instance parameters.
	DRV_I2C_FORCED_WRITE	Includes function that writes to slave irrespective of whether receiving a ACK or NACK from slave
	I2C_STATIC_DRIVER_MODE	Selects the type of STATIC driver

Description

The configuration of the I2C Driver Library is based on the file `sys_config.h`.

This header file contains the configuration selection for the I2C Driver Library. Based on the selections made, the I2C Driver Library may support the selected features. These configuration settings will apply to all instances of the I2C Driver Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_DYNAMIC_BUILD Macro

Dynamic driver build, dynamic device instance parameters.

File

drv_i2c_config_template.h

C

```
#define DRV_DYNAMIC_BUILD 1
```

Description

Dynamic Driver Build Configuration

This value, if used to identify the build type for a driver, will cause the driver to be built to dynamically, identify the instance of the peripheral at run-time using the parameter passed into its API routines.

DRV_I2C_CONFIG_BUILD_TYPE Macro

Selects static or dynamic driver build configuration.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_BUILD_TYPE DRV_DYNAMIC_BUILD
```

Description

I2C Driver Build Configuration Type

This definition selects if I2C device driver is to be used with static or dynamic build parameters. Must be equated to one of the following values:

- DRV_STATIC_BUILD - Build the driver using static accesses to the peripheral identified by the DRV_I2C_INSTANCE macro
- DRV_DYNAMIC_BUILD - Build the driver using dynamic accesses to the peripherals

Affects all the drv_i2c.h driver functions.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BASIC Macro

Enables the device driver to support basic transfer mode.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BASIC
```

Description

Support Basic Transfer Mode

This definition enables the device driver to support basic transfer mode.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_TransmitThenReceive

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BLOCKING Macro

Enables the device driver to support blocking operations.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BLOCKING
```

Description

Support Blocking Operations

This definition enables the device driver to support blocking operations.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_Open
- DRV_I2C_Close
- DRV_I2C_Receive

- DRV_I2C_Transmit
- DRV_I2C_TransmitThenReceive

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_EXCLUSIVE Macro

Enables the device driver to support operation in Exclusive mode.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_EXCLUSIVE
```

Description

Support Exclusive Mode

This definition enables the device driver to support operation in Exclusive mode.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_Open

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_MASTER Macro

Enables the device driver to support operation in Master mode.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_MASTER
```

Description

Support Master Mode

This definition enables the device driver to support operation in Master mode.

Remarks

During the configuration phase, the driver selects a list of operation modes that can be supported. While initializing a hardware instance, the device driver will properly perform the initialization base on the selected modes.

The device driver can support multiple modes within a single build.

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_NON_BLOCKING Macro

Enables the device driver to support non-blocking during operations

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_NON_BLOCKING
```

Description

Support Non-Blocking Operations

This definition enables the device driver to support non-blocking operations.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_Open
- DRV_I2C_Close

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_READ Macro

Enables the device driver to support read operations.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_READ
```

Description

Support Read Mode

This definition enables the device driver to support read operations.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_Receive

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_SLAVE Macro

Enables the device driver to support operation in Slave mode.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_SLAVE
```

Description

Support Slave Mode

This definition enables the device driver to support operation in Slave mode.

Remarks

During the configuration phase, the driver selects a list of operation modes that can be supported. While initializing a hardware instance, the device driver will properly perform the initialization base on the selected modes.

The device driver can support multiple modes within a single build.

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE Macro

Enables the device driver to support write operations.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE
```

Description

Support Write Mode

This definition enables the device driver to support write operations.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_Transmit

Refer to the description of each function in the corresponding help file for details.

DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE_READ Macro

Enables the device driver to support write followed by read.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE_READ
```

Description

Support Write followed by a Read using Restart

This definition enables the device driver to support write followed by read without relinquishing control of the bus. Restart is issued instead of Stop at the end of write. Stop is issued after read operation.

Remarks

The device driver can support multiple modes within a single build.

This definition affects the following functions:

- DRV_I2C_TransmitThenReceive

Refer to the description of each function in the corresponding help file for details.

DRV_STATIC_BUILD Macro

Static driver build, static device instance parameters.

File

drv_i2c_config_template.h

C

```
#define DRV_STATIC_BUILD 0
```

Description

Static Driver Build Configuration

This value, if used to identify the build type for a driver, will cause the driver to be built using a specific statically identified instance of the peripheral.

DRV_I2C_FORCED_WRITE Macro

Includes function that writes to slave irrespective of whether receiving a ACK or NACK from slave

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_FORCED_WRITE true
```

Description

I2C driver objects configuration

When this option is checked, this will include Forced Write function. The Force Write function will send all data bytes to the slave irrespective of receiving ACK or NACK from slave. If writing data to the slave is invoked using DRV_I2C_Transfer, the transaction will be aborted if the Slave NACKs address or any data byte and a STOP condition will be send. This function is typically included for Slaves that require a special reset sequence.

Remarks

None

I2C_STATIC_DRIVER_MODE Macro

Selects the type of STATIC driver

File

drv_i2c_config_template.h

C

```
#define I2C_STATIC_DRIVER_MODE BUFFER_MODEL_STATIC
```

Description

I2C Static Driver type

This selects either the BYTE_MODEL_STATIC or BUFFER_MODEL_STATIC version of I2C driver. The BYTE_MODEL_STATIC version is equivalent to and is referred to as STATIC driver implementation in Harmony Versions 1.06.02 and below. This version of STATIC driver is not recommended for new design and will be deprecated in future release. The BUFFER_MODEL_STATIC supports transfer of buffers and is API compatible with the DYNAMIC implementation of I2C.

Building the Library

This section lists the files that are available in the I2C Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/i2c`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_i2c.h`	This file provides the interface definitions of the I2C driver.
`/drv_i2c_bb.h`	This file provides interface definitions that are transparent to the user when the I2C Driver is used in Bit-bang mode.
`/src/drv_i2c_local.h`	This file provides definitions of the data types that are used in the driver object.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_i2c.c`	This file contains the core implementation of the I2C driver.
`/src/dynamic/drv_i2c_bb.c`	This file implements the I2C Driver in Bit-bang mode.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files exist for this library.

Module Dependencies

The I2C Driver Library depends on the following modules:

- Clock System Service Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_I2C_Deinitialize	Deinitializes the index instance of the I2C module. **Implementation:** Static/Dynamic

	DRV_I2C_Initialize	Initializes hardware and data for the index instance of the I2C module. **Implementation:** Static/Dynamic
	DRV_I2C_Tasks	Maintains the State Machine of the I2C driver and performs all the protocol level actions. **Implementation:** Dynamic

b) Client Setup Functions

	Name	Description
	DRV_I2C_Close	Closes an opened instance of an I2C module driver. **Implementation:** Dynamic
	DRV_I2C_Open	Opens the specified instance of the I2C driver for use and provides an "open-instance" handle. **Implementation:** Dynamic

c) Data Transfer Functions

	Name	Description
	DRV_I2C_BufferEventHandlerSet	Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_I2C_BytesTransferred	Returns the number of bytes transmitted or received in a particular I2C transaction. The transaction is identified by the handle.
	DRV_I2C_Receive	This function reads data written from either Master or Slave. **Implementation:** Dynamic
	DRV_I2C_Transmit	This function writes data to Master or Slave. **Implementation:** Dynamic
	DRV_I2C_TransmitThenReceive	This function writes data to Slave, inserts restart and requests read from slave. **Implementation:** Dynamic
	DRV_I2C_TransmitForced	This function writes data to Master or Slave. **Implementation:** Dynamic

d) Status Functions

	Name	Description
	DRV_I2C_TransferStatusGet	Returns status of data transfer when Master or Slave acts either as a transmitter or a receiver. **Implementation:** Dynamic
	DRV_I2C_Status	Provides the current status of the index instance of the I2C module. **Implementation:** Dynamic

e) Miscellaneous Functions

	Name	Description
	DRV_I2C_QueueFlush	The existing transactions in the queue are voided and the queue pointers are reset to their initial state. This renders the queue empty.
	DRV_I2C_SlaveCallbackSet	Allows a client to identify a Slave Callback function for the driver to call back when drivers needs to initiate a read or write operation.

f) Data Types and Constants

	Name	Description
	DRV_I2C_BUFFER_QUEUE_SUPPORT	Specifies if the Buffer Queue support should be enabled.
	DRV_I2C_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_I2C_INTERRUPT_MODE	Macro controls interrupt based operation of the driver
	DRV_I2C_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.
	DRV_I2C_BB_H	This is macro DRV_I2C_BB_H.

Description

This section describes the Application Programming Interface (API) functions of the I2C Driver Library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_I2C_Deinitialize Function

Deinitializes the index instance of the I2C module.

Implementation: Static/Dynamic

File

drv_i2c.h

C

```
void DRV_I2C_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the index instance of the I2C module, disabling its operation (and any hardware for driver modules). It deinitializes only the specified module instance. It also resets all the internal data structures and fields for the specified instance to the default settings.

Remarks

If the module instance has to be used again, DRV_I2C_Initialize should be called again to initialize the module instance structures.

This function may block if the driver is running in an OS environment that supports blocking operations and the driver requires system resources access. However, the routine will NEVER block for hardware I2C access. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_I2C_Status operation. The driver client must always use DRV_I2C_Status to find out when the module is in the ready state.

Preconditions

The DRV_I2C_Initialize function should have been called before calling this function.

Example

```
SYS_STATUS   i2c_status;

DRV_I2C_Deinitialize(I2C_ID_1);

i2c_status = DRV_I2C_Status(I2C_ID_1);
if (SYS_STATUS_BUSY == i2c_status)
{
    // Do something else and check back later
}
else if (SYS_STATUS_ERROR >= i2c_status)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Index, identifying the instance of the I2C module to be deinitialized

Function

void DRV_I2C_Deinitialize (SYS_MODULE_OBJ object)

DRV_I2C_Initialize Function

Initializes hardware and data for the index instance of the I2C module.

Implementation: Static/Dynamic

File

drv_i2c.h

C

```
SYS_MODULE_OBJ DRV_I2C_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

None.

Description

This function initializes hardware for the index instance of the I2C module, using the hardware initialization given data. It also initializes any internal driver data structures making the driver ready to be opened.

Remarks

This function must be called before any other I2C function is called.

This function should only be called once during system initialization unless DRV_I2C_Deinitialize is first called to deinitialize the device instance before reinitializing it.

This function may block if the driver is running in an OS environment that supports blocking operations and the driver requires system resources access. However, the routine will NEVER block for hardware I2C access. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_I2C_Status operation. The driver client must always use DRV_I2C_Status to find out when the module is in the ready state.

Whenever a call to DRV_I2C_Initialize is made with a SYS_MODULE_INIT* data == 0 the following default configuration will be used. Adjust this configuration at build time as needed.

Preconditions

None.

Example

```
DRV_I2C_INIT        i2c_init_data;
SYS_MODULE_OBJ      objectHandle;

i2c_init_data.i2cId = DRV_I2C_PERIPHERAL_ID_IDX0,
i2c_init_data.i2cMode = DRV_I2C_MODE_MASTER,

OR

i2c_init_data.i2cMode = DRV_I2C_MODE_SLAVE,
//Master mode parameters
i2c_init_data.baudRate = 100000,
i2c_init_data.busspeed = DRV_I2C_SLEW_RATE_CONTROL_IDX0,
i2c_init_data.buslevel = DRV_I2C_SMBus_SPECIFICATION_IDX0,

//Slave mode parameters
i2c_init_data.addWidth = DRV_I2C_7BIT_SLAVE,
i2c_init_data.reservedaddenable  = false,
i2c_init_data.generalcalladdress = false,
i2c_init_data.slaveaddvalue = 0x0060,

//interrupt sources
i2c_init_data.mstrInterruptSource = INT_SOURCE_I2C_2_MASTER,
i2c_init_data.slaveInterruptSource = INT_SOURCE_I2C_2_ERROR,
i2c_init_data.errInterruptSource = INT_SOURCE_I2C_2_ERROR,
i2c_init_data.queueSize = 1,

//callback for Master (Master mode can use callbacks if needed)
i2c_init_data.operationStarting = NULL,
// Slave mode callbacks needed
i2c_init_data.operationStarting = APP_I2CSlaveFunction(),
i2c_init_data.operationEnded = NULL

objectHandle = DRV_I2C_Initialize(DRV_I2C_INDEX_0, (SYS_MODULE_INIT *)&drvI2C0InitData)
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Index, identifying the instance of the I2C module to be initialized

data	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and the default initialization is to be used.

Function

void DRV_I2C_Initialize (const I2C_MODULE_ID index,
const SYS_MODULE_INIT *const data)

DRV_I2C_Tasks Function

Maintains the State Machine of the I2C driver and performs all the protocol level actions.

Implementation: Dynamic

File

drv_i2c.h

C

```
void DRV_I2C_Tasks(SYS_MODULE_OBJ object);
```

Description

This functions maintains the internal state machine of the I2C driver. This function acts as the I2C Master or Slave ISR. When used in polling mode, this function needs to be called repeatedly to achieve I2C data transfer. This function implements all the protocol level details like setting the START condition, sending the address with with R/W request, writing data to the SFR, checking for acknowledge and setting the STOP condition.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance.

Example

SYS_MODULE_OBJ object;
while (true) { DRV_I2C_Tasks (object);

Function

void DRV_I2C_Tasks (SYS_MODULE_OBJ object)

b) Client Setup Functions

DRV_I2C_Close Function

Closes an opened instance of an I2C module driver.

Implementation: Dynamic

File

drv_i2c.h

C

```
void DRV_I2C_Close(DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened instance of an I2C module driver, making the specified handle invalid.

Remarks

After calling This function, the handle passed into drvHandle must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_I2C_Open before the caller may use the driver again.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

Example

```
myI2CHandle = DRV_I2C_Open(I2C_ID_1, DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE);

// Perform data transfer operations

DRV_I2C_Close(myI2CHandle);
```

Parameters

Parameters	Description
drvHandle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_I2C_Close (const DRV_HANDLE drvHandle)

DRV_I2C_Open Function

Opens the specified instance of the I2C driver for use and provides an "open-instance" handle.

Implementation: Dynamic

File

drv_i2c.h

C

```
DRV_HANDLE DRV_I2C_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Returns

If successful, the routine returns a valid open-instance handle (a value identifying both the caller and the module instance). If an error occurs, the returned value is DRV_HANDLE_INVALID.

If an error occurs, the return value is DRV_HANDLE_INVALID. An error can occur when the following is true:

- if the number of client objects allocated via DRV_I2C_INSTANCES_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid

Description

This function opens the specified instance of the I2C module for use and provides a handle that is required to use the remaining driver routines.

This function opens a specified instance of the I2C module driver for use by any client module and provides an "open-instance" handle that must be provided to any of the other I2C driver operations to identify the caller and the instance of the I2C driver/hardware module.

Remarks

The handle returned is valid until the DRV_I2C_Close routine is called.

This function may block if the driver is running in an OS environment that supports blocking operations and the driver requires system resources access. Regarding the hardware I2C access the operation will behave as instructed by the DRV_IO_INTENT parameter.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

Example

```
DRV_HANDLE              i2c_handle;

i2c_handle = DRV_I2C_Open(I2C_ID_1, DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE);
if (DRV_HANDLE_INVALID == i2c_handle)
{
    // Handle open error
}

// Close the device when it is no longer needed.
DRV_I2C_Close(i2c_handle);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the I2C module to be opened.
intent	Flags parameter identifying the intended usage and behavior of the driver. Multiple flags may be ORed together to specify the intended usage of the device. See the DRV_IO_INTENT definition.

Function

DRV_HANDLE DRV_I2C_Open (const I2C_MODULE_ID index,
const DRV_IO_INTENT intent)

c) Data Transfer Functions

DRV_I2C_BufferEventHandlerSet Function

Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

Implementation: Dynamic

File

drv_i2c.h

C

```
void DRV_I2C_BufferEventHandlerSet(const DRV_HANDLE handle, const DRV_I2C_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when a queued buffer transfer has finished. When a client calls either the DRV_I2C_Receive, DRV_I2C_Transmit or DRV_I2C_TransmiThenReceive function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any transmision or reception operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

When in Master mode, a callback event is registered to let the application know that the buffer has been transmitted. DRV_I2C_BUFFER_EVENT_COMPLETE is set when the buffer has been transmitted without any errors. DRV_I2C_BUFFER_EVENT_ERROR is set when buffer transmission or reception has been aborted.

When in Slave mode, since the Master controls when a transmit or receive operation is terminated, a callback is registered every time a byte is written or read from the slave.

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C driver instance.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

Example

```
#define MY_BUFFER_SIZE  10

// function prototype of Event Handler Function
void APP_I2CBufferEventFunction ( DRV_I2C_BUFFER_EVENT event,
                                  DRV_I2C_BUFFER_HANDLE bufferHandle,
                                  uintptr_t context );

//Returned from DRV_I2C_Open
DRV_HANDLE drvI2Chandle;

// myAppObj is an application specific state data object.
```

```
DRV_I2C_BUFFER_EVENT operationStatus;

uint8_t appBuffer[MY_BUFFER_SIZE];

DRV_I2C_BUFFER_HANDLE drvI2CRDBUFHandle

// Opens an instance of I2C driver
drvI2Chandle = DRV_I2C_Open( DRV_I2C_INDEX_0,DRV_IO_INTENT_WRITE );

// Client registers an event handler with driver. This is done once.
DRV_I2C_BufferEventHandlerSet( drvI2Chandle,
                               APP_I2CBufferEventFunction,
                               operationStatus );

drvI2CRDBUFHandle = DRV_I2C_Receive (   drvI2CHandle,
                                        slaveaddress
                                        &appBuffer[],
                                        MY_BUFFER_SIZE,
                                        NULL );

if(NULL == drvI2CRDBUFHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2CBufferEventFunction( DRV_I2C_BUFFER_EVENT event,
                                 DRV_I2C_BUFFER_HANDLE handle,
                                 uintptr_t context)
{

    switch(event)
    {
        case DRV_I2C_BUFFER_EVENT_COMPLETE:
            //perform appropriate action
        break;

        case DRV_I2C_BUFFER_EVENT_ERROR:
            // Error handling here.
        break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_I2C_BufferEventHandlerSet (

const DRV_HANDLE handle,

const DRV_I2C_BUFFER_EVENT_HANDLER eventHandler,

const uintptr_t context)

DRV_I2C_BytesTransferred Function

Returns the number of bytes transmitted or received in a particular I2C transaction. The transaction is identified by the handle.

File

drv_i2c.h

C

```
uint32_t DRV_I2C_BytesTransferred(DRV_HANDLE handle, DRV_I2C_BUFFER_HANDLE bufferHandle);
```

Returns

The number of bytes transferred in a particular I2C transaction.
`numOfBytes = DRV_I2C_BytesTransferred (drvI2CHandle_Master,drvBufferHandle);`

Description

This returns the transmitter and receiver transfer status.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
bufferHandle	A valid buffer handle obtained when calling Transmit/Receive/TransmitThenReceive/TransmitForced or BufferAddRead/BufferAddWrite/BufferAddReadWrite function

Function

uint32_t DRV_I2C_BytesTransferred (DRV_I2C_BUFFER_HANDLE bufferHandle)

DRV_I2C_Receive Function

This function reads data written from either Master or Slave.

Implementation: Dynamic

File

drv_i2c.h

C

```
DRV_I2C_BUFFER_HANDLE DRV_I2C_Receive(DRV_HANDLE handle, uint16_t address, void * buffer, size_t size, void
* callbackContext);
```

Returns

A valid BUFFER HANDLE, NULL if the handle is not obtained.

Description

Master calls this function to read data send by Slave. The Slave calls this function to read data send by Master. In case of Master, a START condition is initiated on the I2C bus.

Remarks

The handle that is passed into the function, drvI2CHandle is obtained by calling the DRV_I2C_OPEN function. If the function could not return a valid buffer handle, then a NULL value is returned. If the slave NACKs the address byte, then further read is not attempted. Master asserts STOP condition and DRV_I2C_BUFFER_EVENT_ERROR is set as the buffer-status. If all the requisite number of bytes have been read then DRV_I2C_BUFFER_EVENT_COMPLETE is set as the buffer status.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

Example

```
drvI2CRDBUFHandle = DRV_I2C_Receive( drvI2CHandle,
                                     deviceaddress,
                                     &rxbuffer[0],
                                     num_of_bytes,
                                     NULL );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
address	Device address of slave shifted so that bits 7 - 1 are address
bits A6	A0. This value is Ignored in slave mode.
buffer	This buffer holds data is received
size	The number of bytes that the Master expects to read from Slave. This value can be kept as the MAX BUFFER SIZE for slave. This is because the Master controls when the READ operation is terminated.
callbackContext	Not implemented, future expansion

Function

DRV_I2C_BUFFER_HANDLE DRV_I2C_Receive (DRV_HANDLE handle,

uint16_t slaveaddress,

void *rxBuffer,

size_t size,

void * callbackContext)

DRV_I2C_Transmit Function

This function writes data to Master or Slave.

Implementation: Dynamic

File

drv_i2c.h

C

```
DRV_I2C_BUFFER_HANDLE DRV_I2C_Transmit(DRV_HANDLE handle, uint16_t slaveaddress, void * buffer, size_t
size, void * context);
```

Returns

A valid BUFFER HANDLE, NULL if the handle is not obtained.

Description

Master calls this function to write data to Slave. The Slave calls this function to write data to Master.

Remarks

The handle that is passed into the function, drvI2CHandle is obtained by calling the DRV_I2C_OPEN function. If the function could not return a valid buffer handle, then a NULL value is returned. If the slave NACKs the address byte or any data bytes, then further write is not attempted. Master asserts STOP condition and DRV_I2C_BUFFER_EVENT_ERROR is set as the buffer-status. If all the requisite number of bytes have been transmitted to the Slave, then DRV_I2C_BUFFER_EVENT_COMPLETE is set as the buffer status.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

Example

```
drvI2CWRBUFHandle = DRV_I2C_Transmit( drvI2CHandle,
                                      deviceaddress,
                                      &txBuffer[0],
                                      num_of_bytes,
                                      NULL);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
address	Device address of slave shifted so that bits 7 - 1 are address
bits A6	A0. This value is Ignored in slave mode.
buffer	Contains data to be transferred

size	The number of bytes that the Master expects to write to Slave. This value can be kept as the MAX BUFFER SIZE for slave. This is because the Master controls when the WRITE operation is terminated.
callbackContext	Not implemented, future expansion

Function

DRV_I2C_BUFFER_HANDLE DRV_I2C_Transmit(DRV_HANDLE handle,
uint16_t slaveaddress,
void *txBuffer,
size_t size,
void *context);

DRV_I2C_TransmitThenReceive Function

This function writes data to Slave, inserts restart and requests read from slave.

Implementation: Dynamic

File

drv_i2c.h

C

```
DRV_I2C_BUFFER_HANDLE DRV_I2C_TransmitThenReceive(DRV_HANDLE handle, uint16_t address, void * writeBuffer,
size_t writeSize, void * readBuffer, size_t readSize, void * callbackContext);
```

Returns

A valid BUFFER HANDLE, NULL if the handle is not obtained.

Description

Master calls this function to send a register address value to the slave and then queries the slave with a read request to read the contents indexed by the register location. The Master sends a restart condition after the initial write before sending the device address with R/W = 1. The restart condition prevents the Master from relinquishing the control of the bus. The slave should not use this function.

Remarks

The handle that is passed into the function, drvI2CHandle is obtained by calling the DRV_I2C_OPEN function. If the function could not return a valid buffer handle, then a NULL value is returned. If there is any error condition during transmission then further transmission or reception is not attempted and STOP condition is asserted on the bus. In case of error condition, DRV_I2C_BUFFER_EVENT_ERROR is set as the buffer-status. If the I2C bus transaction is completed as requested then the buffer status, is set as DRV_I2C_BUFFER_EVENT_COMPLETE.

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

Example

```
drvI2CRDBUFHandle = DRV_I2C_TransmiThenReceive( appData.drvI2CHandle,
                                                deviceaddress,
                                                &drvI2CTXbuffer[0],
                                                registerbytesize,
                                                rxbuffer,
                                                num_of_bytes,
                                                NULL );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
address	Device address of slave shifted so that bits 7 - 1 are address
bits A6	A0. This value is Ignored in slave mode.
writeBuffer	Contains data to be transferred
writeSize	The number of bytes that the Master expects to write to Slave. This value can be kept as the MAX BUFFER SIZE for slave. This is because the Master controls when the WRITE operation is terminated.
readBuffer	This buffer holds data that is send back from slave after read operation.

readSize	The number of bytes the Master expects to be read from the slave
callbackContext	Not implemented, future expansion

Function

DRV_I2C_BUFFER_HANDLE DRV_I2C_TransmiThenReceive (DRV_HANDLE handle,
uint16_t deviceaddress,
void *txBuffer,
size_t writeSize,
void *rxBuffer,
size_t readSize,
void *context)

DRV_I2C_TransmitForced Function

This function writes data to Master or Slave.

Implementation: Dynamic

File

drv_i2c.h

C

```
DRV_I2C_BUFFER_HANDLE DRV_I2C_TransmitForced(DRV_HANDLE handle, uint16_t deviceaddress, void* txBuffer,
size_t txbuflen, DRV_I2C_BUS_ERROR_EVENT eventFlag, void * calbackContext);
```

Returns

A valid BUFFER HANDLE, NULL if the handle is not obtained.

Description

Master calls this function to transmit the entire buffer to the slave even if the slave ACKs or NACKs the address or any of the data bytes. This is typically used for slaves that have to initiate a reset sequence by sending a dummy I2C transaction. Since the slave is still in reset, any or all the bytes can be NACKed. In the normal operation of the driver if the address or data byte is NACKed, then the transmission is aborted and a STOP condition is asserted on the bus.

Remarks

The handle that is passed into the function, drvI2CHandle is obtained by calling the DRV_I2C_OPEN function. If the function could not return a valid buffer handle, then a NULl value is returned. Once all the bytes are transferred the buffer status is set as then DRV_I2C_BUFFER_EVENT_COMPLETE .

Preconditions

The DRV_I2C_Initialize routine must have been called for the specified I2C device instance and the DRV_I2C_Status must have returned SYS_STATUS_READY.

DRV_I2C_Open must have been called to obtain a valid opened device handle.

Example

```
drvI2CWRBUFHandle = DRV_I2C_TransmitForced ( handle,
                                             deviceaddress,
                                             &txBuffer[0],
                                             txbuflen,
                                             NULL,
                                             NULL)
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
address	Device address of slave shifted so that bits 7 - 1 are address
bits A6	A0. This value is Ignored in slave mode.
buffer	Contains data to be transferred
size	The number of bytes that the Master expects to write to Slave. This value can be kept as the MAX BUFFER SIZE for slave. This is because the Master controls when the WRITE operation is terminated.
eventFlag	This field is left for future implementation

callbackContext	Not implemented, future expansion

Function

DRV_I2C_BUFFER_HANDLE DRV_I2C_TransmitForced (DRV_HANDLE handle,
uint16_t deviceaddress,
uint8_t* txBuffer,
uint16_t txbuflen,
DRV_I2C_BUS_ERROR_EVENT eventFlag,
void * calbackContext)

d) Status Functions

DRV_I2C_TransferStatusGet Function

Returns status of data transfer when Master or Slave acts either as a transmitter or a receiver.

Implementation: Dynamic

File

drv_i2c.h

C

```
DRV_I2C_BUFFER_EVENT DRV_I2C_TransferStatusGet(DRV_HANDLE handle, DRV_I2C_BUFFER_HANDLE bufferHandle);
```

Returns

A DRV_I2C_TRANSFER_STATUS value describing the current status of the transfer.

Description

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event. If the event is DRV_I2C_BUFFER_EVENT_COMPLETE, it means that the data was transferred successfully. If the event is DRV_I2C_BUFFER_EVENT_ERROR, it means that the data was not transferred successfully.

Remarks

The handle that is passed into the function, drvI2CBUFHandle is obtained by calling one of the data transfer functions. The drvI2CBUFHandle should be a valid handle and not a NULL value. The DRV_I2C_BufferStatus can be called to check the progress of the data transfer operation. If the buffer is transferred without any error, then DRV_I2C_BUFFER_EVENT_COMPLETE is returned. If an error condition is present, then DRV_I2C_BUFFER_EVENT_ERROR is returned.

Example

```
if(DRV_I2C_BUFFER_EVENT_COMPLETE == DRV_I2C_TransferStatusGet ( handle,
                                                                bufferHandle ))
{
        //perform action
        return true;
}
else
{
        //perform action
        return false;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
bufferHandle	A valid buffer handle obtained when calling Transmit/Receive/TransmitThenReceive/TransmitForced or BufferAddRead/BufferAddWrite/BufferAddReadWrite function

Function

DRV_I2C_BUFFER_EVENT DRV_I2C_TransferStatusGet (DRV_HANDLE handle,
DRV_I2C_BUFFER_HANDLE bufferHandle)

DRV_I2C_Status Function

Provides the current status of the index instance of the I2C module.

Implementation: Dynamic

File

drv_i2c.h

C

```
SYS_STATUS DRV_I2C_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that any previous module operation for the specified I2Cmodule has completed.
- SYS_STATUS_BUSY - Indicates that a previous module operation for the specified I2C module has not yet completed
- SYS_STATUS_ERROR - Indicates that the specified I2C module is in an error state

Description

This function provides the current status of the index instance of the I2C module.

Remarks

The DRV_I2C_Status operation can be used to determine when any of the I2C module level operations has completed. The value returned by the DRV_I2C_Status routine hast to be checked after calling any of the I2C module operations to find out when they have completed.

If the DRV_I2C_Status operation returns SYS_STATUS_BUSY, the previous operation has not yet completed. Once the DRV_I2C_Status operation return SYS_STATUS_READY, any previous operations have completed.

The DRV_I2C_Status function will NEVER block.

If the DRV_I2C_Status operation returns an error value, the error may be cleared by calling the DRV_I2C_Initialize operation. If that fails, the DRV_I2C_Deinitialize operation will need to be called, followed by the DRV_I2C_Initialize operation to return to normal operations.

Preconditions

The DRV_I2C_Initialize function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;
SYS_STATUS          i2c_status;

i2c_status = DRV_I2C_Status(object);
if (SYS_STATUS_BUSY == i2c_status)
{
    // Do something else and check back later
}
else if (SYS_STATUS_ERROR >= status)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Index, identifying the instance of the I2C module to get status for.

Function

SYS_STATUS DRV_I2C_Status (SYS_MODULE_OBJ object)

e) Miscellaneous Functions

DRV_I2C_QueueFlush Function

The existing transactions in the queue are voided and the queue pointers are reset to their initial state. This renders the queue empty.

File

drv_i2c.h

C

```
void DRV_I2C_QueueFlush(DRV_HANDLE handle);
```

Returns

None

```
    //Opens an instance of I2C driver
 drvI2Chandle = DRV_I2C_Open( DRV_I2C_INDEX_0,DRV_IO_INTENT_WRITE );

 DRV_I2C_QueueFlush ( drvI2CHandle );
```

Description

The existing transactions in the queue are voided and the queue pointers are reset to their initial state. This renders the queue empty.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_I2C_QueueFlush (DRV_HANDLE handle)

DRV_I2C_SlaveCallbackSet Function

Allows a client to identify a Slave Callback function for the driver to call back when drivers needs to initiate a read or write operation.

File

drv_i2c.h

C

```
void DRV_I2C_SlaveCallbackSet(const DRV_HANDLE handle, const DRV_I2C_CallBack callback, const uintptr_t
context);
```

Returns

None

```
 #define APP_I2C_BUFFER_SIZE      10

 void APP_I2C_SlaveTransferCallback( DRV_I2C_BUFFER_EVENT event, void * context );

 uint8_t slaveBuffer[ APP_I2C_BUFFER_SIZE ];

 DRV_HANDLE drvI2CHandle;

 void APP_I2C_SlaveTransferCallback( DRV_I2C_BUFFER_EVENT event, void * context )
 {
     switch (event)
     {
         case DRV_I2C_BUFFER_SLAVE_READ_REQUESTED:
         {
             appData.bufferReceive = DRV_I2C_Receive( drvI2CHandle,
                                                      0,
                                                      &slaveBuffer[0],
                                                      APP_I2C_BUFFER_SIZE,
                                                      NULL );
             break;
         }

         case DRV_I2C_BUFFER_SLAVE_WRITE_REQUESTED:
         {
             appData.bufferTransmit = DRV_I2C_Transmit ( drvI2CHandle,
                                                         0,
                                                         &slaveBuffer[0],
                                                         APP_I2C_BUFFER_SIZE,
```

```
                                                    NULL );
            break;
        }

        default:
        {
            break;
        }
    }

    return;
}

// Opens an instance of I2C driver
drvI2Chandle = DRV_I2C_Open( DRV_I2C_INDEX_0, DRV_IO_INTENT_READWRITE );

// Set the operation callback
DRV_I2C_SlaveCallbackSet( drvI2Chandle, APP_I2C_SlaveTransferCallback, 0);
```

Description

This function allows a client to identify a Slave Callback function for the driver to call back when drivers needs to initiate a read or write operation. When the I2C Slave driver receives a read or write event from master the callback is called to initiate the user defined action. The callback should be set before the master requests for read or write operations. The event handler once set, persists until the client closes the driver or sets another event handler.

In Slave mode, a callback event is registered to let the application know that master has requested for either read or write operation. DRV_I2C_BUFFER_SLAVE_READ_REQUESTED is set when the master sends data and wants slave to read. DRV_I2C_BUFFER_SLAVE_WRITE_REQUESTED is set when the master tries to read data from slave and wants slave to send the data.

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
callback	pointer to the callback function.
context	The value of parameter will be passed back to the client unchanged, when the callback function is called. It can be used to identify any client specific data

Function

void DRV_I2C_SlaveCallbackSet (const DRV_HANDLE handle,

const DRV_I2C_CallBack callback,

const uintptr_t context)

f) Data Types and Constants

DRV_I2C_BUFFER_QUEUE_SUPPORT Macro

Specifies if the Buffer Queue support should be enabled.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_BUFFER_QUEUE_SUPPORT false
```

Description

I2C Driver Buffer Queue Support

This macro defines if Buffer Queue support should be enabled. Setting this macro to true will enable buffer queue support and all buffer related driver function.

Remarks

None

DRV_I2C_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_INSTANCES_NUMBER 5
```

Description

I2C driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of I2C modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None

DRV_I2C_INTERRUPT_MODE Macro

Macro controls interrupt based operation of the driver

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_INTERRUPT_MODE true
```

Description

I2C Interrupt Mode Operation Control

This macro controls the interrupt based operation of the driver. The possible values it can take are

- true - Enables the interrupt mode
- false - Enables the polling mode

If the macro value is true, then Interrupt Service Routine for the interrupt should be defined in the application. The DRV_I2C_Tasks() routine should be called in the ISR.

Remarks

None

DRV_I2C_QUEUE_DEPTH_COMBINED Macro

Number of entries of all queues in all instances of the driver.

File

drv_i2c_config_template.h

C

```
#define DRV_I2C_QUEUE_DEPTH_COMBINED 7
```

Description

I2C Driver Instance combined queue depth.

This macro defines the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for transmit and receive operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build). The hardware instance transmit buffer queue will queue transmit buffers submitted by the DRV_I2C_Transmit() function. The hardware instance receive buffer queue will queue receive buffers submitted by the DRV_I2C_Receive() function.

A buffer queue will contain buffer queue entries, each related to a BufferAdd request. This configuration macro defines total number of buffer entries that will be available for use between all I2C driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking read and write requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its transmit and receive buffer queue size.

As an example, consider the case of static single client driver application where full duplex non blocking operation is desired without queuing, the minimum transmit queue depth and minimum receive queue depth should be 1. Hence the total number of buffer entries should be 2.

In the current implementation of I2C driver, queueing of Buffers is not supported. This will be added in a future release.

Remarks

None

DRV_I2C_BB_H Macro

File

drv_i2c_bb.h

C

```
#define DRV_I2C_BB_H
```

Description

This is macro DRV_I2C_BB_H.

Files

Files

Name	Description
drv_i2c.h	I2C module driver interface header.
drv_i2c_bb.h	Contains prototypes for the I2C functions
drv_i2c_config_template.h	I2C device driver configuration file.

Description

drv_i2c.h

I2C module driver interface header.

Functions

	Name	Description
	DRV_I2C_BufferEventHandlerSet	Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_I2C_BytesTransferred	Returns the number of bytes transmitted or received in a particular I2C transaction. The transaction is identified by the handle.
	DRV_I2C_Close	Closes an opened instance of an I2C module driver. **Implementation:** Dynamic
	DRV_I2C_Deinitialize	Deinitializes the index instance of the I2C module. **Implementation:** Static/Dynamic
	DRV_I2C_Initialize	Initializes hardware and data for the index instance of the I2C module. **Implementation:** Static/Dynamic
	DRV_I2C_Open	Opens the specified instance of the I2C driver for use and provides an "open-instance" handle. **Implementation:** Dynamic
	DRV_I2C_QueueFlush	The existing transactions in the queue are voided and the queue pointers are reset to their initial state. This renders the queue empty.
	DRV_I2C_Receive	This function reads data written from either Master or Slave. **Implementation:** Dynamic
	DRV_I2C_SlaveCallbackSet	Allows a client to identify a Slave Callback function for the driver to call back when drivers needs to initiate a read or write operation.

	DRV_I2C_Status	Provides the current status of the index instance of the I2C module. **Implementation:** Dynamic
	DRV_I2C_Tasks	Maintains the State Machine of the I2C driver and performs all the protocol level actions. **Implementation:** Dynamic
	DRV_I2C_TransferStatusGet	Returns status of data transfer when Master or Slave acts either as a transmitter or a receiver. **Implementation:** Dynamic
	DRV_I2C_Transmit	This function writes data to Master or Slave. **Implementation:** Dynamic
	DRV_I2C_TransmitForced	This function writes data to Master or Slave. **Implementation:** Dynamic
	DRV_I2C_TransmitThenReceive	This function writes data to Slave, inserts restart and requests read from slave. **Implementation:** Dynamic

Description

I2C Device Driver Interface Header File

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the I2C module driver.

File Name

drv_i2c.h

Company

Microchip Technology Inc.

drv_i2c_bb.h

Contains prototypes for the I2C functions

Macros

	Name	Description
	DRV_I2C_BB_H	This is macro DRV_I2C_BB_H.

Description

I2C Bit Bang Functions Header File

File Name

drv_i2c_bb.h

Company

Microchip Technology Inc.

drv_i2c_config_template.h

I2C device driver configuration file.

Macros

	Name	Description
	DRV_DYNAMIC_BUILD	Dynamic driver build, dynamic device instance parameters.
	DRV_I2C_BUFFER_QUEUE_SUPPORT	Specifies if the Buffer Queue support should be enabled.
	DRV_I2C_CONFIG_BUILD_TYPE	Selects static or dynamic driver build configuration.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BASIC	Enables the device driver to support basic transfer mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_BLOCKING	Enables the device driver to support blocking operations.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_EXCLUSIVE	Enables the device driver to support operation in Exclusive mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_MASTER	Enables the device driver to support operation in Master mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_NON_BLOCKING	Enables the device driver to support non-blocking during operations

	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_READ	Enables the device driver to support read operations.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_SLAVE	Enables the device driver to support operation in Slave mode.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE	Enables the device driver to support write operations.
	DRV_I2C_CONFIG_SUPPORT_OPERATION_MODE_WRITE_READ	Enables the device driver to support write followed by read.
	DRV_I2C_FORCED_WRITE	Includes function that writes to slave irrespective of whether receiving a ACK or NACK from slave
	DRV_I2C_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_I2C_INTERRUPT_MODE	Macro controls interrupt based operation of the driver
	DRV_I2C_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.
	DRV_STATIC_BUILD	Static driver build, static device instance parameters.
	I2C_STATIC_DRIVER_MODE	Selects the type of STATIC driver

Description

I2C Device Driver Configuration

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_i2c_config.h

Company

Microchip Technology Inc.

I2S Driver Library Help

This section describes the I2S Driver Library.

Introduction

This library provides an interface to manage the Audio Protocol Interface Modes of the Serial Peripheral Interface (SPI) module on the Microchip family of microcontrollers.

Description

The SPI module can be interfaced to most available codec devices to provide microcontroller-based audio solutions. The SPI module provides support to the audio protocol functionality via four standard I/O pins. The four pins that make up the audio protocol interface modes are:

- SDIx: Serial Data Input for receiving sample digital audio data (ADCDAT)
- SDOx: Serial Data Output for transmitting digital audio data (DACDAT)
- SCKx: Serial Clock, also known as bit clock (BCLK)
- /SSx: Left/Right Channel Clock (LRCK)

BCLK provides the clock required to drive the data out or into the module, while LRCK provides the synchronization of the frame based on the protocol mode selected.

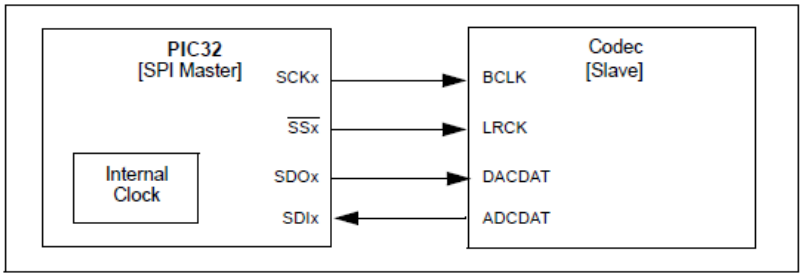
In Master mode, the module generates both the BCLK on the SCKx pin and the LRCK on the /SSx pin. In certain devices, while in Slave mode, the module receives these two clocks from its I2S partner, which is operating in Master mode.

When configured in Master mode, the leading edge of SCK and the LRCK are driven out within one SCK period of starting the audio protocol. Serial data is shifted in or out with timings determined by the protocol mode set.

In Slave mode, the peripheral drives zeros out SDO, but does not transmit the contents of the transmit FIFO until it sees the leading edge of the LRCK, after which time it starts receiving data.

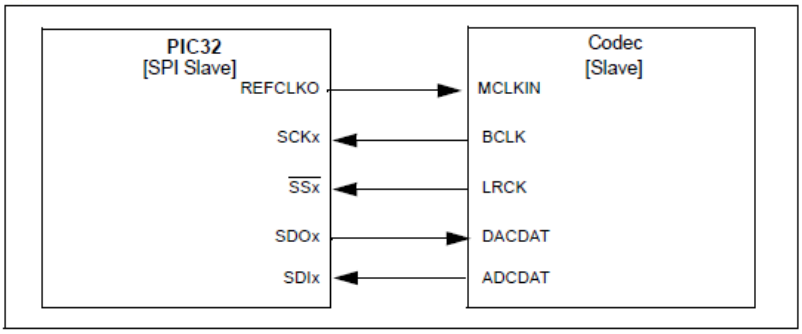
Master Mode

Master Generating its Own Clock – Output BCLK and LRCK



Slave Mode

Codec Device as Master Derives MCLK from PIC32 Reference Clock Out



Audio Protocol Modes

The SPI module supports four audio protocol modes and can be operated in any one of these modes:

- I2S mode
- Left-Justified mode
- Right-Justified mode
- PCM/DSP mode

Using the Library

This topic describes the basic architecture of the I2S Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_i2s.h`

The interface to the I2S Driver Library is defined in the `drv_i2s.h` header file. Any C language source (`.c`) file that uses the I2S Driver Library should include `drv_i2s.h`.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

The SPI Peripheral Library provides the low-level abstraction of the SPI module on the Microchip family of microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the I2S Driver Library interface.

Description

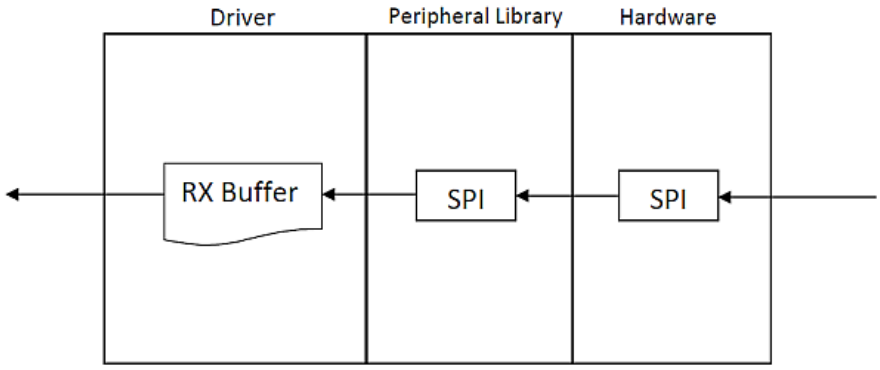
I2S Software Abstraction Block Diagram

Different types of SPIs are available on Microchip microcontrollers. Some have an internal buffer mechanism and some do not. The buffer depth varies across part families. The SPI Peripheral Library provides the ability to access these buffers. The I2S Driver Library abstracts out these differences and provides a unified model for audio data transfer across different types of SPI modules.

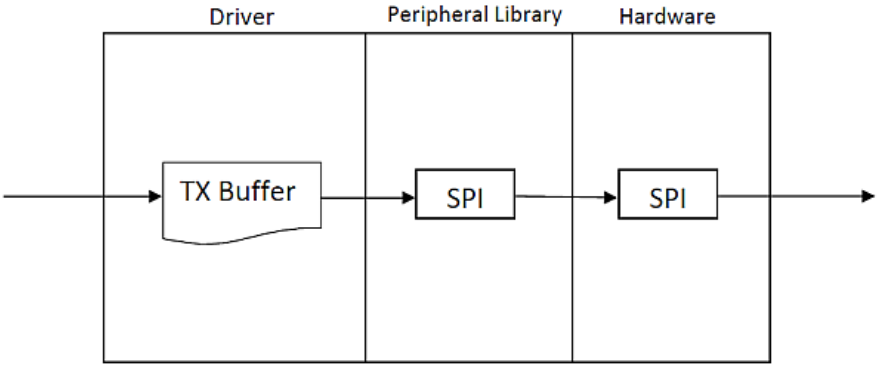
Both the transmitter and receiver provide a buffer in the driver, which transmits and receives data to/from the hardware. The I2S Driver Library provides a set of interfaces to perform the read and the write.

The following diagrams illustrate the model used by the I2S Driver Library for the transmitter and receiver.

Receiver Abstraction Model



Transmitter Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The I2S driver library provides an API interface to transfer/receive digital audio data using supported Audio protocols. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the I2S Driver Library.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open and close functions.
Data Transfer Functions	Provides data transfer functions.
Miscellaneous Functions	Provides driver miscellaneous functions such as baud rate setting, get error functions, etc.
Data Types and Constants	These data types and constants are required while interacting and setting up the I2S Driver Library.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

Note: Not all modes are available on all devices. Please refer to the specific device data sheet to determine the supported modes.

System Access

This section provides information on system access.

Description

System Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the I2S module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_I2S_INIT or by using Initialization Overrides) that are supported by the specific I2S device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- The actual peripheral ID enumerated as the PLIB level module ID (e.g., SPI_ID_2)
- Defining the respective interrupt sources for TX, RX, DMA TX Channel, DMA RX Channel and Error Interrupt

The DRV_I2S_Initialize API returns an object handle of the type SYS_MODULE_OBJ. The object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_I2S_Deinitialize, DRV_I2S_Status, DRV_I2S_Tasks, and DRV_I2S_TasksError.

Notes:

1. The system initialization setting only effect the instance of the peripheral that is being initialized.
2. Configuration of the dynamic driver for DMA mode(uses DMA channel for data transfer) or Non DMA mode can be performed by appropriately setting the 'dmaChannelTransmit' and 'dmaChannelReceive' variables of the DRV_I2S_INIT structure. For example the TX will be in DMA mode when 'dmaChannelTransmit' is initialized to a valid supported channel number from the enum DMA_CHANNEL. TX will be in Non DMA mode when 'dmaChannelTransmit' is initialized to 'DMA_CHANNEL_NONE'.

Example:

```
    DRV_I2S_INIT                  init;
    SYS_MODULE_OBJ              objectHandle;

    init.moduleInit.value           = SYS_MODULE_POWER_RUN_FULL;
    init.spiID                      = SPI_ID_1;
    init.usageMode                  = DRV_I2S_MODE_MASTER;
    init.baudClock                  = SPI_BAUD_RATE_MCLK_CLOCK;
    init.baud                       = 48000;
    init.clockMode                  = DRV_I2S_CLOCK_MODE_IDLE_HIGH_EDGE_FALL;
    init.audioCommWidth             = SPI_AUDIO_COMMUNICATION_24DATA_32FIFO_32CHANNEL;
    init.audioTransmitMode          = SPI_AUDIO_TRANSMIT_STEREO;
    init.inputSamplePhase           = SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE;
    init.protocolMode               = DRV_I2S_AUDIO_I2S;
    init.txInterruptSource          = INT_SOURCE_SPI_1_TRANSMIT;
    init.rxInterruptSource          = INT_SOURCE_SPI_1_RECEIVE;
    init.errorInterruptSource       = INT_SOURCE_SPI_1_ERROR;
    init.queueSizeTransmit          = 3;
    init.queueSizeReceive           = 2;
    init.dmaChannelTransmit         = DMA_CHANNEL_NONE;
    init.dmaChannelReceive          = DMA_CHANNEL_NONE;

  objectHandle = DRV_I2S_Initialize(DRV_I2S_INDEX_1, (SYS_MODULE_INIT*)init);
    if (SYS_MODULE_OBJ_INVALID == objectHandle)
    {
        // Handle error
    }
```

Task Routine

In a polled environment, the system will call DRV_I2S_Tasks and DRV_I2S_TasksError from the System Task Service. In an interrupt-based implementation, DRV_I2S_Tasks and DRV_I2S_TasksError will be called from the Interrupt Service Routine of the I2S. When a DMA channel is used for transmission/reception DRV_I2S_Tasks and DRV_I2S_TasksError will be internally called by the driver from the DMA channel event handler.

Client Access

This section provides information on general client operation.

Description

General Client Operation

For the application to start using an instance of the module, it must call the DRV_I2S_Open function. This provides the settings required to open the I2S instance for operation. If the driver is deinitialized using the function DRV_I2S_Deinitialize, the application must call the DRV_I2S_Open function again to set up the instance of the I2S.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

Example:

```
DRV_HANDLE handle;
handle = DRV_I2S_Open(DRV_I2S_INDEX_0, (DRV_IO_INTENT_WRITE | DRV_IO_INTENT_NONBLOCKING));
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Client Operations - Buffered

This section provides information on buffered client operations.

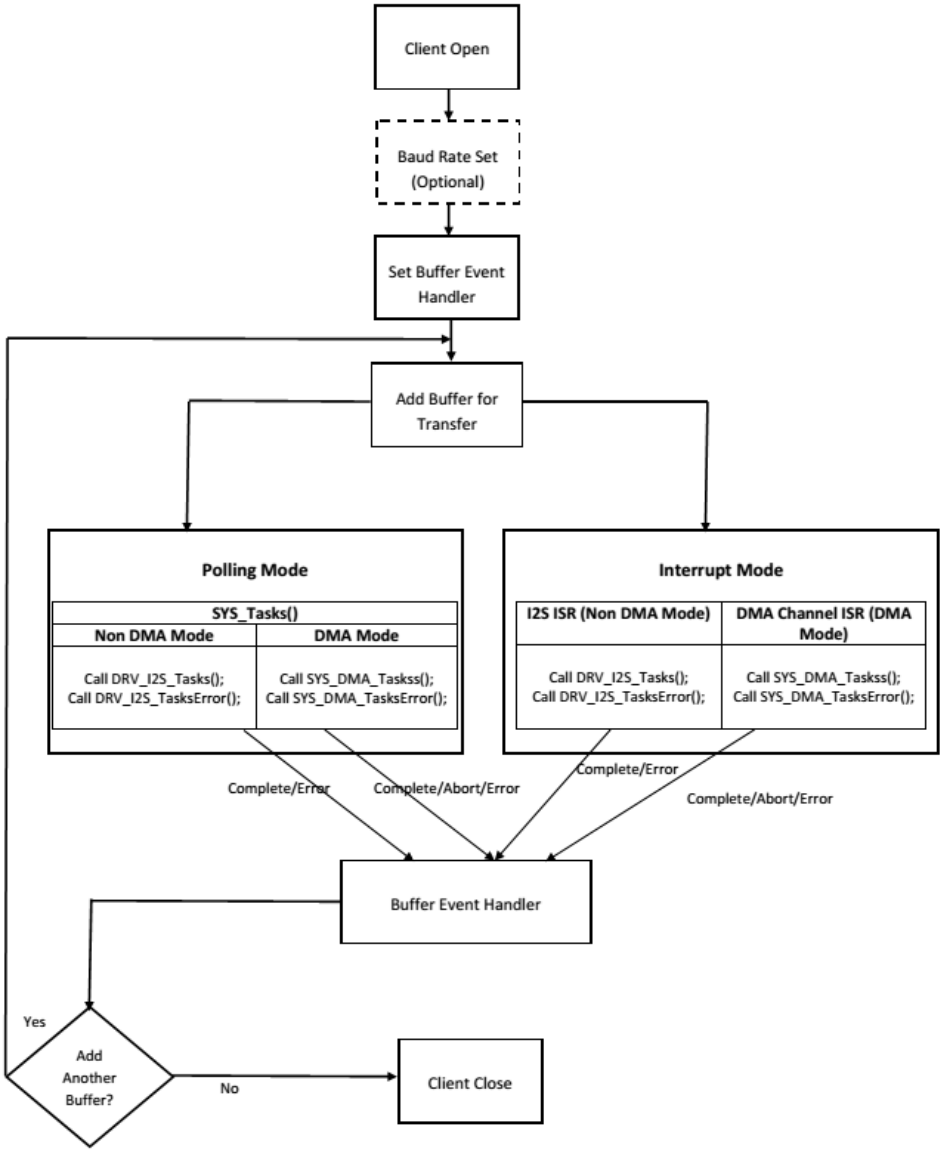
Description

Client Operations - Buffered

Client buffered operations provide a the typical audio interface. The functions DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite, and DRV_I2S_BufferAddWriteRead are the buffered data operation functions. The buffered functions schedules non-blocking operations. The function adds the request to the hardware instance queues and returns a buffer handle. The requesting client also registers a callback event with the driver. The driver notifies the client with DRV_I2S_BUFFER_EVENT_COMPLETE, DRV_I2S_BUFFER_EVENT_ERROR or DRV_I2S_BUFFER_EVENT_ABORT events.

The buffer add requests are processed under DRV_I2S_Tasks, DRV_I2S_TasksError functions. These functions are called from the I2S channel ISR in interrupt mode or from SYS_Tasks routine in Polled mode. When a DMA channel is used for transmission/reception DRV_I2S_Tasks and DRV_I2S_TasksError will be internally called by the driver from the DMA channel event handler.

The following diagram illustrates the buffered data operations



Note: It is not necessary to close and reopen the client between multiple transfers.

An application using the buffered functionality needs to perform the following steps:

1. The system should have completed necessary setup and initializations.
2. If DMA mode is desired, the DMA should be initialized by calling SYS_DMA_Initialize.
3. The necessary ports setup and remapping must be done for I2S lines: ADCDAT, DACDAT, BCLK, LRCK and MCLK (if required).
4. The driver object should have been initialized by calling DRV_I2S_Initialize. If DMA mode is desired, related attributes in the init structure must be set.
5. Open the driver using DRV_I2S_Open with the necessary ioIntent to get a client handle.
6. The necessary BCLK, LRCK, and MCLK should be set up so as to generate the required media bit rate.
7. The necessary Baud rate value should be set up by calling DRV_I2S_BaudrateSet.
8. The Register and event handler for the client handle should be set up by calling DRV_I2S_BufferEventHandlerSet.
9. Add a buffer to initiate the data transfer by calling DRV_I2S_BufferAddWrite/DRV_I2S_BufferAddRead/DRV_I2S_BufferAddWriteRead.
10. Based on polling or interrupt mode service the data processing should be set up by calling DRV_I2S_Tasks, DRV_I2S_TasksError from system tasks or I^2S ISR. When a DMA channel is used for transmission/reception system calls SYS_DMA_Tasks(), SYS_DMA_TasksError() from the system tasks or DMA channel ISR, DRV_I2S_Tasks and DRV_I2S_TasksError will be internally called by the driver from the DMA channel event handler.
11. Repeat step 9 through step 10 to handle multiple buffer transmission and reception.

12. When the client is done it can use DRV_I2S_Close to close the client handle.

Example 1:

```
// The following is an example for a Polled mode buffered transmit

#define SYS_I2S_DRIVER_INDEX DRV_I2S_1 // I2S Uses SPI Hardware
#define BUFFER_SIZE 1000
// I2S initialization structure.
// This should be populated with necessary settings.
// attributes dmaChannelTransmit/dmaChannelReceive
// and dmaInterruptTransmitSource/dmaInterruptReceiveSource
// must be set if DMA mode of operation is desired.
DRV_I2S_INIT i2sInit;
SYS_MODULE_OBJ sysObj;      //I2S module object
DRV_HANDLE handle;          //Client handle
uint32_t i2sClock;          //BCLK frequency
uint32_t baudrate;          //baudrate
uint16_t myAudioBuffer[BUFFER_SIZE]; //Audio buffer to be transmitted
DRV_I2S_BUFFER_HANDLE bufferHandle;
APP_DATA_S state;           //Application specific state
uintptr_t contextHandle;

void SYS_Initialize ( void* data )
{
    // The system should have completed necessary setup and initializations.
    // Necessary ports setup and remapping must be done for I2S lines ADCDAT,
    // DACDAT, BCLK, LRCK and MCLK

    sysObj = DRV_I2S_Initialize(SYS_I2S_DRIVER_INDEX, (SYS_MODULE_INIT*)&i2sInit);
    if (SYS_MODULE_OBJ_INVALID == sysObj)
    {
        // Handle error
    }
}


void App_Task(void)
{
        switch(state)
        {
            case APP_STATE_INIT:
            {
                handle = DRV_I2S_Open(SYS_I2S_DRIVER_INDEX, (DRV_IO_INTENT_WRITE |
DRV_IO_INTENT_NONBLOCKING));
                if(handle != DRV_HANDLE_INVALID )
                {
                    /* Update the state */
                    state = APP_STATE_WAIT_FOR_READY;
                }
            }
            break;

            case  APP_STATE_WAIT_FOR_READY:
            {
                // Necessary clock settings must be done to generate
                // required MCLK, BCLK and LRCK
                DRV_I2S_BaudrateSet(handle, i2sClock, baudrate);

                /* Set the Event handler */
                DRV_I2S_BufferEventHandlerSet(handle,App_BufferEventHandler,
                contextHandle);

                /* Add a buffer to write*/
                DRV_I2S_BufferAddWrite(handle, &bufferHandle
                                        myAudioBuffer, BUFFER_SIZE);
                if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
                {
                    // Error handling here
```

```
                }
                state = APP_STATE_IDLE;
            }
            break;

            case APP_STATE_WAIT_FOR_DONE:
                state = APP_STATE_DONE;
            break;

            case  APP_STATE_DONE:
                // Close done
                DRV_I2S_Close(handle);
            break;

            case  APP_STATE_IDLE:
                // Do nothing
            break;

            default:
            break;
        }
}


void App_BufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    uint8_t temp;

    if(DRV_I2S_BUFFER_EVENT_COMPLETE == event)
    {
        // Can set state = APP_STATE_WAIT_FOR_DONE;
        // Take Action as needed
    }
    else if(DRV_I2S_BUFFER_EVENT_ERROR == event)
    {
        // Take Action as needed
    }
    else if(DRV_I2S_BUFFER_EVENT_ABORT == event)
    {
        // Take Action as needed
    }
    else
    {
        // Do nothing
    }
}


void SYS_Tasks ( void )
{
    DRV_I2S_Tasks((SYS_MODULE_OBJ)sysObj);
    DRV_I2S_TasksError((SYS_MODULE_OBJ)sysObj);

    /* Call the application's tasks routine */
    APP_Tasks ( );
}
```

Example 2:

```
// The following is an example for interrupt mode buffered transmit

#define SYS_I2S_DRIVER_INDEX DRV_I2S_1 // I2S Uses SPI Hardware
#define BUFFER_SIZE 1000
// I2S initialization structure.
// This should be populated with necessary settings.
// attributes dmaChannelTransmit/dmaChannelReceive
// and dmaInterruptTransmitSource/dmaInterruptReceiveSource
// must be set if DMA mode of operation is desired.
```

```
DRV_I2S_INIT i2sInit;
SYS_MODULE_OBJ sysObj; //I2S module object
DRV_HANDLE handle;  //Client handle
uint32_t i2sClock;   //BCLK frequency
uint32_t baudrate;   //baudrate
uint16_t myAudioBuffer[BUFFER_SIZE]; //Audio buffer to be transmitted
DRV_I2S_BUFFER_HANDLE bufferHandle;
APP_DATA_S state; //Application specific state
uintptr_t contextHandle;

void SYS_Initialize ( void* data )
{
    // The system should have completed necessary setup and initializations.
    // Necessary ports setup and remapping must be done for I2S lines ADCDAT,
    // DACDAT, BCLK, LRCK and MCLK

    sysObj = DRV_I2S_Initialize(SYS_I2S_DRIVER_INDEX, (SYS_MODULE_INIT*)&i2sInit);
    if (SYS_MODULE_OBJ_INVALID == sysObj)
    {
        // Handle error
    }
}


void App_Task(void)
{
        switch(state)
        {
            case APP_STATE_INIT:
            {
                handle = DRV_I2S_Open(SYS_I2S_DRIVER_INDEX, (DRV_IO_INTENT_WRITE |
DRV_IO_INTENT_NONBLOCKING));
                if(handle != DRV_HANDLE_INVALID )
                {
                    /* Update the state */
                    state = APP_STATE_WAIT_FOR_READY;
                }
            }
            break;

            case  APP_STATE_WAIT_FOR_READY:
            {
                // Necessary clock settings must be done to generate
                // required MCLK, BCLK and LRCK
                DRV_I2S_BaudrateSet(handle, i2sClock, baudrate);

                /* Set the Event handler */
                DRV_I2S_BufferEventHandlerSet(handle,App_BufferEventHandler,
                contextHandle);

                /* Add a buffer to write*/
                DRV_I2S_BufferAddWrite(handle, &bufferHandle
                                       myAudioBuffer, BUFFER_SIZE);
                if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
                {
                    // Error handling here
                }
                state = APP_STATE_IDLE;
            }
            break;

            case APP_STATE_WAIT_FOR_DONE:

                state = APP_STATE_DONE;
            break;


            case  APP_STATE_DONE:
            {
```

```
                // Close done
                DRV_I2S_Close(handle);
            }
            break;

            case  APP_STATE_IDLE:
                // Do nothing
            break;

            default:
            break;
        }
}


void App_BufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    uint8_t temp;

    if(DRV_I2S_BUFFER_EVENT_COMPLETE == event)
    {
        // Can set state = APP_STATE_WAIT_FOR_DONE;
        // Take Action as needed
    }
    else if(DRV_I2S_BUFFER_EVENT_ERROR == event)
    {
        // Take Action as needed
    }
    else if(DRV_I2S_BUFFER_EVENT_ABORT == event)
    {
        // Take Action as needed
    }
    else
    {
        // Do nothing
    }
}


void SYS_Tasks ( void )
{
    /* Call the application's tasks routine */
    APP_Tasks ( );
}


void __ISR ( _SPI1_VECTOR ) _InterruptHandler_I2S1 ( void )
{
    // Call the "tasks" functions for I2S module
    DRV_I2S_Tasks((SYS_MODULE_OBJ)sysObj);
    DRV_I2S_TasksError((SYS_MODULE_OBJ)sysObj);
}


// If DMA Channel 1 was setup during initialization instead of the previous I2S ISR, the following should
be implemented
void __ISR ( _DMA1_VECTOR ) _InterruptHandler_DMA_CHANNEL_1 ( void )
{
    // Call the DMA system tasks which internally will call the I2S Tasks.
    SYS_DMA_Tasks((SYS_MODULE_OBJ)sysObj);
    SYS_DMA_TasksError((SYS_MODULE_OBJ)sysObj);
}
```

Client Operations - Non-buffered

This section provides information on non-buffered client operations.

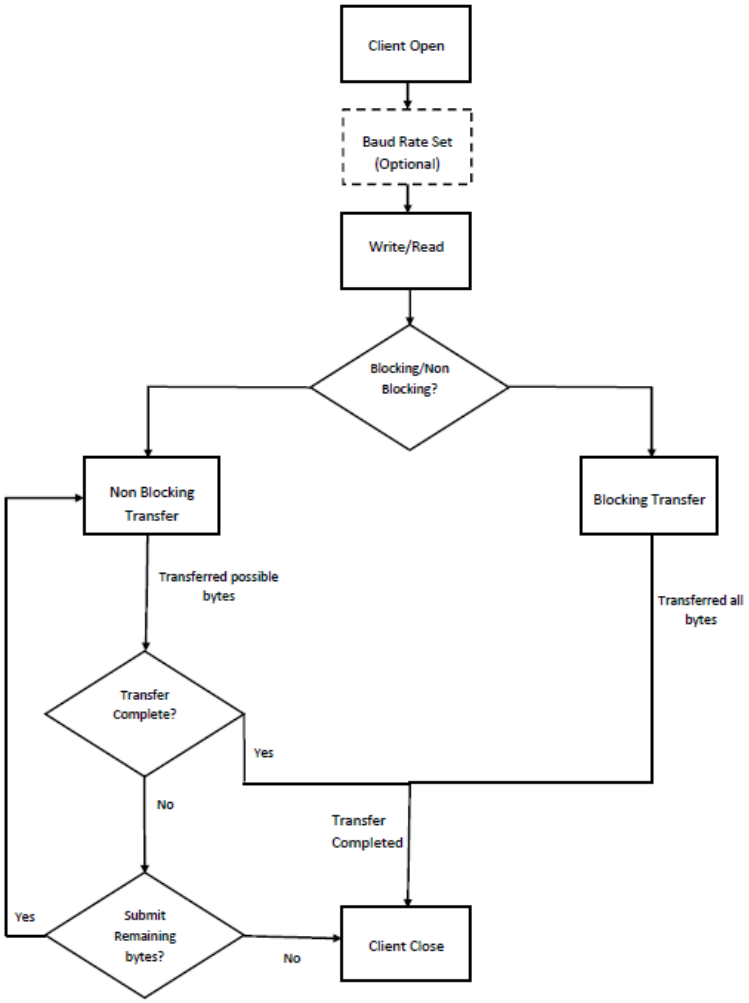
Description

Client Operations - Non-buffered

Client non-buffered operations provide a basic interface for the driver operation. This interface could be used by applications which have do not have buffered data transfer requirements. The functions DRV_I2S_Read and DRV_I2S_Write are the non-buffered data operation functions. The non-buffered functions are blocking/non-blocking depending upon the mode (ioIntent) the client was opened. If the client was opened for blocking mode these functions will only return when (or will block until) the specified data operation is completed or if an error occurred. If the client was opened for non-blocking mode, these functions will return with the number of bytes that were actually accepted for operation. The function will not wait until the data operation has completed.

Note: Non-buffered functions do not support interrupt/DMA mode.

The following diagram illustrates the non-buffered data operations



Note: It is not necessary to close and reopen the client between multiple transfers.

An application using the non-buffered functionality needs to perform the following steps:

1. The system should have completed necessary setup and initializations.
2. The necessary ports setup and remapping must be done for I2S lines: ADCDAT, DACDAT, BCLK, LRCK and MCLK (if required).
3. The driver object should have been initialized by calling DRV_I2S_Initialize.
4. Open the driver using DRV_I2S_Open with the necessary ioIntent to get a client handle.
5. The necessary BCLK, LRCK, and MCLK should be set up so as to generate the required media bit rate.
6. The necessary Baud rate value should be set up by calling DRV_I2S_BaudrateSet.
7. The Transmit/Receive data should be set up by calling DRV_I2S_Write/DRV_I2S_Read.
8. Repeat step 5 through step 7 to handle multiple buffer transmission and reception.

9. When the client is done it can use DRV_I2S_Close to close the client handle.

Example 1:

```
// The following is an example for a blocking transmit
    #define SYS_I2S_DRIVER_INDEX DRV_I2S_1 // I2S Uses SPI Hardware
    #define BUFFER_SIZE 1000
    DRV_I2S_INIT i2sInit; //I2S initialization structure
                          //This should be populated with necessary settings
    SYS_MODULE_OBJ sysObj; //I2S module object
    APP_DATA_S state; //Application specific state
    DRV_HANDLE handle;  //Client handle
    uint32_t i2sClock;   //BCLK frequency
    uint32_t baudrate;   //baudrate
    uint16_t myAudioBuffer[BUFFER_SIZE]; //Audio buffer to be transmitted
    uint32_t count;

    // The system should have completed necessary setup and initializations.
    // Necessary ports setup and remapping must be done for
    // I2S lines ADCDAT, DACDAT, BCLK, LRCK and MCLK

    sysObj = DRV_I2S_Initialize(SYS_I2S_DRIVER_INDEX, (SYS_MODULE_INIT*)&i2sInit);
    if (SYS_MODULE_OBJ_INVALID == sysObj)
    {
        // Handle error
    }
    while(1)
    {
        switch(state)
        {
            case APP_STATE_INIT:
            {
                handle = DRV_I2S_Open(SYS_I2S_DRIVER_INDEX, (DRV_IO_INTENT_WRITE | DRV_IO_INTENT_BLOCKING));
                if(handle != DRV_HANDLE_INVALID )
                {
                    /* Update the state */
                    state = APP_STATE_WAIT_FOR_READY;
                }
            }
            break;
            case  APP_STATE_WAIT_FOR_READY:
            {
                // Necessary clock settings must be done to generate
                // required MCLK, BCLK and LRCK
                DRV_I2S_BaudrateSet(handle, i2sClock, baudrate);
                // Blocks here and transfer the buffer
                count = DRV_I2S_Write(handle, &myAudioBuffer,BUFFER_SIZE);
                if(count == DRV_I2S_WRITE_ERROR)
                {
                    //Handle Error
                } else
                {
                    // Transfer Done
                    state = APP_STATE_DONE;
                }
            }
            break;
            case  APP_STATE_DONE:
            {
                // Close done
                DRV_I2S_Close(handle);
            }
            break;
            default:
            break;
        }
    }
```

Example 2:

```
// Following is an example for a non blocking transmit
    #define SYS_I2S_DRIVER_INDEX DRV_I2S_1 //I2S Uses SPI Hardware
    #define BUFFER_SIZE 1000
    DRV_I2S_INIT i2sInit; //I2S initialization structure.
                          // This should be populated with necessary settings
    SYS_MODULE_OBJ sysObj; //I2S module object
    APP_DATA_S state; //Application specific state
    DRV_HANDLE handle;  //Client handle
    uint32_t i2sClock;   //BCLK frequency
    uint32_t baudrate;   //baudrate
    uint16_t myAudioBuffer[BUFFER_SIZE]; //Audio buffer to be transmitted
    uint32_t count,total,size;

    total = 0;
    size = BUFFER_SIZE;

    // The system should have completed necessary setup and initializations.
    // Necessary ports setup and remapping must be done for I2S lines ADCDAT,
    // DACDAT, BCLK, LRCK and MCLK

    sysObj = DRV_I2S_Initialize(SYS_I2S_DRIVER_INDEX, (SYS_MODULE_INIT*)&i2sInit);
    if (SYS_MODULE_OBJ_INVALID == sysObj)
    {
        // Handle error
    }

    while(1)
    {
        switch(state)
        {
            case APP_STATE_INIT:
            {
                handle = DRV_I2S_Open(SYS_I2S_DRIVER_INDEX, (DRV_IO_INTENT_WRITE |
DRV_IO_INTENT_NONBLOCKING));
                if(handle != DRV_HANDLE_INVALID )
                {
                    /* Update the state */
                    state = APP_STATE_WAIT_FOR_READY;
                }
            }
            break;
            case  APP_STATE_WAIT_FOR_READY:
            {
                // Necessary clock settings must be done to generate
                // required MCLK, BCLK and LRCK
                DRV_I2S_BaudrateSet(handle, i2sClock, baudrate);
                // Transfer whatever possible number of bytes
                count = DRV_I2S_Write(handle, &myAudioBuffer,size);
                if(count == DRV_I2S_WRITE_ERROR)
                {
                    //Handle Error
                } else
                {
                    // 'count' bytes transferred
                    state = APP_STATE_WAIT_FOR_DONE;
                }
            }
            break;
            case APP_STATE_WAIT_FOR_DONE:
                // Can perform other Application tasks here
                // ........................
                // ........................
                // ........................
                size = size - count;
                if(size!=0)
                {
```

```
                // Change the state so as to submit
                // another possible transmission
                state = APP_STATE_WAIT_FOR_READY;
            }
            else
            {
                // We are done
                state = APP_STATE_DONE;
            }
        break;
        case  APP_STATE_DONE:
        {
            if(DRV_I2S_CLOSE_FAILURE == DRV_I2S_Close(handle))
            {
                // Handle error
            }
            else
            {
                // Close done
            }
        }
        break;
        default:
        break;
    }
}
```

Configuring the Library

Client Configuration

	Name	Description
	DRV_I2S_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_I2S_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.

System Configuration

	Name	Description
	DRV_I2S_INDEX	I2S Static Index selection
	DRV_I2S_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_I2S_INTERRUPT_MODE	Macro controls interrupt based operation of the driver
	DRV_I2S_INTERRUPT_SOURCE_ERROR	Defines the interrupt source for the error interrupt
	DRV_I2S_INTERRUPT_SOURCE_RECEIVE	Macro to define the Receive interrupt source in case of static driver
	DRV_I2S_INTERRUPT_SOURCE_TRANSMIT	Macro to define the Transmit interrupt source in case of static driver
	DRV_I2S_PERIPHERAL_ID	Configures the I2S PLIB Module ID
	DRV_I2S_RECEIVE_DMA_CHANNEL	Macro to defines the I2S Driver Receive DMA Channel in case of static driver
	DRV_I2S_STOP_IN_IDLE	Identifies whether the driver should stop operations in stop in Idle mode.
	DRV_I2S_TRANSMIT_DMA_CHANNEL	Macro to defines the I2S Driver Transmit DMA Channel in case of static driver
	DRV_I2S_RECEIVE_DMA_CHAINING_CHANNEL	Macro to defines the I2S Driver Receive DMA Chaining Channel in case of static driver

Description

The configuration of the I2S Driver Library is based on the file `sys_config.h`.

This header file contains the configuration selection for the I2S Driver Library. Based on the selections made, the I2S Driver Library may support the selected features. These configuration settings will apply to all instances of the I2S Driver Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

System Configuration

DRV_I2S_INDEX Macro

I2S Static Index selection

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_INDEX
```

Description

Index - Used for static drivers

I2S Static Index selection for the driver object reference. This macro defines the driver index in case of static and static multi-client build. For example, if this macro is set to DRV_I2S_INDEX_2, then static driver APIs would be DRV_I2S2_Initialize(), DRV_I2S2_Open() etc. When building static drivers, this macro should be different for each static build of the I2S driver that needs to be included in the project.

Remarks

This index is required to make a reference to the driver object

DRV_I2S_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_INSTANCES_NUMBER
```

Description

I2S driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of I2S modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None

DRV_I2S_INTERRUPT_MODE Macro

Macro controls interrupt based operation of the driver

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_INTERRUPT_MODE
```

Description

I2S Interrupt Mode Operation Control

This macro controls the interrupt based operation of the driver. The possible values it can take are

- true - Enables the interrupt mode
- false - Enables the polling mode

If the macro value is true, then Interrupt Service Routine for the interrupt should be defined in the application. The DRV_I2S_Tasks() routine should be called in the ISR.

DRV_I2S_INTERRUPT_SOURCE_ERROR Macro

Defines the interrupt source for the error interrupt

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_INTERRUPT_SOURCE_ERROR
```

Description

Error Interrupt Source

Macro to define the Error interrupt source in case of static driver. The interrupt source defined by this macro will override the errorInterruptSource member of the DRV_I2S_INIT initialization data structure in the driver initialization routine. This value should be set to the I2S module error interrupt enumeration in the Interrupt PLIB for the microcontroller.

DRV_I2S_INTERRUPT_SOURCE_RECEIVE Macro

Macro to define the Receive interrupt source in case of static driver

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_INTERRUPT_SOURCE_RECEIVE
```

Description

Receive Interrupt Source

Macro to define the Receive interrupt source in case of static driver. The interrupt source defined by this macro will override the rxInterruptSource member of the DRV_I2S_INIT initialization data structure in the driver initialization routine. This value should be set to the I2S module receive interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None.

DRV_I2S_INTERRUPT_SOURCE_TRANSMIT Macro

Macro to define the Transmit interrupt source in case of static driver

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_INTERRUPT_SOURCE_TRANSMIT
```

Description

Transmit Interrupt Source

Macro to define the TX interrupt source in case of static driver. The interrupt source defined by this macro will override the txInterruptSource member of the DRV_I2S_INIT initialization data structure in the driver initialization routine. This value should be set to the I2S module transmit interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None.

DRV_I2S_PERIPHERAL_ID Macro

Configures the I2S PLIB Module ID

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_PERIPHERAL_ID
```

Description

I2S Peripheral Library Module ID

This macro configures the PLIB ID if the driver is built statically. This value will override the I2SID member of the DRV_I2S_INIT initialization data structure. In that when the driver is built statically, the I2SID member of the DRV_I2S_INIT data structure will be ignored by the driver initialization routine and this macro will be considered. This should be set to the PLIB ID of I2S module (I2S_ID_1, I2S_ID_2 and so on).

DRV_I2S_RECEIVE_DMA_CHANNEL Macro

Macro to defines the I2S Driver Receive DMA Channel in case of static driver

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_RECEIVE_DMA_CHANNEL
```

Description

I2S Driver Receive DMA Channel

Macro to define the I2S Receive DMA Channel in case of static driver. The DMA channel defined by this macro will override the dmaChannelReceive member of the DRV_I2S_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel in the DMA PLIB for the microcontroller.

Remarks

None.

DRV_I2S_STOP_IN_IDLE Macro

Identifies whether the driver should stop operations in stop in Idle mode.

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_STOP_IN_IDLE
```

Description

I2S driver objects configuration

Identifies whether the driver should stop operations in stop in Idle mode. true - Indicates stop in idle mode. false - Indicates do not stop in Idle mode.

Remarks

None

DRV_I2S_TRANSMIT_DMA_CHANNEL Macro

Macro to defines the I2S Driver Transmit DMA Channel in case of static driver

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_TRANSMIT_DMA_CHANNEL
```

Description

I2S Driver Transmit DMA Channel

Macro to define the I2S Transmit DMA Channel in case of static driver. The DMA channel defined by this macro will override the dmaChannelTransmit member of the DRV_I2S_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel in the DMA PLIB for the microcontroller.

Remarks

None.

DRV_I2S_RECEIVE_DMA_CHAINING_CHANNEL Macro

Macro to defines the I2S Driver Receive DMA Chaining Channel in case of static driver

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_RECEIVE_DMA_CHAINING_CHANNEL
```

Description

I2S Driver Receive DMA Chaining Channel

Macro to define the I2S Receive DMA Chaining Channel in case of static driver. The DMA channel defined by this macro will override the dmaChaningChannelReceive member of the DRV_I2S_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel in the DMA PLIB for the microcontroller.

Remarks

None.

Client Configuration

DRV_I2S_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_CLIENTS_NUMBER
```

Description

I2S Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. This value represents the total number of clients to be supported across all hardware instances. So if I2S1 will be accessed by 2 clients and I2S2 will accessed by 3 clients, then this number should be 5. It is recommended that this be set exactly equal to the number of expected clients. Client support consumes RAM memory space. If this macro is not defined and the DRV_I2S_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - single client operation. If this macro is defined and the DRV_I2S_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - multi client operation.

Remarks

None

DRV_I2S_QUEUE_DEPTH_COMBINED Macro

Number of entries of all queues in all instances of the driver.

File

drv_i2s_config_template.h

C

```
#define DRV_I2S_QUEUE_DEPTH_COMBINED
```

Description

I2S Driver Buffer Queue Entries

This macro defined the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for transmit and receive operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build). The hardware instance transmit buffer queue will queue transmit buffers submitted by the DRV_I2S_BufferAddWrite() function. The hardware instance receive buffer queue will queue receive buffers submitted by the DRV_I2S_BufferAddRead() function.

A buffer queue will contains buffer queue entries, each related to a BufferAdd request. This configuration macro defines total number of buffer

entries that will be available for use between all I2S driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking read and write requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its transmit and receive buffer queue size.

As an example, consider the case of static single client driver application where full duplex non blocking operation is desired without queuing, the minimum transmit queue depth and minimum receive queue depth should be 1. Hence the total number of buffer entries should be 2.

As an example, consider the case of a dynamic driver (say 2 instances) where instance 1 will queue up to 3 write requests and up to 2 read requests, and instance 2 will queue up to 2 write requests and up to 6 read requests, the value of this macro should be 13 (2 + 3 + 2 + 6).

Remarks

The maximum combined queue depth should not be greater than 0xFFFF(ie 65535)

Building the Library

This section lists the files that are available in the I2S Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/i2s`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_i2s.h	This file provides the interface definitions of the I2S driver.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_i2s_dma.c	This file contains the core implementation of the I2S driver with DMA support.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
/src/dynamic/drv_i2s_dma_advanced.c	This file contains the implementation of the I2S driver with DMA support using the channel chaining feature.
/src/dynamic/drv_i2s.c	This file contains the implementation of the I2S driver without DMA support.
/src/dynamic/drv_i2s_read_write.c	This file contains the basic read/write implementation of the I2S driver.

Module Dependencies

The I2S Driver Library depends on the following modules:

- SPI Peripheral Library
- DMA Peripheral Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_I2S_Deinitialize	Deinitializes the specified instance of the I2S driver module. **Implementation:** Dynamic

	DRV_I2S_Initialize	Initializes hardware and data for the instance of the I2S module. **Implementation:** Dynamic
	DRV_I2S_Status	Gets the current status of the I2S driver module. **Implementation:** Dynamic
	DRV_I2S_Tasks	Maintains the driver's receive state machine and implements its ISR. **Implementation:** Dynamic
	DRV_I2S_TasksError	Maintains the driver's error state machine and implements its ISR. **Implementation:** Dynamic

b) Client Setup Functions

	Name	Description
	DRV_I2S_Close	Closes an opened-instance of the I2S driver. **Implementation:** Dynamic
	DRV_I2S_Open	Opens the specified I2S driver instance and returns a handle to it. **Implementation:** Dynamic

c) Data Transfer Functions

	Name	Description
	DRV_I2S_BufferAddRead	Schedule a non-blocking driver read operation. **Implementation:** Dynamic
	DRV_I2S_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_I2S_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_I2S_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_I2S_BufferCombinedQueueSizeGet	This function returns the number of bytes queued (to be processed) in the buffer queue. **Implementation:** Dynamic
	DRV_I2S_BufferQueueFlush	This function flushes off the buffers associated with the client object. **Implementation:** Dynamic
	DRV_I2S_Read	Reads data from the I2S. **Implementation:** Dynamic
	DRV_I2S_Write	Writes data to the I2S. **Implementation:** Dynamic
	DRV_I2S_BufferProcessedSizeGet	This function returns number of bytes that have been processed for the specified buffer. **Implementation:** Dynamic

d) Miscellaneous Functions

	Name	Description
	DRV_I2S_BaudSet	This function sets the baud. **Implementation:** Dynamic
	DRV_I2S_ErrorGet	This function returns the error(if any) associated with the last client request. **Implementation:** Dynamic
	DRV_I2S_ReceiveErrorIgnore	This function enable/disable ignoring of the receive overflow error. **Implementation:** Dynamic
	DRV_I2S_TransmitErrorIgnore	This function enable/disable ignoring of the transmit underrun error. **Implementation:** Dynamic

e) Data Types and Constants

	Name	Description
	DRV_I2S_AUDIO_PROTOCOL_MODE	Identifies the Audio Protocol Mode of the I2S module.
	DRV_I2S_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_I2S_BUFFER_EVENT_HANDLER	Pointer to a I2S Driver Buffer Event handler function
	DRV_I2S_BUFFER_HANDLE	Handle identifying a read or write buffer passed to the driver.
	DRV_I2S_CLOCK_MODE	Identifies the various clock modes of the I2S module.
	DRV_I2S_DATA16	Defines the left and right channel data for 16-bit audio data
	DRV_I2S_DATA24	Defines the left and right channel data for 24-bit audio data

	DRV_I2S_DATA32	Defines the left and right channel data for 32-bit audio data
	DRV_I2S_ERROR	Defines the possible errors that can occur during driver operation.
	DRV_I2S_MODE	Identifies the usage modes of the I2S module.
	DRV_I2S_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_I2S_COUNT	Number of valid I2S driver indices
	DRV_I2S_READ_ERROR	I2S Driver Read Error.
	DRV_I2S_WRITE_ERROR	I2S Driver Write Error.
	DRV_I2S_INDEX_0	I2S driver index definitions
	DRV_I2S_INDEX_1	This is macro DRV_I2S_INDEX_1.
	DRV_I2S_INDEX_2	This is macro DRV_I2S_INDEX_2.
	DRV_I2S_INDEX_3	This is macro DRV_I2S_INDEX_3.
	DRV_I2S_INDEX_4	This is macro DRV_I2S_INDEX_4.
	DRV_I2S_INDEX_5	This is macro DRV_I2S_INDEX_5.
	DRV_I2S_INTERFACE	This structure defines a structure of I2S Driver function pointers.

Description

This section describes the Application Programming Interface (API) functions of the I2S Driver Library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_I2S_Deinitialize Function

Deinitializes the specified instance of the I2S driver module.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the I2S driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_I2S_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_I2S_Initialize
SYS_STATUS          status;


DRV_I2S_Deinitialize(object);

status = DRV_I2S_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_I2S_Initialize routine

Function

void DRV_I2S_Deinitialize(SYS_MODULE_OBJ object)

DRV_I2S_Initialize Function

Initializes hardware and data for the instance of the I2S module.

Implementation: Dynamic

File

drv_i2s.h

C

```
SYS_MODULE_OBJ DRV_I2S_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the I2S driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the I2S module ID. For example, driver instance 0 can be assigned to I2S2. If the driver is built statically, then some of the initialization parameters are overridden by configuration macros. Refer to the description of the DRV_I2S_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This routine must be called before any other I2S routine is called.

This routine should only be called once during system initialization unless DRV_I2S_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

To Enable the DMA mode of operation the init parameters 'dmaChannelTransmit' /'dmaChannelReceive' must be set to valid DMA channel. When DMA mode of operation is enabled, the normal mode(Usual TX and RX) operation is inhibited. When 'dmaChannelTransmit'/'dmaChannelReceive' are set to valid channel numbers the related DMA interrupt source parameters 'dmaInterruptTransmitSource'/ 'dmaInterruptReceiveSource' must be set with appropriate DMA channel interrupt source.

Preconditions

If DMA mode of operation is intended, SYS_DMA_Initialize should have been called before calling this function.

Example

```
DRV_I2S_INIT               init;
SYS_MODULE_OBJ             objectHandle;

init.moduleInit.value          = SYS_MODULE_POWER_RUN_FULL;
init.spiID                     = SPI_ID_1;
init.usageMode                 = DRV_I2S_MODE_MASTER;
init.baudClock                 = SPI_BAUD_RATE_MCLK_CLOCK;
init.baud                      = 48000;
init.clockMode                 = DRV_I2S_CLOCK_MODE_IDLE_HIGH_EDGE_FALL;
init.audioCommWidth            = SPI_AUDIO_COMMUNICATION_24DATA_32FIFO_32CHANNEL;
init.audioTransmitMode         = SPI_AUDIO_TRANSMIT_STEREO;
init.inputSamplePhase          = SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE;
init.protocolMode              = DRV_I2S_AUDIO_I2S;
init.txInterruptSource         = INT_SOURCE_SPI_1_TRANSMIT;
init.rxInterruptSource         = INT_SOURCE_SPI_1_RECEIVE;
init.errorInterruptSource      = INT_SOURCE_SPI_1_ERROR;
init.queueSizeTransmit         = 3;
init.queueSizeReceive          = 2;
init.dmaChannelTransmit        = DMA_CHANNEL_NONE;
init.dmaChannelReceive         = DMA_CHANNEL_NONE;

objectHandle = DRV_I2S_Initialize(DRV_I2S_INDEX_1, (SYS_MODULE_INIT*)init);
```

```
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the driver instance to be initialized
init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

SYS_MODULE_OBJ DRV_I2S_Initialize

(

const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT *const init

);

DRV_I2S_Status Function

Gets the current status of the I2S driver module.

Implementation: Dynamic

File

drv_i2s.h

C

```
SYS_STATUS DRV_I2S_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed

SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed

SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This routine provides the current status of the I2S driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_I2S_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_I2S_Initialize
SYS_STATUS          i2sStatus;

i2sStatus = DRV_I2S_Status(object);
if (SYS_STATUS_READY == i2sStatus)
{
    // This means the driver can be opened using the
    // DRV_I2S_Open() function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_I2S_Initialize routine

Function

SYS_STATUS DRV_I2S_Status(SYS_MODULE_OBJ object)

DRV_I2S_Tasks Function

Maintains the driver's receive state machine and implements its ISR.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal receive state machine and implement its transmit and receive ISR for interrupt-driven implementations. In polling mode, this function should be called from the SYS_Tasks function. In interrupt mode, this function should be called from the interrupt service routine of the I2S that is associated with this I2S driver hardware instance.

In DMA mode of operation, this function should be called from the interrupt service routine of the channel associated with the transmission/reception of the I2s driver hardware instance.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_I2S_Initialize

while (true)
{
    DRV_I2S_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_I2S_Initialize)

Function

void DRV_I2S_Tasks(SYS_MODULE_OBJ object)

DRV_I2S_TasksError Function

Maintains the driver's error state machine and implements its ISR.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_TasksError(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal error state machine and implement its error ISR for interrupt-driven implementations. In polling

mode, this function should be called from the SYS_Tasks() function. In interrupt mode, this function should be called in the error interrupt service routine of the I2S that is associated with this I2S driver hardware instance.

In DMA mode of operation, this function should be called from the interrupt service routine of the channel associated with the transmission/reception of the I2s driver hardware instance.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_I2S_Initialize

while (true)
{
    DRV_I2S_TasksError (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_I2S_Initialize)

Function

void DRV_I2S_TasksError (SYS_MODULE_OBJ object)

b) Client Setup Functions

DRV_I2S_Close Function

Closes an opened-instance of the I2S driver.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_Close(const DRV_HANDLE handle);
```

Returns

- None

Description

This routine closes an opened-instance of the I2S driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_I2S_Open before the caller may use the driver again

Remarks

Usually there is no need for the driver client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_I2S_Open
```

```
DRV_I2S_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_I2S_Close(DRV_Handle handle)

DRV_I2S_Open Function

Opens the specified I2S driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_i2s.h

C

```
DRV_HANDLE DRV_I2S_Open(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_I2S_CLIENTS_NUMBER is

insufficient.

- if the client is trying to open the driver but driver has been opened

exclusively by another client.

- if the driver hardware instance being opened is not initialized or is

invalid.

Description

This routine opens the specified I2S driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options additionally affect the behavior of the DRV_I2S_Read() and DRV_I2S_Write() functions. If the ioIntent is DRV_IO_INTENT_NONBLOCKING, then these function will not block even if the required amount of data could not be processed. If the ioIntent is DRV_IO_INTENT_BLOCKING, these functions will block until the required amount of data is processed.

If ioIntent is DRV_IO_INTENT_READ, the client will only be read from the driver. If ioIntent is DRV_IO_INTENT_WRITE, the client will only be able to write to the driver. If the ioIntent in DRV_IO_INTENT_READWRITE, the client will be able to do both, read and write.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_I2S_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

Function DRV_I2S_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_I2S_Open(DRV_I2S_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

```
  DRV_HANDLE DRV_I2S_Open
(
const SYS_MODULE_INDEX drvIndex,
const    DRV_IO_INTENT    ioIntent
)
```

c) Data Transfer Functions

DRV_I2S_BufferAddRead Function

Schedule a non-blocking driver read operation.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_BufferAddRead(const DRV_HANDLE handle, DRV_I2S_BUFFER_HANDLE * bufferHandle, void * buffer,
size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_I2S_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_I2S_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for write-only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_I2S_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_I2S_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the I2S Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another I2S driver instance. It should not otherwise be called directly in an ISR.

This function supports DMA mode of operation.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S device instance and the DRV_I2S_Status must have returned SYS_STATUS_READY.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_I2S_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;
```

```
// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver

DRV_I2S_BufferEventHandlerSet(myI2SHandle,
                APP_I2SBufferEventHandler, (uintptr_t)&myAppObj);

DRV_I2S_BufferAddRead(myI2Shandle, &bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the I2S instance as returned by the DRV_I2S_Open function
buffer	Buffer where the received data will be stored.
size	Buffer size in bytes
bufferHandle	Pointer to an argument that will contain the return buffer handle

Function

void DRV_I2S_BufferAddRead

(

const DRV_HANDLE handle,

DRV_I2S_BUFFER_HANDLE *bufferHandle,

void * buffer, size_t size

)

DRV_I2S_BufferAddWrite Function

Schedule a non-blocking driver write operation.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_BufferAddWrite(const DRV_HANDLE handle, DRV_I2S_BUFFER_HANDLE * bufferHandle, void * buffer,
size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_I2S_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_I2S_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read-only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_I2S_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_I2S_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the I2S Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another I2S driver instance. It should not otherwise be called directly in an ISR.

This function supports DMA mode of operation.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S device instance and the DRV_I2S_Status must have returned SYS_STATUS_READY.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_I2S_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver

DRV_I2S_BufferEventHandlerSet(myI2SHandle,
                APP_I2SBufferEventHandler, (uintptr_t)&myAppObj);

DRV_I2S_BufferAddWrite(myI2Shandle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:
```

```
            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the I2S instance as return by the DRV_I2S_Open function
buffer	Data to be transmitted
size	Buffer size in bytes
bufferHandle	Pointer to an argument that will contain the return buffer handle

Function

void DRV_I2S_BufferAddWrite

(

const DRV_HANDLE handle,

DRV_I2S_BUFFER_HANDLE *bufferHandle,

void * buffer, size_t size

);

DRV_I2S_BufferAddWriteRead Function

Schedule a non-blocking driver write-read operation.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_BufferAddWriteRead(const DRV_HANDLE handle, DRV_I2S_BUFFER_HANDLE * bufferHandle, void *
transmitBuffer, void * receiveBuffer, size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_I2S_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write-read operation. The function returns with a valid buffer handle in the bufferHandle argument if the write-read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_I2S_BUFFER_HANDLE_INVALID:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only or write only
- if the buffer size is 0
- if the queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_I2S_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_I2S_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the I2S Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another I2S driver instance. It should not otherwise be called directly in an ISR.

This function is useful when there is valid read expected for every I2S write. The transmit and receive size must be same.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S device instance and the DRV_I2S_Status must have returned SYS_STATUS_READY.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READWRITE must have been specified in the DRV_I2S_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybufferTx[MY_BUFFER_SIZE];
uint8_t mybufferRx[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver

DRV_I2S_BufferEventHandlerSet(myI2SHandle,
                APP_I2SBufferEventHandler, (uintptr_t)&myAppObj);

DRV_I2S_BufferAddWriteRead(myI2Shandle, &bufferHandle,
                                    mybufferTx,mybufferRx,MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the I2S instance as returned by the DRV_I2S_Open function
bufferHandle	Pointer to an argument that will contain the return buffer handle
transmitBuffer	Buffer where the transmit data will be stored
receiveBuffer	Buffer where the received data will be stored
size	Buffer size in bytes

Function

void DRV_I2S_BufferAddWriteRead

(

```
const       DRV_HANDLE handle,
    DRV_I2S_BUFFER_HANDLE   *bufferHandle,
void *transmitBuffer, void *receiveBuffer,
size_t size
)
```

DRV_I2S_BufferEventHandlerSet Function

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_BufferEventHandlerSet(DRV_HANDLE handle, const DRV_I2S_BUFFER_EVENT_HANDLER eventHandler,
const uintptr_t contextHandle);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls either the DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite or DRV_I2S_BufferAddWriteRead function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet(myI2SHandle, APP_I2SBufferEventHandler,
                                (uintptr_t)&myAppObj);

DRV_I2S_BufferAddRead(myI2Shandle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE handle, uintptr_t contextHandle)
```

```
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_I2S_BufferEventHandlerSet

(

const DRV_HANDLE handle,

DRV_I2S_BUFFER_EVENT_HANDLER eventHandler,

uintptr_t contextHandle

)

DRV_I2S_BufferCombinedQueueSizeGet Function

This function returns the number of bytes queued (to be processed) in the buffer queue.

Implementation: Dynamic

File

drv_i2s.h

C

```
size_t DRV_I2S_BufferCombinedQueueSizeGet(DRV_HANDLE handle);
```

Returns

Returns the number of the bytes that have been processed for this buffer. Returns 0 for an invalid or an expired client handle.

Description

This function returns the number of bytes queued (to be processed) in the buffer queue of the driver instance associated with the calling client. The client can use this function to know number of remaining bytes (from the buffers submitted by it)is in the queue to be transmitted.

Remarks

None.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

One of DRV_I2S_BufferAddRead/DRV_I2S_BufferAddWrite function must have been called and buffers should have been queued for transmission.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
size_t bufferQueuedSize;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet(myI2SHandle, APP_I2SBufferEventHandle,
                                                    (uintptr_t)&myAppObj);

DRV_I2S_BufferAddRead(myI2Shandle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// The data is being processed after adding the buffer to the queue.
// The user can get to know dynamically available data in the queue to be
// transmitted by calling DRV_I2S_BufferCombinedQueueSizeGet
bufferQueuedSize = DRV_I2S_BufferCombinedQueueSizeGet(myI2SHandle);
```

Parameters

Parameters	Description
handle	Opened client handle associated with a driver object.

Function

size_t DRV_I2S_BufferCombinedQueueSizeGet(DRV_HANDLE handle)

DRV_I2S_BufferQueueFlush Function

This function flushes off the buffers associated with the client object.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_BufferQueueFlush(DRV_HANDLE handle);
```

Returns

None.

Description

This function flushes off the buffers associated with the client object and disables the DMA channel used for transmission.

Remarks

None.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

One of DRV_I2S_BufferAddRead/DRV_I2S_BufferAddWrite function must have been called and buffers should have been queued for transmission.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;
size_t bufferQueuedSize;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet(myI2SHandle, APP_I2SBufferEventHandle,
                                                    (uintptr_t)&myAppObj);

DRV_I2S_BufferAddRead(myI2Shandle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// The data is being processed after adding the buffer to the queue.
// The user can stop the data processing and flushoff the data
// in the queue by calling DRV_I2S_BufferQueueFlush
DRV_I2S_BufferQueueFlush(myI2SHandle);
```

Parameters

Parameters	Description
handle	Opened client handle associated with a driver object.

Function

void DRV_I2S_BufferQueueFlush(DRV_HANDLE handle)

DRV_I2S_Read Function

Reads data from the I2S.

Implementation: Dynamic

File

drv_i2s.h

C

```
size_t DRV_I2S_Read(const DRV_HANDLE handle, uint8_t * buffer, const size_t numBytes);
```

Returns

Number of bytes actually copied into the caller's buffer. Returns DRV_I2S_READ_ERROR in case of an error.

Description

This routine reads data from the I2S. This function is blocking if the driver was opened by the client for blocking operation. This function will not block if the driver was opened by the client for non blocking operation. If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_BLOCKING, this function will only return when (or will block until) numBytes of bytes have been received or if an error occurred.

If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_NON_BLOCKING, this function will return with the number of bytes that were actually read. The function will not wait until numBytes of bytes have been read.

Remarks

This function is thread safe in a RTOS application. It is recommended that this function not be called in I2S Driver Event Handler due to the potential blocking nature of the function. This function should not be called directly in an ISR. It should not be called in the event handler associated with another I2S driver instance.

This function does not supports DMA mode of operation.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_I2S_Open call.

Example

```
DRV_HANDLE      myI2SHandle;    // Returned from DRV_I2S_Open
char            myBuffer[MY_BUFFER_SIZE];
unsigned int    count;
unsigned int    total;

total = 0;
do
{
    count  = DRV_I2S_Read(myI2SHandle, &myBuffer[total],
                          MY_BUFFER_SIZE - total);
    total += count;
    if(count == DRV_I2S_READ_ERROR)
    {
        // Handle error ...
    }
    else
    {
        // Do what needs to be..
    }
} while( total < MY_BUFFER_SIZE );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
buffer	Buffer into which the data read from the I2S instance will be placed.
numBytes	Total number of bytes that need to be read from the module instance (must be equal to or less than the size of the buffer)

Function

size_t DRV_I2S_Read

(

const DRV_HANDLE handle,

uint8_t *buffer,

const size_t numBytes

)

DRV_I2S_Write Function

Writes data to the I2S.

Implementation: Dynamic

File

drv_i2s.h

C

```
size_t DRV_I2S_Write(const DRV_HANDLE handle, uint8_t * buffer, const size_t numBytes);
```

Returns

Number of bytes actually written to the driver. Return DRV_I2S_WRITE_ERROR in case of an error.

Description

This routine writes data to the I2S. This function is blocking if the driver was opened by the client for blocking operation. This function will not block if the driver was opened by the client for non blocking operation. If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_BLOCKING, this function will only return when (or will block until) numbytes of bytes have been transmitted or if an error occurred.

If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_NON_BLOCKING, this function will return with the number of bytes that were actually accepted for transmission. The function will not wait until numBytes of bytes have been transmitted.

Remarks

This function is thread safe in a RTOS application. It is recommended that this function not be called in I2S Driver Event Handler due to the potential blocking nature of the function. This function should not be called directly in an ISR. It should not be called in the event handler associated with another USART driver instance.

This function does not supports DMA mode of operation.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_I2S_Open call.

Example

```
DRV_HANDLE myI2SHandle;    // Returned from DRV_I2S_Open
char myBuffer[MY_BUFFER_SIZE];
int count;
unsigned int total;
total = 0;
do
{
    count  = DRV_I2S_Write(myI2SHandle, &myBuffer[total],
            MY_BUFFER_SIZE - total);
    total += count;
    if(count == DRV_I2S_WRITE_ERROR)
    {
        // Handle error ...
    }
    else
    {
        // Do what needs to be ..
    }
} while( total < MY_BUFFER_SIZE );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
buffer	Buffer containing the data to written.
numbytes	size of the buffer

Function

size_t DRV_I2S_Write

(

const DRV_HANDLE handle,

void * buffer,

const size_t numbytes

)

DRV_I2S_BufferProcessedSizeGet Function

This function returns number of bytes that have been processed for the specified buffer.

Implementation: Dynamic

File

drv_i2s.h

C

```
size_t DRV_I2S_BufferProcessedSizeGet(DRV_HANDLE handle);
```

Returns

Returns the number of the bytes that have been processed for this buffer. Returns 0 for an invalid or an expired buffer handle.

Description

This function returns number of bytes that have been processed for the specified buffer. The client can use this function, in a case where the buffer has terminated due to an error, to obtain the number of bytes that have been processed. If this function is called on a invalid buffer handle, or if the buffer handle has expired, the function returns 0.

Remarks

None.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

One of DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite or DRV_I2S_BufferAddWriteRead function must have been called and a valid buffer handle returned.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet(myI2SHandle, APP_I2SBufferEventHandle,
                                                    (uintptr_t)&myAppObj);

DRV_I2S_BufferAddRead(myI2Shandle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            // We can find out how many bytes were processed in this
            // buffer before the error occurred.

            processedBytes = DRV_I2S_BufferProcessedSizeGet(myI2SHandle);

            break;

        default:
```

```
            break;
        }
    }
```

Parameters

Parameters	Description
bufferhandle	Handle of the buffer of which the processed number of bytes to be obtained.

Function

size_t DRV_I2S_BufferProcessedSizeGet(DRV_HANDLE handle)

d) Miscellaneous Functions

DRV_I2S_BaudSet Function

This function sets the baud.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_BaudSet(DRV_HANDLE handle, uint32_t spiClock, uint32_t baud);
```

Returns

None

Description

This function sets the baud rate for the I2S operation.

Remarks

None.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_HANDLE handle;
uint32_t baud;
uint32_t clock;


// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet(myI2SHandle, APP_I2SBufferEventHandle,
                                                    (uintptr_t)&myAppObj);

// Sets the baud rate to a new value as below
baud = 115200;
clock = 40000000UL;
DRV_I2S_BaudSet(myI2SHandle, clock, baud);

// Further perform the operation needed
DRV_I2S_BufferAddRead(myI2Shandle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);
```

```
if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            // We can find out how many bytes were processed in this
            // buffer before the error occurred.

            processedBytes = DRV_I2S_BufferProcessedSizeGet(myI2SHandle);

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
i2sClock	The Source clock frequency to the i2S module.
baud	The baud to be set.

Function

void DRV_I2S_BaudSet(DRV_HANDLE handle, uint32_t spiClock, uint32_t baud)

DRV_I2S_ErrorGet Function

This function returns the error(if any) associated with the last client request.

Implementation: Dynamic

File

drv_i2s.h

C

```
DRV_I2S_ERROR DRV_I2S_ErrorGet(DRV_HANDLE handle);
```

Returns

A DRV_I2S_ERROR type indicating last known error status.

Description

This function returns the error(if any) associated with the last client request. The DRV_I2S_Read() and DRV_I2S_Write() will update the client error status when these functions return DRV_I2S_READ_ERROR and DRV_I2S_WRITE_ERROR, respectively. If the driver send a

DRV_I2S_BUFFER_EVENT_ERROR to the client, the client can call this function to know the error cause. The error status will be updated on every operation and should be read frequently (ideally immediately after the driver operation has completed) to know the relevant error status.

Remarks

It is the client's responsibility to make sure that the error status is obtained frequently. The driver will update the client error status regardless of whether this has been examined by the client.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_I2S_BUFFER_HANDLE bufferHandle;

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet( myI2SHandle, APP_I2SBufferEventHandle,
                                (uintptr_t)&myAppObj );

DRV_I2S_BufferAddRead( myI2Shandle,&bufferHandle,
                                     myBuffer, MY_BUFFER_SIZE );

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler( DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle )
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_SUCCESS:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_FAILURE:

            // Error handling here.
            // We can find out how many bytes were processed in this
            // buffer before the error occurred. We can also find
            // the error cause.

            processedBytes = DRV_I2S_BufferProcessedSizeGet(myI2SHandle);
            if(DRV_I2S_ERROR_RECEIVE_OVERRUN == DRV_I2S_ErrorGet(myI2SHandle))
            {
                // There was an receive over flow error.
                // Do error handling here.
            }

            break;
```

```
            default:
                break;
        }
    }
```

Parameters

Parameters	Description
bufferhandle	Handle of the buffer of which the processed number of bytes to be obtained.

Function

DRV_I2S_ERROR DRV_I2S_ErrorGet(DRV_HANDLE handle)

DRV_I2S_ReceiveErrorIgnore Function

This function enable/disable ignoring of the receive overflow error.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_ReceiveErrorIgnore(DRV_HANDLE handle, bool errorEnable);
```

Returns

None

Description

A receive overflow is not a critical error; during receive overflow data in the FIFO is not overwritten by receive data. Ignore receive overflow is needed for cases when there is a general performance problem in the system that software must handle properly.

Remarks

None.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_HANDLE handle;
uint32_t baud;
uint32_t baud;*

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet(myI2SHandle, APP_I2SBufferEventHandle,
                                                    (uintptr_t)&myAppObj);

// Enable ignoring of receive overflow error
DRV_I2S_ReceiveErrorIgnore(myI2SHandle, true);

// Further perform the operation needed
DRV_I2S_BufferAddRead(myI2Shandle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}
```

```
// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            // We can find out how many bytes were processed in this
            // buffer before the error occurred.

            processedBytes = DRV_I2S_BufferProcessedSizeGet(bufferHandle);

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
errorIgnore	When set to 'true' enables ignoring of transmit underrun error. When set to 'false' disables ignoring of transmit underrun error.

Function

void DRV_I2S_ReceiveErrorIgnore(DRV_HANDLE handle, bool errorEnable)

DRV_I2S_TransmitErrorIgnore Function

This function enable/disable ignoring of the transmit underrun error.

Implementation: Dynamic

File

drv_i2s.h

C

```
void DRV_I2S_TransmitErrorIgnore(DRV_HANDLE handle, bool errorIgnore);
```

Returns

None

Description

A Transmit underrun error is not a critical error and zeros are transmitted until the SPIxTXB is not empty. Ignore Transmit underrun error is needed for cases when software does not care or does not need to know about underrun conditions.

Remarks

None.

Preconditions

The DRV_I2S_Initialize routine must have been called for the specified I2S driver instance.

DRV_I2S_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_HANDLE handle;
uint32_t baud;
uint32_t baud;*

// myI2SHandle is the handle returned
// by the DRV_I2S_Open function.

// Client registers an event handler with driver. This is done once

DRV_I2S_BufferEventHandlerSet(myI2SHandle, APP_I2SBufferEventHandle,
                                                    (uintptr_t)&myAppObj);

// Enable ignoring of transmit underrun error
DRV_I2S_TransmitErrorIgnore(myI2SHandle, true);

// Further perform the operation needed
DRV_I2S_BufferAddRead(myI2Shandle,&bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_I2S_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_I2SBufferEventHandler(DRV_I2S_BUFFER_EVENT event,
        DRV_I2S_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:

            // Error handling here.
            // We can find out how many bytes were processed in this
            // buffer before the error occurred.

            processedBytes = DRV_I2S_BufferProcessedSizeGet(bufferHandle);

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
errorIgnore	When set to 'true' enables ignoring of transmit underrun error. When set to 'false' disables ignoring of transmit underrun error.

Function

void DRV_I2S_TransmitErrorIgnore(DRV_HANDLE handle, bool errorIgnore)

e) Data Types and Constants

DRV_I2S_AUDIO_PROTOCOL_MODE Enumeration

Identifies the Audio Protocol Mode of the I2S module.

File

drv_i2s.h

C

```
typedef enum {
  DRV_I2S_AUDIO_I2S,
  DRV_I2S_AUDIO_LFET_JUSTIFIED,
  DRV_I2S_AUDIO_RIGHT_JUSTIFIED,
  DRV_I2S_AUDIO_PCM_DSP
} DRV_I2S_AUDIO_PROTOCOL_MODE;
```

Members

Members	Description
DRV_I2S_AUDIO_I2S	I2S Audio Protocol
DRV_I2S_AUDIO_LFET_JUSTIFIED	Left Justified Audio Protocol
DRV_I2S_AUDIO_RIGHT_JUSTIFIED	Right Justified Audio Protocol
DRV_I2S_AUDIO_PCM_DSP	PCM/DSP Audio Protocol

Description

I2S Audio Protocol Mode

This enumeration identifies Audio Protocol Mode of the I2S module.

Remarks

None.

DRV_I2S_BUFFER_EVENT Enumeration

Identifies the possible events that can result from a buffer add request.

File

drv_i2s.h

C

```
typedef enum {
  DRV_I2S_BUFFER_EVENT_COMPLETE,
  DRV_I2S_BUFFER_EVENT_ERROR,
  DRV_I2S_BUFFER_EVENT_ABORT
} DRV_I2S_BUFFER_EVENT;
```

Members

Members	Description
DRV_I2S_BUFFER_EVENT_COMPLETE	Data was transferred successfully.
DRV_I2S_BUFFER_EVENT_ERROR	Error while processing the request
DRV_I2S_BUFFER_EVENT_ABORT	Data transfer aborted (Applicable in DMA mode)

Description

I2S Driver Events

This enumeration identifies the possible events that can result from a buffer add request caused by the client calling either the DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite or DRV_I2S_BufferAddWriteRead functions.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that the client registered with the driver by calling the DRV_I2S_BufferEventHandlerSet function when a buffer transfer request is completed.

DRV_I2S_BUFFER_EVENT_HANDLER Type

Pointer to a I2S Driver Buffer Event handler function

File

drv_i2s.h

C

```
typedef void (* DRV_I2S_BUFFER_EVENT_HANDLER)(DRV_I2S_BUFFER_EVENT event, DRV_I2S_BUFFER_HANDLE
bufferHandle, uintptr_t contextHandle);
```

Returns

None.

Description

I2S Driver Buffer Event Handler Function

This data type defines the required function signature for the I2S driver buffer event handling callback function. A client must register a pointer to a buffer event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive buffer related event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_I2S_BUFFER_EVENT_COMPLETE, this means that the data was transferred successfully.

If the event is DRV_I2S_BUFFER_EVENT_ERROR, this means that the data was not transferred successfully. The bufferHandle parameter contains the buffer handle of the buffer that failed. The DRV_I2S_ErrorGet function can be called to know the error. The DRV_I2S_BufferProcessedSizeGet function can be called to find out how many bytes were processed.

The bufferHandle parameter contains the buffer handle of the buffer that associated with the event.

The context parameter contains a handle to the client context, provided at the time the event handling function was registered using the DRV_I2S_BufferEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the buffer add request.

The buffer handle in bufferHandle expires after this event handler exits. In that the buffer object that was allocated is deallocated by the driver after the event handler exits.

The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

The DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite and DRV_I2S_BufferAddWriteRead functions can be called in the event handler to add a buffer to the driver queue. These functions can only be called to add buffers to the driver whose event handler is running. For example, buffers cannot be added I2S2 driver in I2S1 driver event handler.

Example

```
void APP_MyBufferEventHandler
(
    DRV_I2S_BUFFER_EVENT event,
    DRV_I2S_BUFFER_HANDLE bufferHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_I2S_BUFFER_EVENT_COMPLETE:

            // Handle the completed buffer.
```

```
            break;

        case DRV_I2S_BUFFER_EVENT_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
bufferHandle	Handle identifying the buffer to which the event relates
context	Value identifying the context of the application that registered the event handling function.

DRV_I2S_BUFFER_HANDLE Type

Handle identifying a read or write buffer passed to the driver.

File

drv_i2s.h

C

```
typedef uintptr_t DRV_I2S_BUFFER_HANDLE;
```

Description

I2S Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite, and DRV_I2S_BufferAddWriteRead functions. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from the "buffer add" function is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

DRV_I2S_CLOCK_MODE Enumeration

Identifies the various clock modes of the I2S module.

File

drv_i2s.h

C

```
typedef enum {
  DRV_I2S_CLOCK_MODE_IDLE_LOW_EDGE_RISE,
  DRV_I2S_CLOCK_MODE_IDLE_LOW_EDGE_FALL,
  DRV_I2S_CLOCK_MODE_IDLE_HIGH_EDGE_FALL,
  DRV_I2S_CLOCK_MODE_IDLE_HIGH_EDGE_RISE
} DRV_I2S_CLOCK_MODE;
```

Members

Members	Description
DRV_I2S_CLOCK_MODE_IDLE_LOW_EDGE_RISE	I2S Clock Mode 0 - Idle State Low & Sampling on Rising Edge
DRV_I2S_CLOCK_MODE_IDLE_LOW_EDGE_FALL	I2S Clock Mode 1 - Idle State Low & Sampling on Falling Edge
DRV_I2S_CLOCK_MODE_IDLE_HIGH_EDGE_FALL	I2S Clock Mode 2 - Idle State High & Sampling on Falling Edge
DRV_I2S_CLOCK_MODE_IDLE_HIGH_EDGE_RISE	I2S Clock Mode 3 - Idle State High & Sampling on Rising Edge

Description

I2S Clock Mode Selection

This enumeration identifies the supported clock modes of the I2S module.

Remarks

None.

DRV_I2S_DATA16 Structure

Defines the left and right channel data for 16-bit audio data

File

drv_i2s.h

C

```
typedef struct {
  int16_t leftData;
  int16_t rightData;
} DRV_I2S_DATA16;
```

Members

Members	Description
int16_t leftData;	Left channel data
int16_t rightData;	Right channel data

Description

I2S Driver Audio Data 16

Defines the left and right channel data for 16-bit audio data

Remarks

None.

DRV_I2S_DATA24 Structure

Defines the left and right channel data for 24-bit audio data

File

drv_i2s.h

C

```
typedef struct {
  int32_t leftData : 24;
  int32_t leftDataPad : 8;
  int32_t rightData : 24;
  int32_t rightDataPad : 8;
} DRV_I2S_DATA24;
```

Members

Members	Description
int32_t leftData : 24;	Left channel data
int32_t leftDataPad : 8;	Left channel data pad
int32_t rightData : 24;	Right channel data
int32_t rightDataPad : 8;	Right channel data pad

Description

I2S Driver Audio Data 24

Defines the left and right channel data for 24-bit audio data

Remarks

None.

DRV_I2S_DATA32 Structure

Defines the left and right channel data for 32-bit audio data

File

drv_i2s.h

C

```
typedef struct {
  int32_t leftData;
  int32_t rightDataPad;
} DRV_I2S_DATA32;
```

Members

Members	Description
int32_t leftData;	Left channel data
int32_t rightDataPad;	Right channel data

Description

I2S Driver Audio Data 32

Defines the left and right channel data for 32-bit audio data

Remarks

None.

DRV_I2S_ERROR Enumeration

Defines the possible errors that can occur during driver operation.

File

drv_i2s.h

C

```
typedef enum {
  DRV_I2S_ERROR_NONE,
  DRV_I2S_ERROR_RECEIVE_OVERFLOW,
  DRV_I2S_ERROR_TRANSMIT_UNDERUN,
  DRV_I2S_ERROR_FRAMING,
  DRV_I2S_ERROR_ADDRESS
} DRV_I2S_ERROR;
```

Members

Members	Description
DRV_I2S_ERROR_NONE	Data was transferred successfully.
DRV_I2S_ERROR_RECEIVE_OVERFLOW	Receive overflow error.
DRV_I2S_ERROR_TRANSMIT_UNDERUN	Transmit underrun error.
DRV_I2S_ERROR_FRAMING	Framing error.
DRV_I2S_ERROR_ADDRESS	Channel address error (Applicable in DMA mode)

Description

I2S Driver Error

This data type defines the possible errors that can occur when occur during USART driver operation. These values are returned by DRV_I2S_ErrorGet function.

Remarks

None.

DRV_I2S_MODE Enumeration

Identifies the usage modes of the I2S module.

File

drv_i2s.h

C

```
typedef enum {
  DRV_I2S_MODE_SLAVE,
  DRV_I2S_MODE_MASTER
} DRV_I2S_MODE;
```

Members

Members	Description
DRV_I2S_MODE_SLAVE	I2S Mode Slave
DRV_I2S_MODE_MASTER	I2S Mode Master

Description

I2S Usage Modes Enumeration

This enumeration identifies the whether the I2S module will be used as a master or slave.

Remarks

None.

DRV_I2S_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_i2s.h

C

```
#define DRV_I2S_BUFFER_HANDLE_INVALID ((DRV_I2S_BUFFER_HANDLE)(-1))
```

Description

I2S Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_I2S_BufferAddRead, DRV_I2S_BufferAddWrite and DRV_I2S_BufferAddWriteRead functions if the buffer add request was not successful.

Remarks

None

DRV_I2S_COUNT Macro

Number of valid I2S driver indices

File

drv_i2s.h

C

```
#define DRV_I2S_COUNT
```

Description

I2S Driver Module Count

This constant identifies the maximum number of I2S Driver instances that should be defined by the application. Defining more instances than this constant will waste RAM memory space.

This constant can also be used by the application to identify the number of I2S instances on this microcontroller.

Remarks

This value is part-specific.

DRV_I2S_READ_ERROR Macro

I2S Driver Read Error.

File

drv_i2s.h

C

```
#define DRV_I2S_READ_ERROR ((size_t)(-1))
```

Description

I2S Driver Read Error

This constant is returned by DRV_I2S_Read function when an error occurs.

Remarks

None.

DRV_I2S_WRITE_ERROR Macro

I2S Driver Write Error.

File

drv_i2s.h

C

```
#define DRV_I2S_WRITE_ERROR ((size_t)(-1))
```

Description

I2S Driver Write Error

This constant is returned by DRV_I2S_Write() function when an error occurs.

Remarks

None.

DRV_I2S_INDEX_0 Macro

I2S driver index definitions

File

drv_i2s.h

C

```
#define DRV_I2S_INDEX_0 0
```

Description

Driver I2S Module Index

These constants provide I2S driver index definition.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_I2S_Initialize and DRV_I2S_Open routines to identify the driver instance in use.

DRV_I2S_INDEX_1 Macro

File

drv_i2s.h

C

```
#define DRV_I2S_INDEX_1 1
```

Description

This is macro DRV_I2S_INDEX_1.

DRV_I2S_INDEX_2 Macro

File

drv_i2s.h

C

```
#define DRV_I2S_INDEX_2 2
```

Description

This is macro DRV_I2S_INDEX_2.

DRV_I2S_INDEX_3 Macro

File

drv_i2s.h

C

```
#define DRV_I2S_INDEX_3 3
```

Description

This is macro DRV_I2S_INDEX_3.

DRV_I2S_INDEX_4 Macro

File

drv_i2s.h

C

```
#define DRV_I2S_INDEX_4 4
```

Description

This is macro DRV_I2S_INDEX_4.

DRV_I2S_INDEX_5 Macro

File

drv_i2s.h

C

```
#define DRV_I2S_INDEX_5 5
```

Description

This is macro DRV_I2S_INDEX_5.

DRV_I2S_INTERFACE Structure

This structure defines a structure of I2S Driver function pointers.

File

drv_i2s.h

C

```
typedef struct {
  SYS_MODULE_OBJ (* initialize)(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init);
  void (* deinitialize)(SYS_MODULE_OBJ);
  SYS_STATUS (* status)(SYS_MODULE_OBJ object);
  void (* tasks)(SYS_MODULE_OBJ object);
  void (* tasksError)(SYS_MODULE_OBJ object);
  DRV_HANDLE (* open)(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);
```

```
  void (* close)(const DRV_HANDLE handle);
  void (* bufferAddWrite)(const DRV_HANDLE handle, DRV_I2S_BUFFER_HANDLE *bufferHandle, void *buffer,
size_t size);
  void (* bufferAddRead)(const DRV_HANDLE handle, DRV_I2S_BUFFER_HANDLE *bufferHandle, void *buffer, size_t
size);
  void (* bufferAddWriteRead)(const DRV_HANDLE handle, DRV_I2S_BUFFER_HANDLE *bufferHandle, void
*transmitBuffer, void *receiveBuffer, size_t size);
  size_t (* read)(const DRV_HANDLE handle, uint8_t *buffer, const size_t numBytes);
  size_t (* write)(const DRV_HANDLE handle, uint8_t *buffer, const size_t numBytes);
  void (* eventHandlerSet)(DRV_HANDLE handle, const DRV_I2S_BUFFER_EVENT_HANDLER eventHandler, const
uintptr_t contextHandle);
  size_t (* bufferProcessedSizeGet)(DRV_HANDLE handle);
  size_t (* bufferCombinedQueueSizeGet)(DRV_HANDLE handle);
  void (* bufferQueueFlush)(DRV_HANDLE handle);
  DRV_I2S_ERROR (* errorGet)(DRV_HANDLE handle);
  void (* baudSet)(DRV_HANDLE handle, uint32_t peripheralClock, uint32_t baud);
  void (* setAudioCommunicationMode)(DRV_HANDLE handle, uint8_t audioCommWidth);
  void (* transmitErrorIgnore)(DRV_HANDLE handle, bool errorIgnore);
  void (* receiveErrorIgnore)(DRV_HANDLE handle, bool errorEnable);
} DRV_I2S_INTERFACE;
```

Members

Members	Description
SYS_MODULE_OBJ (* initialize)(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init);	Pointer to the driver Initialization function
void (* deinitialize)(SYS_MODULE_OBJ);	Pointer to the driver Deinitialization function
SYS_STATUS (* status)(SYS_MODULE_OBJ object);	Pointer to the driver Status function
void (* tasks)(SYS_MODULE_OBJ object);	Pointer to the Tasks function
void (* tasksError)(SYS_MODULE_OBJ object);	Pointer to the Error Tasks function
DRV_HANDLE (* open)(const SYS_MODULE_INDEX iDriver, const DRV_IO_INTENT ioIntent);	Pointer to the Open function
void (* close)(const DRV_HANDLE handle);	Pointer to the Close function
void (* bufferAddWrite)(const DRV_HANDLE handle, DRV_I2S_BUFFER_HANDLE *bufferHandle, void *buffer, size_t size);	Pointer to the Buffer Add Write function
void (* bufferAddRead)(const DRV_HANDLE handle, DRV_I2S_BUFFER_HANDLE *bufferHandle, void *buffer, size_t size);	Pointer to the Buffer Add Read function
void (* bufferAddWriteRead)(const DRV_HANDLE handle, DRV_I2S_BUFFER_HANDLE *bufferHandle, void *transmitBuffer, void *receiveBuffer, size_t size);	Pointer to the buffer Add Read Write function
size_t (* read)(const DRV_HANDLE handle, uint8_t *buffer, const size_t numBytes);	Pointer to the driver Read function
size_t (* write)(const DRV_HANDLE handle, uint8_t *buffer, const size_t numBytes);	Pointer to the driver Write function
void (* eventHandlerSet)(DRV_HANDLE handle, const DRV_I2S_BUFFER_EVENT_HANDLER eventHandler, const uintptr_t contextHandle);	Pointer to the driver Buffer Event Handler Set function
size_t (* bufferProcessedSizeGet)(DRV_HANDLE handle);	Pointer to the driver Buffer Processed Size Get function
size_t (* bufferCombinedQueueSizeGet)(DRV_HANDLE handle);	Pointer to the driver Buffer Combined Queue Size Get Function
void (* bufferQueueFlush)(DRV_HANDLE handle);	Pointer to the driver Buffer Queue Flush Function
DRV_I2S_ERROR (* errorGet)(DRV_HANDLE handle);	Pointer to the driver Error Get function
void (* baudSet)(DRV_HANDLE handle, uint32_t peripheralClock, uint32_t baud);	Pointer to the driver Baud Set function
void (* setAudioCommunicationMode)(DRV_HANDLE handle, uint8_t audioCommWidth);	Pointer to the driver Set Audio Communication mode function

void (* transmitErrorIgnore)(DRV_HANDLE handle, bool errorIgnore);	Pointer to the driver Transmit Error Ignore function
void (* receiveErrorIgnore)(DRV_HANDLE handle, bool errorEnable);	Pointer to the driver Receive Error Ignore function

Description

I2S Driver Interface

This structure defines a structure of I2S Driver function pointers. A driver of any peripheral that supports the I2S protocol can export such a structure. The top level I2S Driver abstraction layer will then use this structure to map a I2S Driver call to underlying peripheral driver.

Remarks

None.

Files

Files

Name	Description
drv_i2s.h	I2S Driver Interface header file
drv_i2s_config_template.h	I2S Driver Configuration Template.

Description

drv_i2s.h

I2S Driver Interface header file

Enumerations

	Name	Description
	DRV_I2S_AUDIO_PROTOCOL_MODE	Identifies the Audio Protocol Mode of the I2S module.
	DRV_I2S_BUFFER_EVENT	Identifies the possible events that can result from a buffer add request.
	DRV_I2S_CLOCK_MODE	Identifies the various clock modes of the I2S module.
	DRV_I2S_ERROR	Defines the possible errors that can occur during driver operation.
	DRV_I2S_MODE	Identifies the usage modes of the I2S module.

Functions

	Name	Description
	DRV_I2S_BaudSet	This function sets the baud. **Implementation:** Dynamic
	DRV_I2S_BufferAddRead	Schedule a non-blocking driver read operation. **Implementation:** Dynamic
	DRV_I2S_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Dynamic
	DRV_I2S_BufferAddWriteRead	Schedule a non-blocking driver write-read operation. **Implementation:** Dynamic
	DRV_I2S_BufferCombinedQueueSizeGet	This function returns the number of bytes queued (to be processed) in the buffer queue. **Implementation:** Dynamic
	DRV_I2S_BufferEventHandlerSet	This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Dynamic
	DRV_I2S_BufferProcessedSizeGet	This function returns number of bytes that have been processed for the specified buffer. **Implementation:** Dynamic
	DRV_I2S_BufferQueueFlush	This function flushes off the buffers associated with the client object. **Implementation:** Dynamic
	DRV_I2S_Close	Closes an opened-instance of the I2S driver. **Implementation:** Dynamic
	DRV_I2S_Deinitialize	Deinitializes the specified instance of the I2S driver module. **Implementation:** Dynamic

	DRV_I2S_ErrorGet	This function returns the error(if any) associated with the last client request. **Implementation:** Dynamic
	DRV_I2S_Initialize	Initializes hardware and data for the instance of the I2S module. **Implementation:** Dynamic
	DRV_I2S_Open	Opens the specified I2S driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_I2S_Read	Reads data from the I2S. **Implementation:** Dynamic
	DRV_I2S_ReceiveErrorIgnore	This function enable/disable ignoring of the receive overflow error. **Implementation:** Dynamic
	DRV_I2S_Status	Gets the current status of the I2S driver module. **Implementation:** Dynamic
	DRV_I2S_Tasks	Maintains the driver's receive state machine and implements its ISR. **Implementation:** Dynamic
	DRV_I2S_TasksError	Maintains the driver's error state machine and implements its ISR. **Implementation:** Dynamic
	DRV_I2S_TransmitErrorIgnore	This function enable/disable ignoring of the transmit underrun error. **Implementation:** Dynamic
	DRV_I2S_Write	Writes data to the I2S. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_I2S_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_I2S_COUNT	Number of valid I2S driver indices
	DRV_I2S_INDEX_0	I2S driver index definitions
	DRV_I2S_INDEX_1	This is macro DRV_I2S_INDEX_1.
	DRV_I2S_INDEX_2	This is macro DRV_I2S_INDEX_2.
	DRV_I2S_INDEX_3	This is macro DRV_I2S_INDEX_3.
	DRV_I2S_INDEX_4	This is macro DRV_I2S_INDEX_4.
	DRV_I2S_INDEX_5	This is macro DRV_I2S_INDEX_5.
	DRV_I2S_READ_ERROR	I2S Driver Read Error.
	DRV_I2S_WRITE_ERROR	I2S Driver Write Error.

Structures

	Name	Description
	DRV_I2S_DATA16	Defines the left and right channel data for 16-bit audio data
	DRV_I2S_DATA24	Defines the left and right channel data for 24-bit audio data
	DRV_I2S_DATA32	Defines the left and right channel data for 32-bit audio data
	DRV_I2S_INTERFACE	This structure defines a structure of I2S Driver function pointers.

Types

	Name	Description
	DRV_I2S_BUFFER_EVENT_HANDLER	Pointer to a I2S Driver Buffer Event handler function
	DRV_I2S_BUFFER_HANDLE	Handle identifying a read or write buffer passed to the driver.

Description

I2S Driver Interface

The I2S device driver provides a simple interface to manage the I2S module on Microchip microcontrollers. This file provides the interface definition for the I2S driver.

File Name

drv_i2s.h

Company

Microchip Technology Inc.

drv_i2s_config_template.h

I2S Driver Configuration Template.

Macros

	Name	Description
	DRV_I2S_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_I2S_INDEX	I2S Static Index selection
	DRV_I2S_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_I2S_INTERRUPT_MODE	Macro controls interrupt based operation of the driver
	DRV_I2S_INTERRUPT_SOURCE_ERROR	Defines the interrupt source for the error interrupt
	DRV_I2S_INTERRUPT_SOURCE_RECEIVE	Macro to define the Receive interrupt source in case of static driver
	DRV_I2S_INTERRUPT_SOURCE_TRANSMIT	Macro to define the Transmit interrupt source in case of static driver
	DRV_I2S_PERIPHERAL_ID	Configures the I2S PLIB Module ID
	DRV_I2S_QUEUE_DEPTH_COMBINED	Number of entries of all queues in all instances of the driver.
	DRV_I2S_RECEIVE_DMA_CHAINING_CHANNEL	Macro to defines the I2S Driver Receive DMA Chaining Channel in case of static driver
	DRV_I2S_RECEIVE_DMA_CHANNEL	Macro to defines the I2S Driver Receive DMA Channel in case of static driver
	DRV_I2S_STOP_IN_IDLE	Identifies whether the driver should stop operations in stop in Idle mode.
	DRV_I2S_TRANSMIT_DMA_CHANNEL	Macro to defines the I2S Driver Transmit DMA Channel in case of static driver

Description

I2S Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_i2s_config_template.h

Company

Microchip Technology Inc.

Input Capture Driver Library

This section describes the Input Capture Driver Library.

Introduction

The Input Capture Static Driver provides a high-level interface to manage the Input Capture module on the Microchip family of microcontrollers.

Description

Through the MHC, this driver provides APIs for the following:

- Initializing the module
- Starting/Stopping of the capture
- 16/32-bit data reads
- Buffer empty status

Library Interface

Functions

	Name	Description
	DRV_IC_Initialize	Initializes the Input Capture instance for the specified driver index. **Implementation:** Static
	DRV_IC_BufferIsEmpty	Returns the Input Capture instance buffer empty status for the specified driver index. **Implementation:** Static
	DRV_IC_Capture16BitDataRead	Reads the 16-bit Input Capture for the specified driver index. **Implementation:** Static

	DRV_IC_Capture32BitDataRead	Reads the 32-bit Input Capture for the specified driver index. **Implementation:** Static
	DRV_IC_Start	Starts the Input Capture instance for the specified driver index. **Implementation:** Static
	DRV_IC_Stop	Stops the Input Capture instance for the specified driver index. **Implementation:** Static

Description

This section describes the Application Programming Interface (API) functions of the Input Capture Driver Library.

Functions

DRV_IC_Initialize Function

Initializes the Input Capture instance for the specified driver index.

Implementation: Static

File

help_drv_ic.h

C

```
void DRV_IC_Initialize();
```

Returns

None.

Description

This routine initializes the Input Capture driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters. The driver instance index is independent of the Input Capture module ID. For example, driver instance 0 can be assigned to Input Capture 2.

Remarks

This routine must be called before any other Input Capture routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Function

void DRV_IC_Initialize(void)

DRV_IC_BufferIsEmpty Function

Returns the Input Capture instance buffer empty status for the specified driver index.

Implementation: Static

File

help_drv_ic.h

C

```
bool DRV_IC_BufferIsEmpty();
```

Returns

Boolean

- 1 - Buffer is empty
- 0 - Buffer is not empty

Description

Returns the Input Capture instance buffer empty status for the specified driver index. The function should be called to determine whether or not the IC buffer has data.

Remarks

None.

Preconditions

DRV_IC_Initialize has been called.

Function

bool DRV_IC_BufferIsEmpty(void)

DRV_IC_Capture16BitDataRead Function

Reads the 16-bit Input Capture for the specified driver index.

Implementation: Static

File

help_drv_ic.h

C

```
uint16_t DRV_IC_Capture16BitDataRead();
```

Returns

uint16_t value of the data read from the Input Capture.

Description

This routine reads the 16-bit data for the specified driver index.

Remarks

None.

Preconditions

DRV_IC_Initialize has been called.

Function

uint16_t DRV_IC_Capture16BitDataRead(void)

DRV_IC_Capture32BitDataRead Function

Reads the 32-bit Input Capture for the specified driver index.

Implementation: Static

File

help_drv_ic.h

C

```
uint32_t DRV_IC_Capture32BitDataRead();
```

Returns

uint32_t value of the data read from the Input Capture.

Description

This routine reads the 32-bit data for the specified driver index

Remarks

None.

Preconditions

DRV_IC_Initialize has been called.

Function

uint32_t DRV_IC_Capture32BitDataRead(void)

DRV_IC_Start Function

Starts the Input Capture instance for the specified driver index.

Implementation: Static

File

help_drv_ic.h

C

```
void DRV_IC_Start();
```

Returns

None.

Description

This routine starts the Input Capture driver for the specified driver index, starting an input capture.

Remarks

None.

Preconditions

DRV_IC_Initialize has been called.

Function

void DRV_IC_Start(void)

DRV_IC_Stop Function

Stops the Input Capture instance for the specified driver index.

Implementation: Static

File

help_drv_ic.h

C

```
void DRV_IC_Stop();
```

Returns

None.

Description

This routine stops the Input Capture driver for the specified driver index, stopping an input capture.

Remarks

None.

Preconditions

DRV_IC_Initialize has been called.

Function

void DRV_IC_Stop(void)

Input System Service Touch Driver Library

This section describes the Touch Driver Libraries that support the Input System Service. These libraries are variants of libraries previously created to support the Touch System Service, which is being deprecated by the Input System Service.

Touch Driver Libraries in service of the Touch System Service:

ADC Touch Driver Library This topic describes the ADC Touch Driver Library.

mXT336T Touch Driver Library This topic describes the mXT336T Touch Driver Library.

Input System Service Touch ADC Driver Library

This touch driver library establishes a software resistive touch controller using the Analog-to-Digital Converter (ADC) module. This driver provides application routines to read touch input data from the touch screen. Touch events are detected using an interrupt service routine rather than polling. This driver also allows provides the capability to set translation coefficients that allow an application to map a raw screen values to actual native display size using through 4-point calibration technique.

Using the Library

This topic describes the basic architecture of the Touch ADC Driver Library for the Input System Service and provides information and examples on its use.

Interface Header File:

`<project>/firmware/src/system_config/<target_config>/driver/input/touch_adc/drv_touch_adc.h`

The interface to the Touch ADC Driver library is defined in the drv_touch_adc.h header file related to the currently active project target configuration. This file is automatically generated by MHGC. Any C language source (`.c`) file that uses the ADC Touch Driver library should include this header.

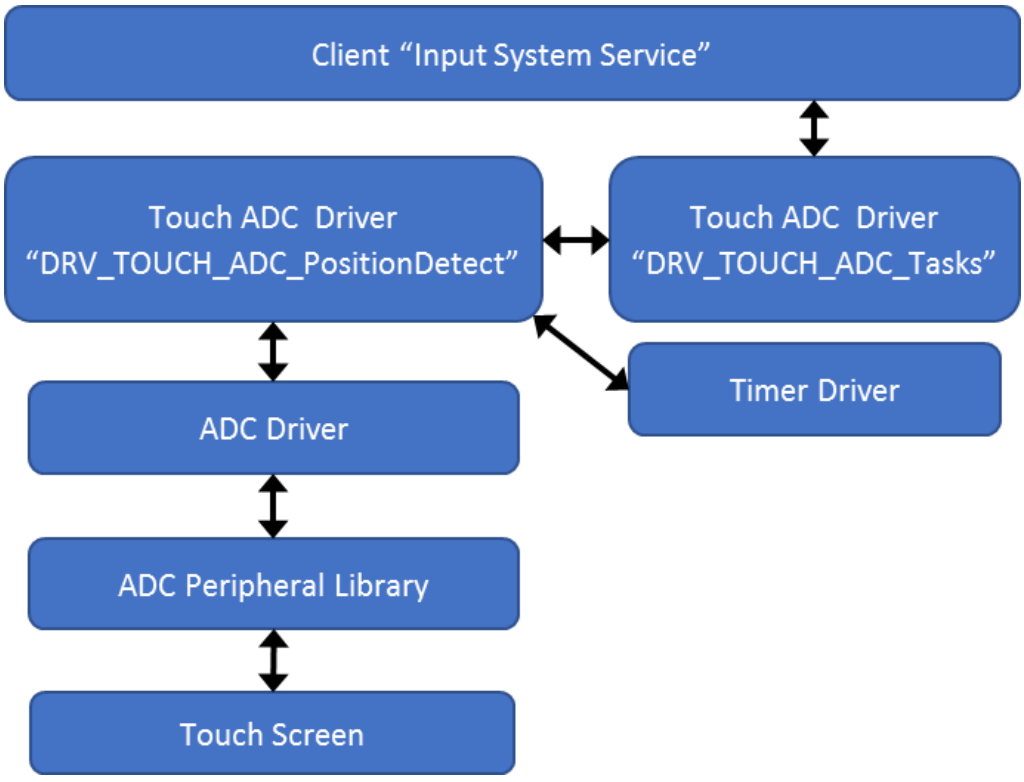
Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

This topic describes the components of the Touch ADC solution in support of the Input System Service.

- Touch ADC Driver – Finite-State machine that biases the x-axis and y-axis analog pins to measure resistance on a touch screen. Identifies touch events up, down or move. Forwards events to Harmony Service, Input System.
- ADC Driver – Provides the 4-wire resistive touch interface. It is used to driver hardware pin configuration to sample and measure touch screen resistance.
- Input system Service – Client level service which makes available touch events to the graphics library.

Touch ADC Driver Abstraction Model.



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Touch ADC Driver module.

Library Interface Section	Description
System Functions	Provides system interfaces, device initialization, deinitialization, open, close, task, and status functions.

How the Library Works

The library provides interfaces to support:

- System functions, which provide system module interfaces, device initialization, deinitialization, open, close, and status functions.
- Timer – rate at which to detect a new position

Of note are the following routine, which are found in:

`<project>/firmware/src/system_config/<target_config>/driver/input/touch_adc/src/drv_touch_adc.c`:

- `DRV_TOUCH_ADC_Tasks()` - sends a move, release, or press event to SYS_Input Services. It is called from `system_tasks.c`.
- `DRV_TOUCH_ADC_PositionDetect()` – performs resistance measurements to determine x and y positions.
- `DRV_TOUCH_ADC_TouchGetX()` – returns the x coordinate.
- `DRV_TOUCH_ADC_TouchGetY()` – returns the y coordinate.
- `DRV_TOUCH_ADCCoefficientSet()` – stores 4-translation calibrate coefficients

Initializing the Driver

Before the Touch ADC Driver can be opened, it must be configured and initialized. MHGC automatically includes the needed #define constants and source code into the project's `system_init.c` file.

The driver initialization is configured through the `DRV_TOUCH_ADC_INIT` data structure that is passed to the `DRV_TOUCH_ADC_Initialize` function.

Opening the Driver

To use the Touch ADC Driver, the application must open the driver. This is done by calling the `DRV_TOUCH_ADC_Open` function. If successful, the `DRV_TOUCH_ADC_Open` function will return a handle to the driver. This handle records the association between the client and the driver instance that was opened. The `DRV_TOUCH_ADC_Open` function may return `DRV_HANDLE_INVALID` in the situation where the driver is not ready to be opened. When this occurs, the application can try opening the driver again. Note that the open function may return an invalid handle in other (error) cases as well. The following code shows an example of the driver being opened.

```
DRV_HANDLE handle;
handle = DRV_TOUCH_ADC_Open( DRV_TOUCH_ADC_INDEX_0,DRV_IO_INTENT_EXCLUSIVE );
if( DRV_HANDLE_INVALID == handle )
{
// Unable
}
```

Tasks Routine

This routine communicates with the System Input Service on the state of the current X and Y positions detected. The routine checks 4 different conditions:

1. Invalid position – does not send an event upstream.
2. Same position – sends a still event to input system service
3. Move position – sends a move event to input system service.
4. Release condition – sends a release event to input system service
5. Press condition- sends a press event to input system service.

Touch Detection

This routine uses the services of the Touch ADC to establish a voltage divider on 4 ADC pins. The table below shows the pin configurations required to read the x and y values. The X value is read by a bias on x-axis and a measurement on ADC Y+ pin. The Y value is read by a bias on the y-axis and a measurement on ADC X+ pin.

Operation	X+ Pin Action	Y+Pin Action	X-GPIO Pin State	Y-GPIO Pin State
Get X	Pin Output (set)	Pin Input (Read X value)	Pin Input	Pin Output (clear)
Get Y	Pin Input (read Y value)	Pin Output (set)	Pin Output (clear)	Pin Input

(Here "set" means SYS_PORTS_PinSet() and "clear" means SYS_PORTS_PinClear().)

Configuring the Library

The configuration of the Touch ADC Driver is performed using the MPLAB Harmony Configurator.

Building the Library

The section lists the files that are available in the Touch ADC Library.

Description

The section lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/input/drv_touch_adc.h`

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library. Note this file is automatically included in the project by MHGC.

Source File Name
`<project>/firmware/src/system_config/<target_config>/driver/input/touch_adc/drv_touch_adc.h`

Required File(s)

All of the required files listed below are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name
`<project>/firmware/src/system_config/<target_config>/driver/input/touch_adc/src/drv_touch_adc.c`

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The ADC Touch Driver Library depends on the following modules:

- Interrupt System Service Library
- Ports System Service Library
- Input System Service Library
- ADC Driver Library

Library Interface

Files

Files

Name	Description
drv_touch_adc.h	Touch ADC Driver interface file.

Description

drv_touch_adc.h

Touch ADC Driver interface file.

Description

Touch ADC Driver Interface File

This is a simple 4-wire resistive touch screen driver. The file consist of touch controller ADC driver interfaces. It implements the driver interfaces which read the touch input data from display overlay through the ADC peripheral.

Remarks

This driver is based on the MPLAB Harmony ADC driver.

File Name

drv_touch_adc.c

Input System Service mXT336T Touch Driver Library

This topic describes the mXT336T Touch Driver Library that supports theInput System Service.

The library provides an interface to manage the mXT336T Touch Driver module on the Microchip family of microcontrollers in different modes of operation. It supports the Input System Service as a client by providing touch events detected by the maXTouch® mXT336T Capacitive Touch Controller.

Description

The MPLAB Harmony mXT336T Touch Driver provides a high-level interface to the mXT336T Capacitive Touch Controller. This driver provides application routines to read the touch input data from the touch screen. Currently, the mXT336T Touch Driver supports non-gestural single-fingered and gestural two-finger touch inputs.

The mXT336T Capacitive Touch Controller notifies the host of the availability of touch input data through an external interrupt on the host. The mXT336T driver allows the application to map a controller pin as the external interrupt pin used by the mXT336T.

The driver contains the standard MPLAB Harmony driver interfaces including: initialization, destruction, status, tasks, open, close, and interrupt-driven read.

The driver contains no direct access to input events. All driver output is directed towards the MPLAB Harmony Input System Service and applications desiring to listen to input events must register with that service.

The aria_quickstart demonstration interfaces with the mXT336T Touch Driver Library. Please refer to the What is MPLAB Harmony? section in Volume I of MPLAB Harmony's built-in documentation for how the driver interacts with the framework.

Using the Library

This topic describes the basis architecture of the mXT336T Touch Driver Library that supports the Input System Service and provides information and examples on its use.

Description

Interface Header File: `./framework/driver/input/touch/mxt336t/drv_mxt336t.h`

The interface to the mXT336T Touch Driver library is defined in the `drv_mxt336t.h` header file. Any C language source (`.c`) file that uses the mXT336T Touch Driver library should include this header.

The mXT336T Touch Driver is based on the Object Protocol for the maXTouch® mXT336T Touchscreen Controller.

The aria_quickstart demonstration interfaces with the mXT336T Touch Driver Library. Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the mXT336T Touch Driver Library on the Microchip family microcontrollers. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

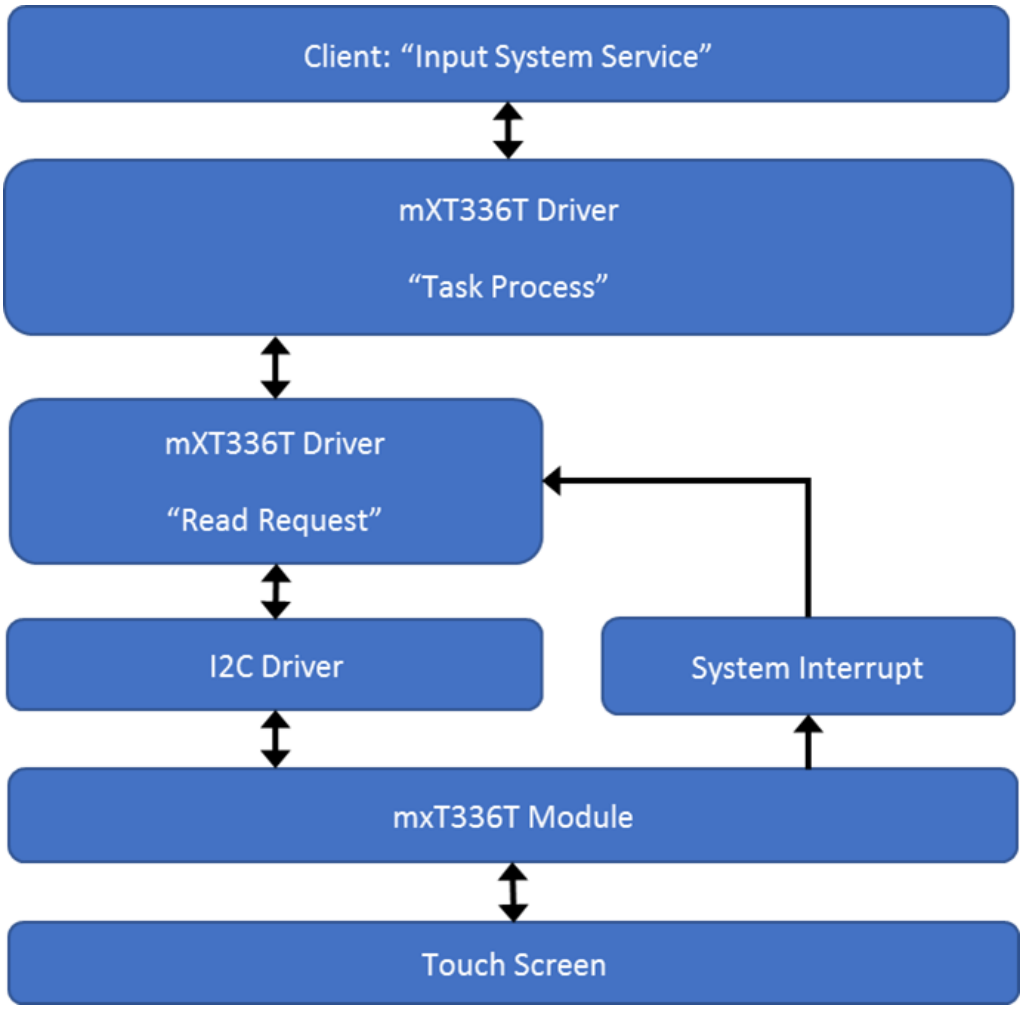
The mXT336T Touch Driver has routines to perform the following operations:

- Sending read request

- Reading the touch input data
- Access to the touch input data

The driver initialization routines allow the application to initialize the driver. The driver must be initialized before it can be used by application. Once the touch input is available (by the assertion of the external interrupt input to the host) a touch input read request is sent to the mXT336t and input data is retrieved in a buffer. The buffer data is then decoded to get the x and y coordinate of the touch screen in the form of the number of pixels. After touch event data has been received it is propagated to the Input System Service which then distributes it to all interested parts of the application.

mXT336T Driver Abstraction Model



Library Overview

This section contains information about how the Touch Driver operates in a system.

Description

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the mXT336T Touch Driver.

Library Interface Section	Description
Device-Specific Functions	Provides mXT336T-specific system module interfaces, device initialization, deinitialization, open, close, task, and status functions.
Generic Functions	Provides generic system module interfaces, device initialization, deinitialization, open, close, task, and status functions.

How the Library Works

This section describes the workings of this Touch Driver library.

Description

The library provides interfaces to support:

- System functions, which provide system module interfaces, device initialization, deinitialization, open, close, task, and status functions.
- Read Request function, which provides Touch input data read request function.

Initializing the Driver

Before the mXT336T Touch Driver can be opened, it must be configured and initialized. The driver build time configuration is defined by the configuration macros. Refer to the Building the Library section for the location of and more information on the various configuration macros and how these macros should be designed. The driver initialization is configured through the DRV_MXT336T_INIT data structure that is passed to the DRV_MXT336T_Initialize function. The initialization parameters include the interrupt source, interrupt pin remap configuration and touch screen resolution. The following code shows an example of initializing the mXT336T Touch Driver.

Example:

```
/* The following code shows an example of designing the
* DRV_TOUCH_INIT data structure. It also shows how an example
* usage of the DRV_TOUCH_MXT336T_Initialize function.
*/
This entire example section can be replaced with:

const DRV_MXT336T_INIT drvMXT336TInitData =
{
.drvOpen =  DRV_I2C_Open,
.orientation = 0,
.horizontalResolution = 480,
.verticalResolution = 272,
};

sysObj.drvMXT336T = DRV_MXT336T_Initialize(0, (SYS_MODULE_INIT *)&drvMXT336TInitData);
```

Touch Input Read Request

To read the touch input from the mXT336T touch controller device, a read request must be registered. This is done by calling DRV_MXT336T_ReadRequest. If successful, it registers a buffer read request to the I2C command queue. It also adds an input decode command to the mXT336T command queue once the I2C returns with touch input data. It can return error if the driver instance object is invalid or the mXT336T command queue is full. The read request is to be called from the mXT336T ISR. This ISR is triggered once the touch input is available.

The following code shows an example of a mXT336T read request registration:

```
SYS_MODULE_OBJ object; // Returned from DRV_TOUCH_MXT336T_Initialize
void ISR(_EXTERNAL_INT_VECTOR, ipl5) _IntHandlerDrvMxt336t(void)
{
DRV__MXT336T_ReadRequest ( object );
// Do Other Tasks
.
.
.
}
```

Tasks Routine

This routine processes the mXT336T commands from the command queue. If the state of the command is initialize or done it returns. If the read request registration is successful the state of command is to decode input. The tasks routine decodes the input and updates the global variables storing the touch input data in form of x and y coordinates. The mXT336T Touch Driver task routine is to be called from SYS_Tasks. The following code shows an example:

```
SYS_MODULE_OBJ drvMXT336T;
SYS_MODULE_OBJ drvMxt0;; // Returned from DRV_TOUCH_MXT336T_Initialize
void SYS_Tasks( void )
{
DRV_MXT336T_Tasks(sysObj.drvMXT336T);
DRV_MXT_Tasks(sysObj.drvMxt0);
// Do other tasks
}
```

Configuring in MPLAB Harmony Configurator

The graphics demo aria_quickstart has several target configurations that provide examples of driver setup. Any configuration ending in meb2 or _meb2_wvga can be used.

The MPLAB Harmony Configurator Pin Settings tab should be configured to select the correct pin for the external interrupt. For example,

pic32mk_gp_db:

91	RF6	5V	MXT336T_TOUCH_INT	GPIO_CN

pic32mz_da_sk_extddr:

A14	RB1	-	MXT336T_TOUCH_INT	INT4

pic32mz_da_sk_intddr, pic32mz_da_noddr:

B9	RB1	-	MXT336T_TOUCH_INT	INT4

pic32mz_ef_sk:

23	RE8	-	MXT336T_TOUCH_INT	INT1

Library Interface

Files

Files

Name	Description
drv_input_mxt336t.h	Touch controller MXT336T Driver interface header file.

Description

drv_input_mxt336t.h

Touch controller MXT336T Driver interface header file.

Description

Touch Controller MXT336T Driver Interface File

This header file describes the macros, data structure and prototypes of the touch controller MXT336T driver interface.

File Name

drv_MXT336T.c

MIIM Driver Library

This section describes the MII Management (MIIM) Driver Library.

Introduction

The MIIM Driver library provides access to the MII Management interface (MIIM) of the Microchip PIC32 microcontrollers.

Description

The MIIM Driver is implemented as a driver object that provides APIs for:

- Asynchronous read/write and scan operations for accessing the external PHY registers
- Notification when MIIM operations have completed

- Driver status information
- Possibility to query or abort an ongoing operation.

Using the Library

This topic describes the basic architecture of the MIIM Driver Library and provides information and examples about its use.

Description

Interface Header File: `drv_miim.h`

The interface to the MIIM library is defined in the `drv_miim.h` header file.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the MIIM module on the Microchip family of microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

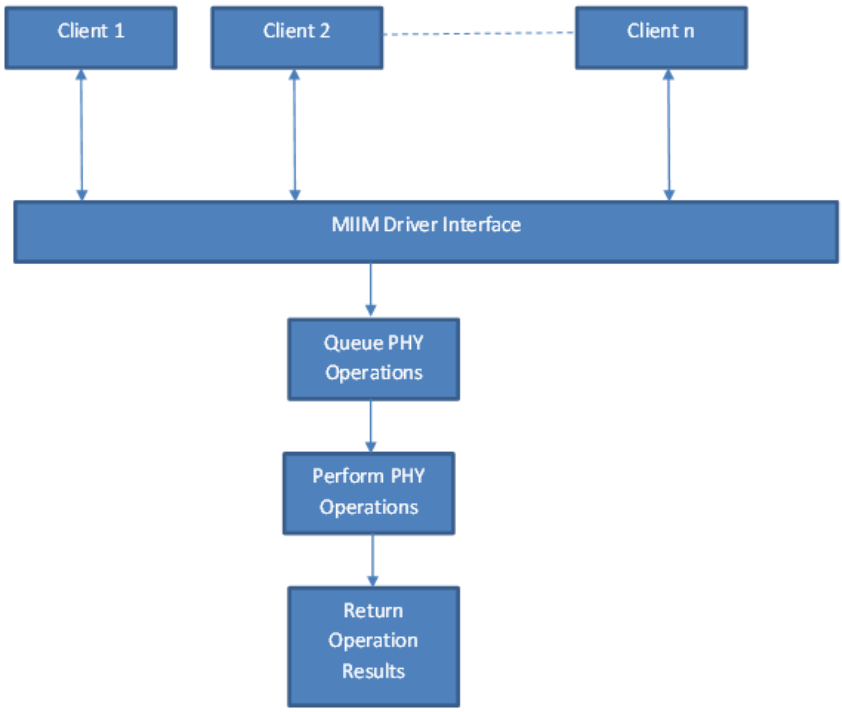
Description

The MIIM driver clients access PHY registers using the MIIM Driver API. The driver abstracts out the hardware details of the MIIM interface and provides a PHY register access mechanism to the application. The MIIM Driver provides read, write, and scan access to the PHY registers, together with driver and operation status APIs. The driver schedules operations requested by multiple clients and performs them sequentially, informing the clients about the operations outcome.

The user can poll for a certain operation status or can register callbacks to be notified of the completion of a scheduled operation.

A scheduled operation can be aborted, if not yet started.

MIIM Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information about how the driver operates in a system.

Description

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the MIIM module.

Library Interface Section	Description
Functions	This section provides general interface routines.

Data Types and Constants	This section provides various definitions describing this API.

Configuring the Library

This section contains related configuration macros.

Macros

	Name	Description
	DRV_MIIM_INDEX_0	MIIM driver index definitions.
	DRV_MIIM_INDEX_COUNT	Number of valid MIIM driver indices.
	_DRV_MIIM_CONFIG_H	This is macro _DRV_MIIM_CONFIG_H.
	DRV_MIIM_CLIENT_OP_PROTECTION	Enables/Disables Client Operation Protection feature.
	DRV_MIIM_COMMANDS	Enables/Disables MIIM commands feature.
	DRV_MIIM_INSTANCE_CLIENTS	Selects the maximum number of clients.
	DRV_MIIM_INSTANCE_OPERATIONS	Selects the maximum number of simultaneous operations for an instance.
	DRV_MIIM_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.

Description

The configuration of the MIIM Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the MIIM Driver. Based on the selections made, the MIIM Driver may support the selected features. These configuration settings will apply to all instances of the MIIM Driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_MIIM_INDEX_0 Macro

MIIM driver index definitions.

File

drv_miim.h

C

```
#define DRV_MIIM_INDEX_0 0
```

Description

MIIM Driver Module Index Numbers

These constants provide the MIIM driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_MIIM_Initialize and DRV_MIIM_Open routines to identify the driver instance in use.

DRV_MIIM_INDEX_COUNT Macro

Number of valid MIIM driver indices.

File

drv_miim.h

C

```
#define DRV_MIIM_INDEX_COUNT 1
```

Description

MIIM Driver Module Index Count

This constant identifies the number of valid MIIM driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from part-specific header files defined as part of the peripheral libraries.

_DRV_MIIM_CONFIG_H Macro

File

drv_miim_config.h

C

```
#define _DRV_MIIM_CONFIG_H
```

Description

This is macro _DRV_MIIM_CONFIG_H.

DRV_MIIM_CLIENT_OP_PROTECTION Macro

Enables/Disables Client Operation Protection feature.

File

drv_miim_config.h

C

```
#define DRV_MIIM_CLIENT_OP_PROTECTION 0
```

Description

MIIM client Operation Protection

Because of the recirculation of the operation handles and client handles the possibility exists that a misbehaved client inadvertently gets the results of a previous completed operations that now belongs to a different client. When this feature is enabled, extra protection is added for an operation handle to uniquely identify a client that has started the operation and extra check is done that operation belongs to the client that asks for the result.

Remarks

Set the value to 1 to enable, 0 to disable the feature.

Enabling this feature requires a small overhead in code and data size.

DRV_MIIM_COMMANDS Macro

Enables/Disables MIIM commands feature.

File

drv_miim_config.h

C

```
#define DRV_MIIM_COMMANDS 0
```

Description

MIIM PHY Commands

Adds a MIIM command to the TCP/IP command menu allowing to read/write a PHY register.

Remarks

Set the value to 1 to enable, 0 to disable the feature.

Currently the MIIM commands are integrated in the TCP/IP commands. To have the MIIM commands available the TCP/IP commands need to be enabled.

DRV_MIIM_INSTANCE_CLIENTS Macro

Selects the maximum number of clients.

File

drv_miim_config.h

C

```
#define DRV_MIIM_INSTANCE_CLIENTS 2
```

Description

MIIM number of clients

This definition select the MIIM Maximum Number of Clients per instance.

Remarks

By default the 1st MIIM client is the DRV_ETHPHY. An extra client is allowed.

DRV_MIIM_INSTANCE_OPERATIONS Macro

Selects the maximum number of simultaneous operations for an instance.

File

drv_miim_config.h

C

```
#define DRV_MIIM_INSTANCE_OPERATIONS 4
```

Description

MIIM instance operations

This definition selects the maximum number of simultaneous operations that can be supported by this driver. Note that this represents operations for all clients

Remarks

None.

DRV_MIIM_INSTANCES_NUMBER Macro

Selects the maximum number of hardware instances that can be supported by the dynamic driver.

File

drv_miim_config.h

C

```
#define DRV_MIIM_INSTANCES_NUMBER 1
```

Description

MIIM hardware instance configuration

This definition selects the maximum number of hardware instances that can be supported by the dynamic driver. Usually set to 1.

Remarks

None.

Building the Library

This section lists the files that are available in the MIIM Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/miim/`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_miim.h`	This is the MIIM Driver Library's interface header file.
`/config/drv_miim.config.h`	This file contains the configuration macros.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_miim.c	This file contains the source code for the dynamic implementation of the MIIM Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Library Interface

This section describes the Application Programming Interface (API) functions of the MIIM Driver Library.
Refer to each section for a detailed description.

a) Functions

	Name	Description
	DRV_MIIM_ClientStatus	Gets the current client-specific status the MIIM driver.
	DRV_MIIM_Close	Closes an opened instance of the MIIM driver.
	DRV_MIIM_Deinitialize	Deinitializes the specified instance of the MIIM driver module.
	DRV_MIIM_DeregisterCallback	Deregisters an notification callback function for the client operations.
	DRV_MIIM_Initialize	Initializes the MIIM driver.
	DRV_MIIM_Open	Opens the specified MIIM driver instance and returns a handle to it.
	DRV_MIIM_OperationAbort	Aborts a current client operation initiated by the MIIM driver.
	DRV_MIIM_OperationResult	Gets the result of a client operation initiated by the MIIM driver.
	DRV_MIIM_Read	Initiates a SMI/MIIM read transaction.
	DRV_MIIM_RegisterCallback	Registers an notification callback function for the client operations.
	DRV_MIIM_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings.
	DRV_MIIM_Scan	Initiates a SMI/MIIM scan (periodic read)transaction.
	DRV_MIIM_Setup	Sets up a MIIM client.
	DRV_MIIM_Status	Provides the current status of the MIIM driver module.
	DRV_MIIM_Tasks	Maintains the driver's state machine.
	DRV_MIIM_Write	Initiates a SMI/MIIM write transaction.

b) Data Types and Constants

	Name	Description
	DRV_MIIM_INIT	Contains all the data necessary to initialize the MIIM device.
	DRV_MIIM_OBJECT_BASE	Declaration of a MIIM base object.
	DRV_MIIM_CALLBACK_HANDLE	Handle that identifies a client registration operation.
	DRV_MIIM_CLIENT_STATUS	Defines the possible results of operations that can succeed or fail
	DRV_MIIM_OPERATION_CALLBACK	Notification function that will be called when a MIIM operation is completed and the driver client needs to be notified.
	DRV_MIIM_OPERATION_FLAGS	List of flags that apply to a client operation.
	DRV_MIIM_OPERATION_HANDLE	MIIM operation handle.
	DRV_MIIM_SETUP	Contains all the data necessary to set up the MIIM device.
	DRV_MIIM_SETUP_FLAGS	List of flags that apply to a client setup operation.
	DRV_MIIM_OBJECT_BASE_Default	The supported basic MIIM driver (DRV_MIIM_OBJECT_BASE). This object is implemented by default as using the standard MIIM interface. It can be overwritten dynamically when needed.

a) Functions

DRV_MIIM_ClientStatus Function

Gets the current client-specific status the MIIM driver.

File

drv_miim.h

C

```
DRV_MIIM_CLIENT_STATUS DRV_MIIM_ClientStatus(DRV_HANDLE handle);
```

Returns

- DRV_MIIM_CLIENT_STATUS_READY - if the client handle represents a valid MIIM client
- DRV_MIIM_CLIENT_STATUS_ERROR - if the client handle is an invalid MIIM client

Description

This function gets the client-specific status of the MIIM driver associated with the given handle.

Remarks

This function can be used to check that a client handle points to a valid MIIM client. The MIIM driver queues operations so it will always return DRV_MIIM_CLIENT_STATUS_READY.

Preconditions

- The DRV_MIIM_Initialize routine must have been called.
- DRV_MIIM_Open must have been called to obtain a valid opened device handle.

Example

Function

DRV_MIIM_CLIENT_STATUS DRV_MIIM_ClientStatus(DRV_HANDLE handle)

DRV_MIIM_Close Function

Closes an opened instance of the MIIM driver.

File

drv_miim.h

C

```
void DRV_MIIM_Close(DRV_HANDLE handle);
```

Returns

None

Description

This function closes an opened instance of the MIIM driver, invalidating the handle.

Remarks

- After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_MIIM_Open before the caller may use the driver again.
- Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_MIIM_Initialize routine must have been called for the specified MIIM driver instance.

DRV_MIIM_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_MIIM_Open
```

```
DRV_MIIM_Close(handle);
```

Function

void DRV_MIIM_Close(DRV_HANDLE handle)

DRV_MIIM_Deinitialize Function

Deinitializes the specified instance of the MIIM driver module.

File

drv_miim.h

C

```
void DRV_MIIM_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the specified instance of the MIIM driver module, disabling its operation (and any hardware)and invalidates all of the internal data.

Remarks

- Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

Preconditions

The DRV_MIIM_Initialize function must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

Function

void DRV_MIIM_Deinitialize (SYS_MODULE_OBJ object)

DRV_MIIM_DeregisterCallback Function

Deregisters an notification callback function for the client operations.

File

drv_miim.h

C

```
DRV_MIIM_RESULT DRV_MIIM_DeregisterCallback(DRV_HANDLE handle, DRV_MIIM_CALLBACK_HANDLE cbHandle);
```

Returns

- DRV_MIIM_RES_OK if the operation succeeded.
- an error code otherwise

Description

This function deregisters a previously registered client notification callback function.

Remarks

There is only one notification callback function available per client. To register a new callback function use DRV_MIIM_DeregisterCallback first.

Preconditions

- The DRV_MIIM_Initialize routine must have been called.
- DRV_MIIM_Open must have been called to obtain a valid opened device handle.

Example

Function

DRV_MIIM_RESULT DRV_MIIM_DeregisterCallback(DRV_HANDLE handle, DRV_MIIM_CALLBACK_HANDLE cbHandle);

DRV_MIIM_Initialize Function

Initializes the MIIM driver.

File

drv_miim.h

C

```
SYS_MODULE_OBJ DRV_MIIM_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

- a valid handle to a driver object, if successful.
- SYS_MODULE_OBJ_INVALID if initialization failed.

Description

This function initializes the MIIM driver, making it ready for clients to open and use it.

Remarks

- This function must be called before any other MIIM routine is called.
- This function should only be called once during system initialization unless DRV_MIIM_Deinitialize is called to deinitialize the driver instance.
- The returned object must be passed as argument to DRV_MIIM_Reinitialize, DRV_MIIM_Deinitialize, DRV_MIIM_Tasks and DRV_MIIM_Status routines.

Preconditions

None.

Example

Function

SYS_MODULE_OBJ DRV_MIIM_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init)

DRV_MIIM_Open Function

Opens the specified MIIM driver instance and returns a handle to it.

File

drv_miim.h

C

```
DRV_HANDLE DRV_MIIM_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

- valid open-instance handle if successful (a number identifying both the caller and the module instance).
- DRV_HANDLE_INVALID if an error occurs

Description

This function opens the specified MIIM driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_MIIM_Close routine is called.

This function will NEVER block waiting for hardware.

Preconditions

The DRV_MIIM_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_MIIM_Open(DRV_MIIM_INDEX_0, 0);
if (DRV_HANDLE_INVALID == handle)
```

```
{
    // Unable to open the driver
}
```

Function

DRV_HANDLE DRV_MIIM_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent)

DRV_MIIM_OperationAbort Function

Aborts a current client operation initiated by the MIIM driver.

File

drv_miim.h

C

```
DRV_MIIM_RESULT DRV_MIIM_OperationAbort(DRV_HANDLE handle, DRV_MIIM_OPERATION_HANDLE opHandle);
```

Returns

DRV_MIIM_RES_OK for success; operation has been aborted

< 0 - an error has occurred and the operation could not be completed

Description

Aborts a current client operation initiated by the MIIM driver.

Remarks

This operation will stop/abort a scan operation started by DRV_MIIM_Scan.

Preconditions

- The DRV_MIIM_Initialize routine must have been called.
- DRV_MIIM_Open must have been called to obtain a valid opened device handle.
- A driver operation was started

Example

Function

DRV_MIIM_RESULT DRV_MIIM_OperationAbort(DRV_HANDLE handle, DRV_MIIM_OPERATION_HANDLE opHandle)

DRV_MIIM_OperationResult Function

Gets the result of a client operation initiated by the MIIM driver.

File

drv_miim.h

C

```
DRV_MIIM_RESULT DRV_MIIM_OperationResult(DRV_HANDLE handle, DRV_MIIM_OPERATION_HANDLE opHandle, uint16_t*
pOpData);
```

Returns

- DRV_MIIM_RESULT value describing the current operation result: DRV_MIIM_RES_OK for success; operation has been completed successfully and pOpData updated DRV_MIIM_RES_PENDING operation is in progress an DRV_MIIM_RESULT error code if the operation failed.

Description

Returns the result of a client operation initiated by the MIIM driver.

Remarks

This function will not block for hardware access and will immediately return the current status.

This function returns the result of the last driver operation. It will return DRV_MIIM_RES_PENDING if an operation is still in progress. Otherwise a DRV_MIIM_RESULT describing the operation outcome.

Note that for a scan operation DRV_MIIM_RES_PENDING will be returned when there's no new scan data available. DRV_MIIM_RES_OK means the scan data is fresh.

Preconditions

- The DRV_MIIM_Initialize routine must have been called.
- DRV_MIIM_Open must have been called to obtain a valid opened device handle.
- A driver operation was started

Example

Function

DRV_MIIM_RESULT DRV_MIIM_OperationResult(DRV_HANDLE handle, DRV_MIIM_OPERATION_HANDLE opHandle, uint16_t* pOpData)

DRV_MIIM_Read Function

Initiates a SMI/MIIM read transaction.

File

drv_miim.h

C

```
DRV_MIIM_OPERATION_HANDLE DRV_MIIM_Read(DRV_HANDLE handle, unsigned int rIx, unsigned int phyAdd,
DRV_MIIM_OPERATION_FLAGS opFlags, DRV_MIIM_RESULT* pOpResult);
```

Returns

A not NULL DRV_MIIM_OPERATION_HANDLE if the operation was successfully scheduled. NULL if the operation failed. More details in pOpResult.

Description

This function initiates a SMI/MIIM read transaction for a given MIIM register.

Remarks

If operation was scheduled successfully, the result will be DRV_MIIM_RES_PENDING. Otherwise an error code will be returned.

Upon the operation completion:

- If the operation is to be discarded (DRV_MIIM_OPERATION_FLAG_DISCARD is set) there will be no notification to the client. The operation associated resources will be released.

- If the operation is not to be discarded, then:
  - if the client has registered an operation notification callback (DRV_MIIM_RegisterCallback) then the callback will be called. After that the operation associated resources will be released.
  - if there is no notification callback the MIIM driver will wait for the client to poll and read the operation result using DRV_MIIM_OperationResult(). Only then the operation will be released.

A completed non-discardable operation will remain available for returning the result until the client is somehow notified of the operation result. When polling is used, DRV_MIIM_OperationResult() needs to be called to free the operation associated resources.

Preconditions

- The DRV_MIIM_Initialize routine must have been called.
- DRV_MIIM_Open must have been called to obtain a valid opened device handle.

Example

Function

DRV_MIIM_OPERATION_HANDLE DRV_MIIM_Read(DRV_HANDLE handle, unsigned int rIx, unsigned int phyAdd, DRV_MIIM_OPERATION_FLAGS opFlags, DRV_MIIM_RESULT* pOpResult);

DRV_MIIM_RegisterCallback Function

Registers an notification callback function for the client operations.

File

drv_miim.h

C

```
DRV_MIIM_CALLBACK_HANDLE DRV_MIIM_RegisterCallback(DRV_HANDLE handle, DRV_MIIM_OPERATION_CALLBACK
cbFunction, DRV_MIIM_RESULT* pRegResult);
```

Returns

- a valid DRV_MIIM_CALLBACK_HANDLE if the operation succeeded.
- NULL otherwise

Description

This function registers a client callback function. The function will be called by the MIIM driver when a scheduled operation is completed.

Remarks

There is only one notification callback function available per client. To register a new callback function use DRV_MIIM_DeregisterCallback first.

Preconditions

- The DRV_MIIM_Initialize routine must have been called.
- DRV_MIIM_Open must have been called to obtain a valid opened device handle.

Example

Function

DRV_MIIM_CALLBACK_HANDLE DRV_MIIM_RegisterCallback(DRV_HANDLE handle, DRV_MIIM_OPERATION_CALLBACK cbFunction, DRV_MIIM_RESULT* pRegResult);

DRV_MIIM_Reinitialize Function

Reinitializes the driver and refreshes any associated hardware settings.

File

drv_miim.h

C

```
void DRV_MIIM_Reinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
```

Returns

None.

Description

This function reinitializes the driver and refreshes any associated hardware settings using the initialization data given, but it will not interrupt any ongoing operations.

Remarks

- This function can be called multiple times to reinitialize the module.
- This operation can be used to refresh any supported hardware registers as specified by the initialization data or to change the power state of the module.

- This function is currently NOT IMPLEMENTED.

Preconditions

The DRV_MIIM_Initialize function must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

Function

void DRV_MIIM_Reinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init)

DRV_MIIM_Scan Function

Initiates a SMI/MIIM scan (periodic read)transaction.

File

drv_miim.h

C

```
DRV_MIIM_OPERATION_HANDLE DRV_MIIM_Scan(DRV_HANDLE handle, unsigned int rIx, unsigned int phyAdd,
DRV_MIIM_OPERATION_FLAGS opFlags, DRV_MIIM_RESULT* pOpResult);
```

Returns

A not NULL DRV_MIIM_OPERATION_HANDLE if the operation was successfully scheduled. NULL if the operation failed. More details in pOpResult.

Description

This function initiates a SMI/MIIM scan transaction for a given MIIM register.

Remarks

If operation was scheduled successfully, the result will be DRV_MIIM_RES_PENDING. Otherwise an error code will be returned.

When a new scan result is available:

- If the operation is to be discarded (DRV_MIIM_OPERATION_FLAG_DISCARD is set) there will be no notification to the client.

- If the operation is not to be discarded, then:
  - if the client has registered an operation notification callback (DRV_MIIM_RegisterCallback) then the notification callback will be called.
  - if there is no notification callback the MIIM driver will wait for the client to poll and read the operation result using DRV_MIIM_OperationResult(). Only then the operation will be released.

A scheduled scan operation will remain active in the background and will be available for returning the scan results. When polling is used, DRV_MIIM_OperationResult()will return the latest scan result. The operation associated resources will be released and scan stopped only when DRV_MIIM_OperationAbort() is called.

While scan is active all other transactions (including from other clients) will be inhibited! Use carefully!

Preconditions

- The DRV_MIIM_Initialize routine must have been called.
- DRV_MIIM_Open must have been called to obtain a valid opened device handle.

Example

Function

DRV_MIIM_OPERATION_HANDLE DRV_MIIM_Scan(DRV_HANDLE handle, unsigned int rIx, unsigned int phyAdd, DRV_MIIM_OPERATION_FLAGS opFlags, DRV_MIIM_RESULT* pOpResult);

DRV_MIIM_Setup Function

Sets up a MIIM client.

File

drv_miim.h

C

```
DRV_MIIM_RESULT DRV_MIIM_Setup(DRV_HANDLE handle, const DRV_MIIM_SETUP* pSetUp);
```

Returns

- DRV_MIIM_RES_OK if the setup operation has been performed successfully
- an DRV_MIIM_RESULT error code if the set up procedure failed.

Description

This function performs the set up of a MIIM client. It programs the MIIM operation using the supplied frequencies.

Remarks

None.

Preconditions

- The DRV_MIIM_Initialize routine must have been called.

- DRV_MIIM_Open must have been called to obtain a valid device handle.

Example

Function

DRV_MIIM_RESULT DRV_MIIM_Setup(DRV_HANDLE handle, const DRV_MIIM_SETUP* pSetUp)

DRV_MIIM_Status Function

Provides the current status of the MIIM driver module.

File

drv_miim.h

C

```
SYS_STATUS DRV_MIIM_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that any previous module operation for the specified module has completed
- SYS_STATUS_BUSY - Indicates that a previous module operation for the specified module has not yet completed
- SYS_STATUS_ERROR - Indicates that the specified module is in an error state

Description

This function provides the current status of the MIIM driver module.

Remarks

- Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.
- SYS_STATUS_BUSY - Indicates that the driver is busy with a previous system level operation and cannot start another
- SYS_STATUS_ERROR - Indicates that the driver is in an error state
- Any value less than SYS_STATUS_ERROR is also an error state.
- SYS_MODULE_DEINITIALIZED - Indicates that the driver has been deinitialized
- If the status operation returns SYS_STATUS_BUSY, the a previous system level operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.
- The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.
- This function will NEVER block waiting for hardware.
- If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The DRV_MIIM_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_MIIM_Initialize
SYS_STATUS          status;

status = DRV_MIIM_Status(object);
if (SYS_STATUS_ERROR >= status)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_MIIM_Initialize

Function

SYS_STATUS DRV_MIIM_Status (SYS_MODULE_OBJ object)

DRV_MIIM_Tasks Function

Maintains the driver's state machine.

File

drv_miim.h

C

```
void DRV_MIIM_Tasks(SYS_MODULE_OBJ object);
```

Returns

None

Description

This function is used to maintain the driver's internal state machine.

Remarks

- This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks)
- This function will never block or access any resources that may cause it to block.

Preconditions

The DRV_MIIM_Initialize routine must have been called for the specified MIIM driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_MIIM_Initialize

while (true)
{
    DRV_MIIM_Tasks (object);

    // Do other tasks
}
```

Function

void DRV_MIIM_Tasks(SYS_MODULE_OBJ object)

DRV_MIIM_Write Function

Initiates a SMI/MIIM write transaction.

File

drv_miim.h

C

```
DRV_MIIM_OPERATION_HANDLE DRV_MIIM_Write(DRV_HANDLE handle, unsigned int rIx, unsigned int phyAdd, uint16_t
wData, DRV_MIIM_OPERATION_FLAGS opFlags, DRV_MIIM_RESULT* pOpResult);
```

Returns

A not NULL DRV_MIIM_OPERATION_HANDLE if the operation was successfully scheduled. NULL if the operation failed. More details in pOpResult.

Description

This function initiates a SMI/MIIM write transaction for a given MIIM register.

Remarks

If operation was scheduled successfully, the result will be DRV_MIIM_RES_PENDING. Otherwise an error code will be returned.

Upon the operation completion:

- If the operation is to be discarded (DRV_MIIM_OPERATION_FLAG_DISCARD is set) there will be no notification to the client. The operation associated resources will be released.

- If the operation is not to be discarded, then:

- if the client has registered an operation notification callback (DRV_MIIM_RegisterCallback) then the notification callback will be called. After that the operation associated resources will be released.
- if there is no notification callback the MIIM driver will wait for the client to poll and read the operation result using DRV_MIIM_OperationResult(). Only then the operation will be released.

A completed non-discardable operation will remain available for returning the result until the client is somehow notified of the operation result. When polling is used, DRV_MIIM_OperationResult() needs to be called to free the operation associated resources.

A write operation normally uses DRV_MIIM_OPERATION_FLAG_DISCARD if it is not interested when the operation has completed.

Preconditions

- The DRV_MIIM_Initialize routine must have been called.
- DRV_MIIM_Open must have been called to obtain a valid opened device handle.

Example

Function

DRV_MIIM_OPERATION_HANDLE DRV_MIIM_Write(DRV_HANDLE handle, unsigned int rIx, unsigned int phyAdd, uint16_t wData, DRV_MIIM_OPERATION_FLAGS opFlags, DRV_MIIM_RESULT* pOpResult);

b) Data Types and Constants

DRV_MIIM_INIT Structure

Contains all the data necessary to initialize the MIIM device.

File

drv_miim.h

C

```
struct DRV_MIIM_INIT {
  SYS_MODULE_INIT moduleInit;
  uintptr_t ethphyId;
};
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
uintptr_t ethphyId;	Identifies peripheral (PLIB-level) ID

Description

MIIM Device Driver Initialization Data

This data structure contains all the data necessary to initialize the MIIM device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_MIIM_Initialize routine.

DRV_MIIM_OBJECT_BASE Structure

Declaration of a MIIM base object.

File

drv_miim.h

C

```
struct DRV_MIIM_OBJECT_BASE {
  SYS_MODULE_OBJ (* DRV_MIIM_Initialize)(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
  void (* DRV_MIIM_Reinitialize)(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
  void (* DRV_MIIM_Deinitialize)(SYS_MODULE_OBJ object);
  SYS_STATUS (* DRV_MIIM_Status)(SYS_MODULE_OBJ object);
  void (* DRV_MIIM_Tasks)(SYS_MODULE_OBJ object);
  DRV_HANDLE (* DRV_MIIM_Open)(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
  DRV_MIIM_RESULT (* DRV_MIIM_Setup)(DRV_HANDLE handle, const DRV_MIIM_SETUP* pSetUp);
  void (* DRV_MIIM_Close)(DRV_HANDLE handle);
```

```
  DRV_MIIM_CLIENT_STATUS (* DRV_MIIM_ClientStatus)(DRV_HANDLE handle);
  DRV_MIIM_CALLBACK_HANDLE (* DRV_MIIM_RegisterCallback)(DRV_HANDLE handle, DRV_MIIM_OPERATION_CALLBACK
cbFunction, DRV_MIIM_RESULT* pRegResult);
  DRV_MIIM_RESULT (* DRV_MIIM_DeregisterCallback)(DRV_HANDLE handle, DRV_MIIM_CALLBACK_HANDLE cbHandle);
  DRV_MIIM_OPERATION_HANDLE (* DRV_MIIM_Read)(DRV_HANDLE handle, unsigned int rIx, unsigned int phyAdd,
DRV_MIIM_OPERATION_FLAGS opFlags, DRV_MIIM_RESULT* pOpResult);
  DRV_MIIM_OPERATION_HANDLE (* DRV_MIIM_Write)(DRV_HANDLE handle, unsigned int rIx, unsigned int phyAdd,
uint16_t wData, DRV_MIIM_OPERATION_FLAGS opFlags, DRV_MIIM_RESULT* pOpResult);
  DRV_MIIM_OPERATION_HANDLE (* DRV_MIIM_Scan)(DRV_HANDLE handle, unsigned int rIx, unsigned int phyAdd,
DRV_MIIM_OPERATION_FLAGS opFlags, DRV_MIIM_RESULT* pOpResult);
  DRV_MIIM_RESULT (* DRV_MIIM_OperationResult)(DRV_HANDLE handle, DRV_MIIM_OPERATION_HANDLE opHandle,
uint16_t* pOpData);
  DRV_MIIM_RESULT (* DRV_MIIM_OperationAbort)(DRV_HANDLE handle, DRV_MIIM_OPERATION_HANDLE opHandle);
};
```

Description

MIIM Driver Base Object

This data structure identifies the required basic interface of the MIIM driver. Any dynamic MIIM driver has to export this interface.

Remarks

This object provides the basic MIIM functionality. Any derived driver can override the basic functionality while maintaining the required interface.

DRV_MIIM_CALLBACK_HANDLE Type

Handle that identifies a client registration operation.

File

drv_miim.h

C

```
typedef const void* DRV_MIIM_CALLBACK_HANDLE;
```

Description

Type: MIIM Callback Registration handle

A handle that a client obtains when calling DRV_MIIM_RegisterCallback. It can be used to deregister the notification callback: DRV_MIIM_DeregisterCallback

Remarks

A valid registration handle is not NULL. An invalid registration handle == 0.

DRV_MIIM_CLIENT_STATUS Enumeration

Defines the possible results of operations that can succeed or fail

File

drv_miim.h

C

```
typedef enum {
  DRV_MIIM_CLIENT_STATUS_ERROR,
  DRV_MIIM_CLIENT_STATUS_READY
} DRV_MIIM_CLIENT_STATUS;
```

Members

Members	Description
DRV_MIIM_CLIENT_STATUS_ERROR	Unspecified error condition. Client does not exist
DRV_MIIM_CLIENT_STATUS_READY	Up and running, can accept operations

Description

MIIM Driver Operation Result *

MIIM Driver Operation Result Codes

This enumeration defines the possible results of any of the MIIM driver operations that have the possibility of failing. This result should be checked to ensure that the operation achieved the desired result.

DRV_MIIM_OPERATION_CALLBACK Type

Notification function that will be called when a MIIM operation is completed and the driver client needs to be notified.

File

drv_miim.h

C

```
typedef void (* DRV_MIIM_OPERATION_CALLBACK)(DRV_HANDLE cliHandle, DRV_MIIM_OPERATION_HANDLE opHandle,
DRV_MIIM_RESULT opResult, uint16_t opData);
```

Description

Type: MIIM Driver Operation Complete Callback

The format of an operation callback notification function registered with the MIIM driver.

Remarks

None.

Parameters

Parameters	Description
cliHandle	the client handle. This is the handle that identifies the client (obtained with DRV_MIIM_Open) that initiated the operation.
opHandle	the operation handle. This is the handle that identifies the operation (obtained with DRV_MIIM_Read, DRV_MIIM_Write, etc.)
opResult	operation result DRV_MIIM_RES_OK if operation completed successfully, otherwise an error code
opData	operation specific data, only if the result is DRV_MIIM_RES_OK For read/scan operation this is the MIIM read data. For write operation this is that data that was written with MIIM.

DRV_MIIM_OPERATION_FLAGS Enumeration

List of flags that apply to a client operation.

File

drv_miim.h

C

```
typedef enum {
  DRV_MIIM_OPERATION_FLAG_NONE,
  DRV_MIIM_OPERATION_FLAG_DISCARD
} DRV_MIIM_OPERATION_FLAGS;
```

Members

Members	Description
DRV_MIIM_OPERATION_FLAG_NONE	No flag specified
DRV_MIIM_OPERATION_FLAG_DISCARD	Upon completion discard the operation result. The client will not poll to check the result nor will need notification This allows dummy operations, discarded as they complete

Description

MIIM Driver Operation flags

This enumeration identifies the operation-specific flags supported by the MIIM driver.

Remarks

Currently only 8 bit flags are supported.

Multiple flags can be simultaneously set.

DRV_MIIM_OPERATION_HANDLE Type

MIIM operation handle.

File

drv_miim.h

C

```
typedef const void* DRV_MIIM_OPERATION_HANDLE;
```

Description

Type: DRV_MIIM_OPERATION_HANDLE

A handle that identifies an operation started by a client. This handle can be used by the client to query the operation status, result, etc. It is also used when the operation complete notification occurs.

Remarks

A valid operation handle is not NULL. An invalid operation handle == 0.

DRV_MIIM_SETUP Structure

Contains all the data necessary to set up the MIIM device.

File

drv_miim.h

C

```
typedef struct {
  uint32_t hostClockFreq;
  uint32_t maxBusFreq;
  DRV_MIIM_SETUP_FLAGS setupFlags;
} DRV_MIIM_SETUP;
```

Members

Members	Description
uint32_t hostClockFreq;	The clock frequency on which this MIIM module operates on, Hz
uint32_t maxBusFreq;	The MIIM bus maximum supported frequency, Hz This is a maximum value. The actual generated value may differ.
DRV_MIIM_SETUP_FLAGS setupFlags;	Setup flags

Description

MIIM Device Driver Set up Data

This data structure contains all the data necessary to configure the MIIM device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_MIIM_Setup routine.

DRV_MIIM_SETUP_FLAGS Enumeration

List of flags that apply to a client setup operation.

File

drv_miim.h

C

```
typedef enum {
  DRV_MIIM_SETUP_FLAG_NONE,
  DRV_MIIM_SETUP_FLAG_PREAMBLE_SUPPRESSED,
  DRV_MIIM_SETUP_FLAG_PREAMBLE_DEFAULT,
  DRV_MIIM_SETUP_FLAG_SCAN_ADDRESS_INCREMENT,
  DRV_MIIM_SETUP_FLAG_SCAN_ADDRESS_DEFAULT
} DRV_MIIM_SETUP_FLAGS;
```

Members

Members	Description
DRV_MIIM_SETUP_FLAG_NONE	No flag specified. Default value

DRV_MIIM_SETUP_FLAG_PREAMBLE_SUPPRESSED	Suppress the normal 32 bit MIIM preamble field. Some PHYs support suppressed preamble
DRV_MIIM_SETUP_FLAG_PREAMBLE_DEFAULT	Include the 32 bit MIIM preamble field. Default operation.
DRV_MIIM_SETUP_FLAG_SCAN_ADDRESS_INCREMENT	Scan operation will read across a range of PHY addresses Scan will start with address 1 through the address set in the scan operation
DRV_MIIM_SETUP_FLAG_SCAN_ADDRESS_DEFAULT	Scan operation will read just one PHY address. Default operation.

Description

MIIM Driver Set up flags

This enumeration identifies the setup specific flags supported by the MIIM driver.

Remarks

Multiple flags can be simultaneously set.

DRV_MIIM_OBJECT_BASE_Default Variable

File

drv_miim.h

C

```
const DRV_MIIM_OBJECT_BASE DRV_MIIM_OBJECT_BASE_Default;
```

Description

The supported basic MIIM driver (DRV_MIIM_OBJECT_BASE). This object is implemented by default as using the standard MIIM interface. It can be overwritten dynamically when needed.

Files

Files

Name	Description
drv_miim.h	MIIM Device Driver Interface File
drv_miim_config.h	MIIM driver configuration definitions template.

Description

This section lists the source and header files used by the Media Interface Independent Management (MIIM)Driver Library.

drv_miim.h

MIIM Device Driver Interface File

Enumerations

	Name	Description
	DRV_MIIM_CLIENT_STATUS	Defines the possible results of operations that can succeed or fail
	DRV_MIIM_OPERATION_FLAGS	List of flags that apply to a client operation.
	DRV_MIIM_SETUP_FLAGS	List of flags that apply to a client setup operation.

Functions

	Name	Description
	DRV_MIIM_ClientStatus	Gets the current client-specific status the MIIM driver.
	DRV_MIIM_Close	Closes an opened instance of the MIIM driver.
	DRV_MIIM_Deinitialize	Deinitializes the specified instance of the MIIM driver module.
	DRV_MIIM_DeregisterCallback	Deregisters an notification callback function for the client operations.
	DRV_MIIM_Initialize	Initializes the MIIM driver.
	DRV_MIIM_Open	Opens the specified MIIM driver instance and returns a handle to it.
	DRV_MIIM_OperationAbort	Aborts a current client operation initiated by the MIIM driver.
	DRV_MIIM_OperationResult	Gets the result of a client operation initiated by the MIIM driver.
	DRV_MIIM_Read	Initiates a SMI/MIIM read transaction.

	DRV_MIIM_RegisterCallback	Registers an notification callback function for the client operations.
	DRV_MIIM_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings.
	DRV_MIIM_Scan	Initiates a SMI/MIIM scan (periodic read)transaction.
	DRV_MIIM_Setup	Sets up a MIIM client.
	DRV_MIIM_Status	Provides the current status of the MIIM driver module.
	DRV_MIIM_Tasks	Maintains the driver's state machine.
	DRV_MIIM_Write	Initiates a SMI/MIIM write transaction.

Macros

	Name	Description
	DRV_MIIM_INDEX_0	MIIM driver index definitions.
	DRV_MIIM_INDEX_COUNT	Number of valid MIIM driver indices.

Structures

	Name	Description
	DRV_MIIM_INIT	Contains all the data necessary to initialize the MIIM device.
	DRV_MIIM_OBJECT_BASE	Declaration of a MIIM base object.
	DRV_MIIM_SETUP	Contains all the data necessary to set up the MIIM device.

Types

	Name	Description
	DRV_MIIM_CALLBACK_HANDLE	Handle that identifies a client registration operation.
	DRV_MIIM_OPERATION_CALLBACK	Notification function that will be called when a MIIM operation is completed and the driver client needs to be notified.
	DRV_MIIM_OPERATION_HANDLE	MIIM operation handle.

Variables

	Name	Description
	DRV_MIIM_OBJECT_BASE_Default	The supported basic MIIM driver (DRV_MIIM_OBJECT_BASE). This object is implemented by default as using the standard MIIM interface. It can be overwritten dynamically when needed.

Description

MIIM Device Driver Interface

The MIIM device driver provides a simple interface to manage an MIIM peripheral using MIIM (SMI)interface. This file defines the interface definitions and prototypes for the MIIM driver.

File Name

drv_miim.h

Company

Microchip Technology Inc.

drv_miim_config.h

MIIM driver configuration definitions template.

Macros

	Name	Description
	_DRV_MIIM_CONFIG_H	This is macro _DRV_MIIM_CONFIG_H.
	DRV_MIIM_CLIENT_OP_PROTECTION	Enables/Disables Client Operation Protection feature.
	DRV_MIIM_COMMANDS	Enables/Disables MIIM commands feature.
	DRV_MIIM_INSTANCE_CLIENTS	Selects the maximum number of clients.
	DRV_MIIM_INSTANCE_OPERATIONS	Selects the maximum number of simultaneous operations for an instance.
	DRV_MIIM_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.

Description

MIIM Driver Configuration Definitions for the Template Version
These definitions statically define the driver's mode of operation.

File Name

drv_miim_config.h

Company

Microchip Technology Inc.

Motor Control PWM (MCPWM) Driver Library

This section describes the MCPWM Driver Library.

Introduction

The MCPWM Static Driver provides a high-level interface to manage the MCPWM module on the Microchip family of microcontrollers.

Description

Through MHC, this driver provides APIs to initialize, enable, and disable the MCPWM module.

Library Interface

Function(s)

	Name	Description
	DRV_MCPWM_Disable	Disables the MCPWM instance for the specified driver index. **Implementation:** Static
	DRV_MCPWM_Enable	Enables the MCPWM instance for the specified driver index. **Implementation:** Static
	DRV_MCPWM_Initialize	Initializes the MCPWM instance for the specified driver index. **Implementation:** Static

Description

This section describes the Application Programming Interface (API) functions of the MCPWM Driver Library.

Function(s)

DRV_MCPWM_Disable Function

Disables the MCPWM instance for the specified driver index.

Implementation: Static

File

drv_mcpwm.h

C

```
void DRV_MCPWM_Disable();
```

Returns

None.

Description

This routine disables the MCPWM Driver instance for the specified driver instance.

Preconditions

DRV_MCPWM_Initialize has been called.

Function

void DRV_MCPWM_Disable(void)

DRV_MCPWM_Enable Function

Enables the MCPWM instance for the specified driver index.

Implementation: Static

File

drv_mcpwm.h

C

```
void DRV_MCPWM_Enable();
```

Returns

None.

Description

This routine enables the MCPWM Driver instance for the specified driver instance, making it ready for clients to use it. The enable routine is specified by the MHC parameters.

Preconditions

DRV_MCPWM_Initialize has been called.

Function

void DRV_MCPWM_Enable(void)

DRV_MCPWM_Initialize Function

Initializes the MCPWM instance for the specified driver index.

Implementation: Static

File

drv_mcpwm.h

C

```
void DRV_MCPWM_Initialize();
```

Returns

None.

Description

This routine initializes the MCPWM Driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Remarks

This routine must be called before any other MCPWM routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Function

void DRV_MCPWM_Initialize(void)

Files

Files

Name	Description
drv_mcpwm.h	MCPWM driver interface declarations for the static single instance driver.

Description

drv_mcpwm.h

MCPWM driver interface declarations for the static single instance driver.

Functions

	Name	Description
	DRV_MCPWM_Disable	Disables the MCPWM instance for the specified driver index. **Implementation:** Static
	DRV_MCPWM_Enable	Enables the MCPWM instance for the specified driver index. **Implementation:** Static
	DRV_MCPWM_Initialize	Initializes the MCPWM instance for the specified driver index. **Implementation:** Static

Description

Motor Control PWM (MCPWM) Driver Interface Declarations for Static Single Instance Driver

The MCPWM device driver provides a simple interface to manage the MCPWM module on Microchip microcontrollers. This file defines the interface Declarations for the MCPWM driver.

Remarks

Static interfaces incorporate the driver instance number within the names of the routines, eliminating the need for an object ID or object handle.

Static single-open interfaces also eliminate the need for the open handle.

File Name

drv_mcpwm.h

Company

Microchip Technology Inc.

NVM Driver Library

This section describes the Non-volatile Memory (NVM) Driver Library.

Migrating Applications from v1.03.01 and Earlier Releases of MPLAB Harmony

Provides information on migrating applications from v1.03.01 and earlier releases of MPLAB Harmony to release v1.04 and later.

Description

The NVM Driver Library APIs have changed beginning with the v1.04 release of MPLAB Harmony. Applications that were developed using the earlier version of the MPLAB Harmony NVM Driver (v1.03.01 and earlier) will not build unless the application calls to NVM Driver are updated. While the MHC utility provides an option to continue creating applications using the v1.03.01 and earlier NVM Driver API, it is recommended that existing applications migrate to the latest API to take advantage of the latest features in the NVM Driver. The following sections describe the API changes and other considerations while updating the application for changes in the NVM Driver.

All NVM Driver Demonstration Applications and NVM Driver related documentation have been updated to the latest (new) API. The following sections do not discuss changes in the NVM Driver configuration related code. This code is updated automatically when the project is regenerated using the MHC utility. Only the application related API changes are discussed.

The following table shows the beta API and corresponding v1.04 and Later MPLAB Harmony NVM Driver API.

v1.03.01 and Earlier NVM Driver API	v1.04 and Later NVM Driver API	v1.04 and Later API Notes
DRV_NVM_Initialize	DRV_NVM_Initialize	The init structure now has additional members that allow the NVM media address and geometry to be specified.
DRV_NVM_Deinitialize	DRV_NVM_Deinitialize	No change.
DRV_NVM_Status	DRV_NVM_Status	No change.
DRV_NVM_Open	DRV_NVM_Open	No change.
DRV_NVM_Close	DRV_NVM_Close	No change.

DRV_NVM_Read	DRV_NVM_Read	Parameters have changed: • Returns the command handle associated with the read operation as an output parameter • Data is now read in terms of blocks. The read block size is specified in the NVM Geometry.
DRV_NVM_Write	DRV_NVM_Write	Parameters have changed: • Returns the command handle associated with the write operation as an output parameter • Data is now written in terms of blocks. The write block size is specified in the NVM Geometry.
DRV_NVM_Erase	DRV_NVM_Erase	Parameters have changed: • Returns the command handle associated with the erase operation as an output parameter • NVM Flash is erased in terms of blocks. The erase block size is specified in the NVM Geometry.
DRV_NVM_EraseWrite	DRV_NVM_EraseWrite	Parameters have changed: • Returns the command handle associated with the Erase/Write operation as an output parameter. • Data is now written in terms of blocks. The write block size is specified in the NVM Geometry.
DRV_NVM_BlockEventHandlerSet	DRV_NVM_EventHandlerSet	Function name and parameter type have changed.
DRV_NVM_ClientStatus	Not Available	This API is no longer available.
DRV_NVM_BufferStatus	DRV_NVM_CommandStatus	The DRV_NVM_Read, DRV_NVM_Write, DRV_NVM_Erase, and DRV_NVM_EraseWrite functions now return a command handle associated with the operation. The status of the operation can be checked by passing the command handle to this function.
Not Available	DRV_NVM_GeometryGet	This API gives the following geometrical details of the NVM Flash: • Media Property • Number of Read/Write/Erase regions in the flash device • Number of Blocks and their size in each region of the device
Not Available	DRV_NVM_IsAttached	Returns the physical attach status of the NVM Flash.
Not Available	DRV_NVM_IsWriteProtected	Returns the write protect status of the NVM Flash.
Not Available	DRV_NVM_AddressGet	Returns the NVM Media Start address.

NVM Driver Initialization

DRV_NVM_INIT now takes the following two additional initialization parameters:

- mediaStartAddress - NVM Media Start address. The driver treats this address as the start address for read, write and erase operations.
- nvmMediaGeometry - Indicates the layout of the media in terms of read, write and erase regions.

The following code examples show how the driver initialization was performed with 1.03 APIs and how it is performed with the 1.04 APIs:

Example 1: v1.03 and Earlier Code

```
const DRV_NVM_INIT drvNvmInit =
{
    .moduleInit.sys.powerState = SYS_MODULE_POWER_RUN_FULL,
    .nvmID = NVM_ID_0,
    .interruptSource = INT_SOURCE_FLASH_CONTROL,
};

void SYS_Initialize (void *data)
{
.
.
    // Initialize NVM Driver Layer
    sysObj.drvNvm = DRV_NVM_Initialize(DRV_NVM_INDEX_0, (SYS_MODULE_INIT *)&drvNvmInit);
.
}
```

Example: v1.04 and Later Code

```
/* NVM Geometry structure */
SYS_FS_MEDIA_REGION_GEOMETRY NVMGeometryTable[3] =
{
    {
        .blockSize = 1,
```

```
            .numBlocks = (DRV_NVM_MEDIA_SIZE * 1024),
        },
        {
           .blockSize = DRV_NVM_ROW_SIZE,
           .numBlocks = ((DRV_NVM_MEDIA_SIZE * 1024)/DRV_NVM_ROW_SIZE)
        },
        {
           .blockSize = DRV_NVM_PAGE_SIZE,
           .numBlocks = ((DRV_NVM_MEDIA_SIZE * 1024)/DRV_NVM_PAGE_SIZE)
        }
    };

    const SYS_FS_MEDIA_GEOMETRY NVMGeometry =
    {
        .mediaProperty = SYS_FS_MEDIA_WRITE_IS_BLOCKING,
        .numReadRegions = 1,
        .numWriteRegions = 1,
        .numEraseRegions = 1,
        .geometryTable = (SYS_FS_MEDIA_REGION_GEOMETRY *)&NVMGeometryTable
    };

    const DRV_NVM_INIT drvNvmInit =
    {
        .moduleInit.sys.powerState = SYS_MODULE_POWER_RUN_FULL,
        .nvmID = NVM_ID_0,
        .interruptSource = INT_SOURCE_FLASH_CONTROL,
        .mediaStartAddress = 0x9D010000,
        .nvmMediaGeometry = (SYS_FS_MEDIA_GEOMETRY *)&NVMGeometry
    };

    void SYS_Initialize (void *data)
    {
    .
    .
        // Initialize NVM Driver Layer
        sysObj.drvNvm = DRV_NVM_Initialize(DRV_NVM_INDEX_0, (SYS_MODULE_INIT *)&drvNvmInit);
    .
    .
    }
```

Addressing in NVM Driver

The v1.03.01 and earlier Read, Write, Erase and EraseWrite APIs took the actual address on which the operation was to be performed. The unit of access was bytes.

In v1.04 the addressing mechanism has been modified. The media start address is set in the DRV_NVM_Initialize. This address is used as the base address for the Read, Write, Erase and EraseWrite APIs. The unit of access is in terms of blocks. The NVM Geometry specifies the media layout in terms of:

- Number of erase, read and write regions
- Block size for erase, read and write operations.
- Number of blocks in erase, read and write regions

For example, in PIC32MZ family devices:

- Read block size = 1 byte
- Write block size = ROW Size = 2048 bytes
- Erase block size = PAGE Size = 16384 bytes

If the size of media is 32 KB then the following table illustrates the address range and number of blocks for the read, write and erase regions:

Region Type	Block Size	Number of blocks	Address range
Read Region	1 Byte	32 KB / Read block size = 32768	0–32767
Write Region	2048 Bytes	32 KB / Write block size = 16 blocks	0–15
Erase Region	16384 Bytes	32 KB / Erase block size = 2 blocks	0–1

Erasing Data on NVM Flash

The NVM Geometry indicates the number of erase blocks and the size of a single erase block. The Erase API takes in the erase block start address and the number of blocks to be erased. The following code examples show how to perform the erase operation in v1.03.01 and earlier and how to perform it with v1.04 and later.

Example: v**1.03.01 and Earlier Code**

```
DRV_HANDLE      myNVMHandle;    // Returned from DRV_NVM_Open
DRV_NVM_BUFFER_HANDLE bufferHandle;

bufferHandle = DRV_NVM_Erase(myNVMHandle, (uint8_t*)NVM_BASE_ADDRESS, DRV_NVM_PAGE_SIZE);
if(DRV_NVM_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Do error handling here
}

// Wait until the buffer completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.
while(DRV_NVM_BufferStatus(bufferHandle) != DRV_NVM_BUFFER_COMPLETED);
```

Example: v1.04 and Later Code

```
/* This code example shows how to erase NVM Media data */
DRV_HANDLE nvmHandle;
DRV_NVM_COMMAND_HANDLE nvmCommandHandle;
DRV_NVM_COMMAND_STATUS commandStatus;
uint32_t blockAddress;
uint32_t nBlocks;

blockAddress = 0;
nBlocks = 1;

DRV_NVM_Erase(nvmHandle, &nvmCommandHandle, blockAddress, nBlocks);
if(DRV_NVM_COMMAND_HANDLE_INVALID == nvmCommandHandle)
{
    /* Failed to queue the erase request. Handle the error. */
}
// Wait until the command completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.

commandStatus = DRV_NVM_CommandStatus(nvmHandle, nvmCommandHandle);
if(DRV_NVM_COMMAND_COMPLETED == commandStatus)
{
    /* Erase completed */
}
else if (DRV_NVM_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Erase Failed */
}
```

Writing Data to NVM Flash

The NVM Geometry indicates the number of write blocks and the size of a single write block. The Write API takes in the write block start address and the number of blocks to be written. The following code examples show how the write operation was performed in v1.03.01 and earlier and how to perform it with v1.04 and later APIs:

Example : v1.03.01 and Earlier Code

```
DRV_HANDLE myNVMHandle;    // Returned from DRV_NVM_Open
char       myBuffer[2 * DRV_NVM_ROW_SIZE];

// Destination address should be row aligned.
char       *destAddress = (char *)NVM_BASE_ADDRESS_TO_WRITE;

unsigned int    count = 2 * MY_BUFFER_SIZE;
DRV_NVM_BUFFER_HANDLE bufferHandle;

bufferHandle  = DRV_NVM_Write(myNVMHandle, destAddress, &myBuffer[total], count);
if(DRV_NVM_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Do error handling here
}

// Wait until the buffer completes. This should not
// be a while loop if a part of cooperative multi-tasking
```

```
// routine. In that case, it should be invoked in task
// state machine.
while(DRV_NVM_BufferStatus(bufferHandle) != DRV_NVM_BUFFER_COMPLETED);
```

Example: v1.04 and Later Code

```
/* This code example shows how to write data to NVM Media */
DRV_HANDLE nvmHandle;
DRV_NVM_COMMAND_HANDLE nvmCommandHandle;
DRV_NVM_COMMAND_STATUS commandStatus;
uint8_t writeBuf[DRV_NVM_ROW_SIZE];
uint32_t blockAddress;
uint32_t nBlocks;

blockAddress = 0;
nBlocks = 1;

DRV_NVM_Write(nvmHandle, &nvmCommandHandle, (uint8_t *)writeBuf, blockAddress, nBlocks);
if(DRV_NVM_COMMAND_HANDLE_INVALID == nvmCommandHandle)
{
    /* Failed to queue the write request. Handle the error. */
}
// Wait until the command completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.

commandStatus = DRV_NVM_CommandStatus(nvmHandle, nvmCommandHandle);
if(DRV_NVM_COMMAND_COMPLETED == commandStatus)
{
    /* Write completed */
}
else if (DRV_NVM_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Write Failed */
}
```

Reading Data from NVM Flash

The NVM Geometry indicates the number of read blocks and the size of a single read block. The Read API takes in the read block start address and the number of blocks to be read. The following code examples show how the read operation was performed with v1.03.01 and earlier APIs and how to perform the same with v1.04 and later APIs:

Example: v1.03.01 and Earlier Code

```
DRV_HANDLE      myNVMHandle;    // Returned from DRV_NVM_Open
char            myBuffer[MY_BUFFER_SIZE];
char            *srcAddress = NVM_BASE_ADDRESS_TO_READ_FROM;
unsigned int    count = MY_BUFFER_SIZE;
DRV_NVM_BUFFER_HANDLE bufferHandle;

bufferHandle  = DRV_NVM_Read(myNVMHandle, &myBuffer[total], srcAddress, count);
if(DRV_NVM_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Do error handling here
}

// Wait until the buffer completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.
while(DRV_NVM_BufferStatus(bufferHandle) != DRV_NVM_BUFFER_COMPLETED);
```

Example: v1.04 and Later Code

```
/* This code example shows how to read data from NVM Media */
DRV_HANDLE nvmHandle;
DRV_NVM_COMMAND_HANDLE nvmCommandHandle;
DRV_NVM_COMMAND_STATUS commandStatus;
uint8_t readBuf[DRV_NVM_ROW_SIZE];
uint32_t blockAddress;
uint32_t nBlocks;

blockAddress = 0;
```

```
nBlocks = DRV_NVM_ROW_SIZE;

DRV_NVM_Read(nvmHandle, &nvmCommandHandle, (uint8_t *)readBuf, blockAddress, nBlocks);
if(DRV_NVM_COMMAND_HANDLE_INVALID == nvmCommandHandle)
{
    /* Failed to queue the read request. Handle the error. */
}
// Wait until the command completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.

commandStatus = DRV_NVM_CommandStatus(nvmHandle, nvmCommandHandle);
if(DRV_NVM_COMMAND_COMPLETED == commandStatus)
{
    /* Read completed */
}
else if (DRV_NVM_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Read Failed */
}
```

Introduction

The NVM Driver library provides APIs that can be used to interface with the NVM module (controller plus memory) for memory needs.

Description

The NVM Driver provides APIs for block access of the physical media through NVM Driver APIs. As shown in the NVM Driver Abstraction Model, an application or a client can access the physical media using multiple methods, which eventually are facilitated through the NVM Driver.

Memory Devices for PIC Microcontrollers

Depending on the device, there are two primary forms of on-chip memory: Programmable Flash memory and data EEPROM memory. The access mechanism for both of these types are varied.

Flash Program Memory

The Flash program memory is readable, writeable, and erasable during normal operation over the entire operating voltage range.

A read from program memory is executed at one byte/word at a time depending on the width of the data bus.

A write to the program memory is executed in either blocks of specific sizes or a single word depending on the type of processor used.

An erase is performed in blocks. A bulk erase may be performed from user code depending on the type of processor supporting the operation.

Writing or erasing program memory will cease instruction fetches until the operation is complete, restricting memory access, and therefore preventing code execution. This is controlled by an internal programming timer.

There are three processor dependant methods for program memory modification:

- Run-Time Self-Programming (RTSP)
- In-Circuit Serial Programming (ICSP)
- EJTAG programming

This section describes the RTSP techniques.

Using the Library

This topic describes the basic architecture of the NVM Driver Library and provides information and examples on its use.

Description

Interface Header Files: `drv_nvm.h`

The interface to the NVM Driver Library is defined in the `drv_nvm.h` header file. Any C language source (`.c`) file that uses the NVM Driver library should include `drv_nvm.h`.

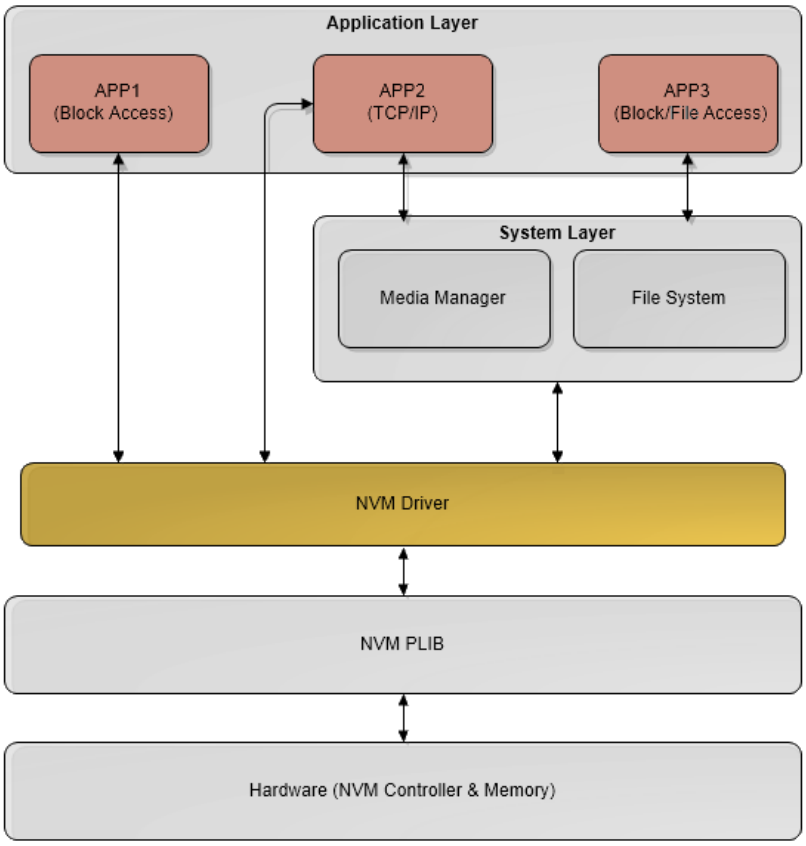
Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the NVM module on the Microchip family of microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

NVM Driver Abstraction Model



Abstraction Model

As shown in the previous diagram, the NVM Driver sits between the Peripheral Libraries and the application or system layer to facilitate block and file access to the NVM media (currently Flash). The application scenarios show how different layers can be accessed by different applications with certain needs. For example, APP1 can access the NVM Driver directly to erase, write, or read NVM with direct addressing. APP2, in this case TCP/IP, can bypass the system layer and access the NVM Driver layer if necessary to fulfill its robust data needs. Finally, APP3 accesses the NVM Driver through the File System Layer using block access methods, so the application does not need to keep track of the physical layout of the media.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the NVM module.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Core Functions	Provides open, close, status and other setup functions.
Client Block Transfer Functions	Provides buffered data operation functions available in the core configuration.
Miscellaneous Functions	Provides driver miscellaneous functions related to versions and others.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality
- Media Functionality

Note: Not all modes are available on all devices. Please refer to the specific device data sheet to determine the modes supported for your device.

NVM System Initialization

This section provides information for system initialization and reinitialization.

Description

The system performs the initialization and the reinitialization of the device driver with settings that affect only the instance of the device that is being initialized or reinitialized. During system initialization each instance of the NVM module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_NVM_INIT or by using initialization overrides) that are supported by the specific NVM device hardware:

- Device requested power state: One of the system module power states. For specific details please refer to **Data Types and Constants** in the Library Interface section.
- The actual peripheral ID enumerated as the PLIB level module ID (e.g., NVM_ID_0)
- Defining the respective interrupt sources
- NVM Media Start Address
- NVM Media Geometry

The DRV_NVM_Initialize function returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the initialize interface would be used by the other system interfaces, such as DRV_NVM_Deinitialize, DRV_NVM_Status, and DRV_NVM_Tasks.

Note: The system initialization and the reinitialization settings, only affect the instance of the peripheral that is being initialized or reinitialized.

The SYS_MODULE_INDEX is passed to the DRV_NVM_Initialize function to determine which type of memory is selected using:
DRV_NVM_INDEX_0 - FLASH

Example:

```
const DRV_NVM_INIT drvNvmInit =
{
    .moduleInit.sys.powerState = SYS_MODULE_POWER_RUN_FULL,
    .nvmID = NVM_ID_0,
    .interruptSource = INT_SOURCE_FLASH_CONTROL,
    .mediaStartAddress = 0x9D010000,
    .nvmMediaGeometry = (SYS_FS_MEDIA_GEOMETRY *)&NVMGeometry
};
void SYS_Initialize (void *data)
{
.
.
.

   // Initialize NVM Driver Layer
     sysObj.drvNvm = DRV_NVM_Initialize(DRV_NVM_INDEX_0, (SYS_MODULE_INIT *)&drvNvmInit);
.
.
.
}
```

Tasks Routine

The system will call DRV_NVM_Tasks, from system task service (in a polled environment) or DRV_NVM_Tasks will be called from the Interrupt Service Routine (ISR) of the NVM.

Client Access Operation

This section provides information for general client operation.

Description

General Client Operation

For the application to start using an instance of the module, it must call the DRV_NVM_Open function. This provides the configuration required to open the NVM instance for operation. If the driver is deinitialized using the function DRV_NVM_Deinitialize, the application must call the DRV_NVM_Open function again to set up the instance of the NVM.

For the various options available for I/O INTENT please refer to **Data Types and Constants** in the Library Interface section.

Example:

```
DRV_HANDLE handle;

handle = DRV_NVM_Open(DRV_NVM_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Client Block Data Operation

This topic provides information on client block data operation.

Description

The NVM Driver provides a block interface to access the NVM media. The interface provides functionality to read, write, erase, and erase-write the NVM media. These interface functions depend on the block sizes and boundaries of the individual devices. The interfaces are responsible for keeping this information transparent from the application.

Erasing Data on the NVM:

The following steps outline the sequence for erasing data on the NVM media:

1. The system should have completed necessary initialization and DRV_NVM_Tasks should either be running in a polled environment, or in an interrupt environment.
2. The driver should have been opened with the necessary intent.
3. Provide the block start address and the number of blocks to be erased and begin the erase process using the DRV_NVM_Erase.
4. The client can check the state of the erase request by invoking the DRV_NVM_CommandStatus and passing the command handle returned by the erase request.
5. The client will be able to close itself by calling the DRV_NVM_Close.

Example:

```
// This code shows how to erase NVM Media data
DRV_HANDLE              nvmHandle;
DRV_NVM_COMMAND_HANDLE nvmCommandHandle;

DRV_NVM_COMMAND_STATUS commandStatus;

uint32_t blockAddress;
uint32_t nBlocks;

blockAddress = 0;
nBlocks = 1;

DRV_NVM_Erase(nvmHandle, &nvmCommandHandle, blockAddress, nBlocks);

if(DRV_NVM_COMMAND_HANDLE_INVALID == nvmCommandHandle)
{
    /* Failed to queue the erase request. Handle the error. */
}

// Wait until the command completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.
commandStatus = DRV_NVM_CommandStatus(nvmHandle, nvmCommandHandle);
if(DRV_NVM_COMMAND_COMPLETED == commandStatus)
{
    /* Erase completed */
}
else if (DRV_NVM_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Erase Failed */
}
```

Writing Data to the NVM:

The following steps outline the sequence to be followed for writing data to the NVM Media:

1. The system should have completed necessary initialization and DRV_NVM_Tasks should either be running in a polled environment, or in an interrupt environment.

2. The driver should have been opened with the necessary intent.
3. The client should ensure that blocks of addresses to which write is being performed should be in the erased state.
4. Provide the data to be written, block start address and the number of blocks to be written and begin write using the DRV_NVM_Write.
5. The client can check the state of the write request by invoking the DRV_NVM_CommandStatus and passing the command handle returned by the write request.
6. The client will be able to close itself by calling the DRV_NVM_Close.

Example:

```
// This code shows how to write data to NVM Media
DRV_HANDLE              nvmHandle;
DRV_NVM_COMMAND_HANDLE nvmCommandHandle;

DRV_NVM_COMMAND_STATUS commandStatus;

uint8_t writeBuf[DRV_NVM_ROW_SIZE];
uint32_t blockAddress;
uint32_t nBlocks;

blockAddress = 0;
nBlocks = 1;

DRV_NVM_Write(nvmHandle, &nvmCommandHandle, (uint8_t *)writeBuf, blockAddress, nBlocks);

if(DRV_NVM_COMMAND_HANDLE_INVALID == nvmCommandHandle)
{
    /* Failed to queue the write request. Handle the error. */
}

// Wait until the command completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.
commandStatus = DRV_NVM_CommandStatus(nvmHandle, nvmCommandHandle);
if(DRV_NVM_COMMAND_COMPLETED == commandStatus)
{
    /* Write completed */
}
else if (DRV_NVM_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Write Failed */
}
```

Reading Data from the NVM:

The following steps outline the sequence to be followed for reading data from the NVM Media:

1. The system should have completed necessary initialization and DRV_NVM_Tasks should either be running in a polled environment, or in an interrupt environment.
2. The driver should have been opened with the necessary intent.
3. Provide the target buffer, block start address and the number of blocks to be read and begin reading using the DRV_NVM_Read.
4. The client can check the state of the read request by invoking the DRV_NVM_CommandStatus and passing the command handle returned by the read request.
5. The client will be able to close itself by calling the DRV_NVM_Close.

Example:

```
// This code shows how to read data from NVM Media
DRV_HANDLE              nvmHandle;
DRV_NVM_COMMAND_HANDLE nvmCommandHandle;

DRV_NVM_COMMAND_STATUS commandStatus;

uint8_t readBuf[DRV_NVM_ROW_SIZE];
uint32_t blockAddress;
uint32_t nBlocks;

blockAddress = 0;
nBlocks = DRV_NVM_ROW_SIZE;

DRV_NVM_Read(nvmHandle, &nvmCommandHandle, (uint8_t *)readBuf, blockAddress, nBlocks);
```

```
if(DRV_NVM_COMMAND_HANDLE_INVALID == nvmCommandHandle)
{
    /* Failed to queue the read request. Handle the error. */
}

// Wait until the command completes. This should not
// be a while loop if a part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.

commandStatus = DRV_NVM_CommandStatus(nvmHandle, nvmCommandHandle);
if(DRV_NVM_COMMAND_COMPLETED == commandStatus)
{
    /* Read completed */
}
else if (DRV_NVM_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Read Failed */
}
```

Configuring the Library

Macros

	Name	Description
	DRV_NVM_BUFFER_OBJECT_NUMBER	Selects the maximum number of buffer objects
	DRV_NVM_CLIENTS_NUMBER	Selects the maximum number of clients
	DRV_NVM_INSTANCES_NUMBER	Selects the maximum number of Driver instances that can be supported by the dynamic driver.
	DRV_NVM_INTERRUPT_MODE	Macro specifies operation of the driver to be in the interrupt mode or polled mode
	DRV_NVM_ROW_SIZE	Specifies the NVM Driver Program Row Size in bytes.
	DRV_NVM_ERASE_WRITE_ENABLE	Enables support for NVM Driver Erase Write Feature.
	DRV_NVM_PAGE_SIZE	Specifies the NVM Driver Program Page Size in bytes.
	DRV_NVM_DISABLE_ERROR_CHECK	Disables the error checks in the driver.
	DRV_NVM_MEDIA_SIZE	Specifies the NVM Media size.
	DRV_NVM_MEDIA_START_ADDRESS	Specifies the NVM Media start address.
	DRV_NVM_SYS_FS_REGISTER	Register to use with the File system

Description

The configuration of the NVM Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the NVM Driver. Based on the selections made, the NVM Driver may support the selected features. These configuration settings will apply to all instances of the NVM Driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_NVM_BUFFER_OBJECT_NUMBER Macro

Selects the maximum number of buffer objects

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_BUFFER_OBJECT_NUMBER 5
```

Description

NVM Driver maximum number of buffer objects

This definition selects the maximum number of buffer objects. This indirectly also specifies the queue depth. The NVM Driver can queue up DRV_NVM_BUFFER_OBJECT_NUMBER of read/write/erase requests before return a DRV_NVM_BUFFER_HANDLE_INVALID due to the queue being full. Buffer objects are shared by all instances of the driver. Increasing this number increases the RAM requirement of the driver.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_CLIENTS_NUMBER Macro

Selects the maximum number of clients

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_CLIENTS_NUMBER 1
```

Description

NVM maximum number of clients

This definition selects the maximum number of clients that the NVM driver can supported at run time. This constant defines the total number of NVM driver clients that will be available to all instances of the NVM driver.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_INSTANCES_NUMBER Macro

Selects the maximum number of Driver instances that can be supported by the dynamic driver.

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_INSTANCES_NUMBER 1
```

Description

NVM Driver instance configuration

This definition selects the maximum number of Driver instances that can be supported by the dynamic driver. In case of this driver, multiple instances of the driver could use the same hardware instance.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_INTERRUPT_MODE Macro

Macro specifies operation of the driver to be in the interrupt mode or polled mode

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_INTERRUPT_MODE true
```

Description

NVM interrupt and polled mode operation control

This macro specifies operation of the driver to be in the interrupt mode or polled mode

- true - Select if interrupt mode of NVM operation is desired
- false - Select if polling mode of NVM operation is desired

Not defining this option to true or false will result in build error.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_ROW_SIZE Macro

Specifies the NVM Driver Program Row Size in bytes.

File

drv_nvm.h

C

```
#define DRV_NVM_ROW_SIZE (NVM_ROW_SIZE)
```

Description

NVM Driver Program Row Size.

This definition specifies the NVM Driver Program Row Size in bytes. This parameter is device specific and is obtained from the device specific processor header file. The Program Row Size is the minimum block size that can be programmed in one program operation.

Remarks

None

DRV_NVM_ERASE_WRITE_ENABLE Macro

Enables support for NVM Driver Erase Write Feature.

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_ERASE_WRITE_ENABLE
```

Description

NVM Driver Erase Write Feature Enable

Specifying this macro enable row erase write feature. If this macro is specified, the drv_nvm_erasewrite.c file should be added in the project. Support for DRV_NVM_EraseWrite() function then gets enabled.

Remarks

This macro is optional and should be specified only if the DRV_NVM_EraseWrite() function is required.

DRV_NVM_PAGE_SIZE Macro

Specifies the NVM Driver Program Page Size in bytes.

File

drv_nvm.h

C

```
#define DRV_NVM_PAGE_SIZE (NVM_PAGE_SIZE)
```

Description

NVM Driver Program Page Size.

This definition specifies the NVM Driver Program Page Size in bytes. This parameter is device specific and is obtained from the device specific processor header file.

Remarks

None

DRV_NVM_DISABLE_ERROR_CHECK Macro

Disables the error checks in the driver.

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_DISABLE_ERROR_CHECK
```

Description

NVM Driver Disable Error Checks

Specifying this macro disables the error checks in the driver. Error checks like parameter validation, NULL checks etc, will be disabled in the driver in order to optimize the code space.

Remarks

This macro is optional and should be specified only if code space is a constraint.

DRV_NVM_MEDIA_SIZE Macro

Specifies the NVM Media size.

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_MEDIA_SIZE 32
```

Description

NVM Media Size

This definition specifies the NVM Media Size to be used. The size is specified in number of Kilo Bytes. The media size MUST never exceed physical available NVM Memory size. Application code requirements should be kept in mind while defining this parameter.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_MEDIA_START_ADDRESS Macro

Specifies the NVM Media start address.

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_MEDIA_START_ADDRESS 0x9D010000
```

Description

NVM Media Start Address

This definition specifies the NVM Media Start address parameter.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_NVM_SYS_FS_REGISTER Macro

Register to use with the File system

File

drv_nvm_config_template.h

C

```
#define DRV_NVM_SYS_FS_REGISTER
```

Description

NVM Driver Register with File System

Specifying this macro enables the NVM driver to register its services with the SYS FS.

Remarks

This macro is optional and should be specified only if the NVM driver is to be used with the File System.

Building the Library

This section lists the files that are available in the NVM Driver Library.

Description

This section list the files that are available in the `\src` folder of the NVM Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/nvm`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_nvm.h`	Header file that exports the driver API.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_nvm.c` `/src/dynamic/drv_nvm_erasewrite.c`	Dynamic NVM Driver implementation file. Dynamic NVM Driver Erase/Write implementation file.
`/src/static/drv_nvm_static.c`	Static NVM Driver implementation file for single clients.
`/src/static_multi/drv_nvm_static_multi.c`	Static NVM Driver implementation file for multiple clients.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The NVM Driver Library depends on the following modules:

- Interrupt System Service Library
- Ports System Service Library

Library Interface

a) System Functions

	Name	Description
	DRV_NVM_Initialize	Initializes the NVM instance for the specified driver index **Implementation:** Static/Dynamic
	DRV_NVM_Deinitialize	Deinitializes the specified instance of the NVM driver module **Implementation:** Static/Dynamic
	DRV_NVM_Status	Gets the current status of the NVM driver module. **Implementation:** Static/Dynamic

b) Client Core Functions

	Name	Description
	DRV_NVM_Open	Opens the specified NVM driver instance and returns a handle to it **Implementation:** Static/Dynamic
	DRV_NVM_Close	Closes an opened-instance of the NVM driver **Implementation:** Static/Dynamic

	DRV_NVM_Read	Reads blocks of data from the specified address in memory. **Implementation:** Static/Dynamic
	DRV_NVM_Write	Writes blocks of data starting from the specified address in flash memory. **Implementation:** Static/Dynamic
	DRV_NVM_Erase	Erase the specified number of blocks of the Flash memory. **Implementation:** Static/Dynamic
	DRV_NVM_EraseWrite	Erase and Write blocks of data starting from a specified address in flash memory. **Implementation:** Static/Dynamic
	DRV_NVM_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed. **Implementation:** Static/Dynamic

c) Client Block Data Functions

	Name	Description
	DRV_NVM_Tasks	Maintains the driver's erase and write state machine and implements its ISR. **Implementation:** Static/Dynamic

d) Status Functions

	Name	Description
	DRV_NVM_AddressGet	Returns the NVM media start address **Implementation:** Static/Dynamic
	DRV_NVM_CommandStatus	Gets the current status of the command. **Implementation:** Static/Dynamic
	DRV_NVM_GeometryGet	Returns the geometry of the device. **Implementation:** Static/Dynamic

e) Miscellaneous Functions

	Name	Description
	DRV_NVM_IsAttached	Returns the physical attach status of the NVM. **Implementation:** Static/Dynamic
	DRV_NVM_IsWriteProtected	Returns the write protect status of the NVM. **Implementation:** Static/Dynamic

f) Data Types and Constants

	Name	Description
	DRV_NVM_INDEX_0	NVM driver index definitions
	DRV_NVM_INIT	Defines the data required to initialize or reinitialize the NVM driver
	DRV_NVM_INDEX_1	This is macro DRV_NVM_INDEX_1.
	DRV_NVM_EVENT	Identifies the possible events that can result from a request.
	DRV_NVM_EVENT_HANDLER	Pointer to a NVM Driver Event handler function
	DRV_NVM_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_NVM_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.
	DRV_NVM_COMMAND_HANDLE_INVALID	This value defines the NVM Driver's Invalid Command Handle.

Description

This section describes the Application Programming Interface (API) functions of the NVM Driver Library.

Refer to each section for a detailed description.

a) System Functions

DRV_NVM_Initialize Function

Initializes the NVM instance for the specified driver index

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
SYS_MODULE_OBJ DRV_NVM_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the NVM driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This routine must be called before any other NVM routine is called.

This routine should only be called once during system initialization unless DRV_NVM_Deinitialize is called to deinitialize the driver instance.

This routine will NEVER block for hardware access. If the operation requires time to allow the hardware to reinitialize, it will be reported by the DRV_NVM_Status operation. The system must use DRV_NVM_Status to find out when the driver is in the ready state.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this routine.

Preconditions

None.

Example

```
// This code snippet shows an example
// of initializing the NVM Driver.

SYS_MODULE_OBJ  objectHandle;

SYS_FS_MEDIA_REGION_GEOMETRY gNvmGeometryTable[3] =
{
    {
        // Read Region Geometry
        .blockSize = 1,
        .numBlocks = (DRV_NVM_MEDIA_SIZE * 1024),
    },
    {
        // Write Region Geometry
        .blockSize = DRV_NVM_ROW_SIZE,
        .numBlocks = ((DRV_NVM_MEDIA_SIZE * 1024)/DRV_NVM_ROW_SIZE)
    },
    {
        // Erase Region Geometry
        .blockSize = DRV_NVM_PAGE_SIZE,
        .numBlocks = ((DRV_NVM_MEDIA_SIZE * 1024)/DRV_NVM_PAGE_SIZE)
    }
};

const SYS_FS_MEDIA_GEOMETRY gNvmGeometry =
{
    .mediaProperty = SYS_FS_MEDIA_WRITE_IS_BLOCKING,

    // Number of read, write and erase entries in the table
    .numReadRegions = 1,
    .numWriteRegions = 1,
    .numEraseRegions = 1,
    .geometryTable = &gNvmGeometryTable
};

// FLASH Driver Initialization Data
const DRV_NVM_INIT drvNvmInit =
{
    .moduleInit.sys.powerState = SYS_MODULE_POWER_RUN_FULL,
    .nvmID = NVM_ID_0,
    .interruptSource = INT_SOURCE_FLASH_CONTROL,
    .mediaStartAddress = NVM_BASE_ADDRESS,
    .nvmMediaGeometry = &gNvmGeometry
};
```

```
//usage of DRV_NVM_INDEX_0 indicates usage of Flash-related APIs
objectHandle = DRV_NVM_Initialize(DRV_NVM_INDEX_0, (SYS_MODULE_INIT*)&drvNVMInit);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized also the type of memory used
init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_NVM_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

);

DRV_NVM_Deinitialize Function

Deinitializes the specified instance of the NVM driver module

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
void DRV_NVM_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the NVM driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

Preconditions

Function DRV_NVM_Initialize should have been called before calling this function.

Parameter: object - Driver object handle, returned from the DRV_NVM_Initialize routine

Example

```
// This code snippet shows an example
// of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_NVM_Initialize
SYS_STATUS          status;

DRV_NVM_Deinitialize(object);

status = DRV_NVM_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Function

void DRV_NVM_Deinitialize

(

SYS_MODULE_OBJ object
);

DRV_NVM_Status Function

Gets the current status of the NVM driver module.

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
SYS_STATUS DRV_NVM_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and accept requests for new operations.
SYS_STATUS_UNINITIALIZED - Indicates the driver is not initialized.

Description

This routine provides the current status of the NVM driver module.

Remarks

This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_NVM_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_NVM_Initialize
SYS_STATUS          NVMStatus;

NVMStatus = DRV_NVM_Status(object);
else if (SYS_STATUS_ERROR >= NVMStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_NVM_Initialize routine

Function

SYS_STATUS DRV_NVM_Status
(
SYS_MODULE_OBJ object
);

b) Client Core Functions

DRV_NVM_Open Function

Opens the specified NVM driver instance and returns a handle to it

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
DRV_HANDLE DRV_NVM_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, DRV_HANDLE_INVALID is returned. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_NVM_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver hardware instance being opened is not initialized or is invalid

Description

This routine opens the specified NVM driver instance and provides a handle. This handle must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_NVM_Close routine is called. This routine will NEVER block waiting for hardware. If the driver has has already been opened, it cannot be opened exclusively.

Preconditions

Function DRV_NVM_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_NVM_Open(DRV_NVM_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

```
  DRV_HANDLE DRV_NVM_Open
(
const SYS_MODULE_INDEX index,
const       DRV_IO_INTENT ioIntent
);
```

DRV_NVM_Close Function

Closes an opened-instance of the NVM driver

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
void DRV_NVM_Close(const DRV_HANDLE handle);
```

Returns

None

Description

This routine closes an opened-instance of the NVM driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_NVM_Open before the caller may use the driver again. Usually there is no need for the driver client to verify that the Close

operation has completed.

Preconditions

The DRV_NVM_Initialize routine must have been called for the specified NVM driver instance.

DRV_NVM_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_NVM_Open

DRV_NVM_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_NVM_Close

(

const DRV_HANDLE handle

);

DRV_NVM_Read Function

Reads blocks of data from the specified address in memory.

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
void DRV_NVM_Read(const DRV_HANDLE handle, DRV_NVM_COMMAND_HANDLE * commandHandle, void * targetBuffer,
uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_NVM_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This routine reads blocks of data from the specified address in memory. This operation is blocking and returns with the required data in the target buffer. If an event handler is registered with the driver the event handler would be invoked from within this function to indicate the status of the operation. This function should not be used to read areas of memory which are queued to be programmed or erased. If required, the program or erase operations should be allowed to complete. The function returns DRV_NVM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the driver handle is invalid
- if the target buffer pointer is NULL
- if the number of blocks to be read is zero or more than the actual number of blocks available
- if a buffer object could not be allocated to the request
- if the client opened the driver in write only mode

Remarks

None.

Preconditions

The DRV_NVM_Initialize routine must have been called for the specified NVM driver instance.

DRV_NVM_Open must have been called with DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE as the ioIntent to obtain a valid opened device handle.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = NVM_BASE_ADDRESS_TO_READ_FROM;
```

```
uint32_t    nBlock = 2;
DRV_NVM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myNVMHandle is the handle returned
// by the DRV_NVM_Open function.

DRV_NVM_Read(myNVMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_NVM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}
else
{
    // Read Successful
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
targetBuffer	Buffer into which the data read from the NVM Flash instance will be placed
blockStart	Start block address in NVM memory from where the read should begin. It can be any address of the flash.
nBlock	Total number of blocks to be read. Each Read block is of 1 byte.

Function

void DRV_NVM_Read

(

const DRV_HANDLE handle,

DRV_NVM_COMMAND_HANDLE * commandHandle,

void * targetBuffer,

uint32_t blockStart,

uint32_t nBlock

);

DRV_NVM_Write Function

Writes blocks of data starting from the specified address in flash memory.

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
void DRV_NVM_Write(const DRV_HANDLE handle, DRV_NVM_COMMAND_HANDLE * commandHandle, void * sourceBuffer,
uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_NVM_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data into flash memory. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_NVM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer object could not be allocated to the request
- if the source buffer pointer is NULL
- if the client opened the driver for read only
- if the number of blocks to be written is either zero or more than the number of blocks actually available

- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_NVM_EVENT_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_NVM_EVENT_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

Performing a flash programming operation while executing (fetching) instructions from program Flash memory, the CPU stalls (waits) until the programming operation is finished. The CPU will not execute any instruction, or respond to interrupts, during this time. If any interrupts occur during the programming cycle, they remain pending until the cycle completes. This makes the NVM write operation blocking in nature.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

DRV_NVM_Open() routine must have been called to obtain a valid opened device handle. DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified as a parameter to this routine.

The flash address location which has to be written, must have be erased before using the DRV_NVM_Erase() routine.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = NVM_BASE_ADDRESS_TO_WRITE_TO;
uint32_t    nBlock = 2;
DRV_NVM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myNVMHandle is the handle returned
// by the DRV_NVM_Open function.

// Client registers an event handler with driver

DRV_NVM_EventHandlerSet(myNVMHandle, APP_NVMEventHandler, (uintptr_t)&myAppObj);

DRV_NVM_Write(myNVMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_NVM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_NVMEventHandler(DRV_NVM_EVENT event,
        DRV_NVM_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_NVM_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_NVM_EVENT_COMMAND_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
sourceBuffer	The source buffer containing data to be programmed into NVM Flash
blockStart	Start block address of NVM Flash where the write should begin. This address should be aligned on a block boundary.
nBlock	Total number of blocks to be written.

Function

```
void DRV_NVM_Write
(
const       DRV_HANDLE handle,
      DRV_NVM_COMMAND_HANDLE * commandHandle,
void * sourceBuffer,
uint32_t blockStart,
uint32_t nBlock
);
```

DRV_NVM_Erase Function

Erase the specified number of blocks of the Flash memory.

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
void DRV_NVM_Erase(const DRV_HANDLE handle, DRV_NVM_COMMAND_HANDLE * commandHandle, uint32_t blockStart,
uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It Will be DRV_NVM_COMMAND_HANDLE_INVALID if the request was not queued.

Description

This function schedules a non-blocking erase operation of flash memory. The function returns with a valid erase handle in the commandHandle argument if the erase request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_NVM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer object could not be allocated to the request
- if the client opened the driver for read only
- if the number of blocks to be erased is either zero or more than the number of blocks actually available
- if the erase queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_NVM_EVENT_COMMAND_COMPLETE event if the erase operation was successful or DRV_NVM_EVENT_COMMAND_ERROR event if the erase operation was not successful.

Remarks

Performing a flash erase operation while executing (fetching) instructions from program Flash memory, the CPU stalls (waits) until the erase operation is finished. The CPU will not execute any instruction, or respond to interrupts, during this time. If any interrupts occur during the programming cycle, they remain pending until the cycle completes. This make the NVM erase operation blocking in nature.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() routine must have been called with DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE to obtain a valid opened device handle.

Example

```
// Destination address should be block aligned.
uint32_t blockStart;
uint32_t nBlock;
DRV_NVM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myNVMHandle is the handle returned
// by the DRV_NVM_Open function.

// Client registers an event handler with driver

DRV_NVM_EventHandlerSet(myNVMHandle, APP_NVMEventHandler, (uintptr_t)&myAppObj);

DRV_NVM_Erase( myNVMHandle, &commandHandle, blockStart, nBlock );

if(DRV_NVM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer queue is processed.

void APP_NVMEventHandler(DRV_NVM_EVENT event,
        DRV_NVM_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_NVM_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_NVM_EVENT_COMMAND_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
blockStart	Start block address in NVM memory from where the erase should begin. This should be aligned on a DRV_NVM_PAGE_SIZE byte boundary.
nBlock	Total number of blocks to be erased.

Function

```
void DRV_NVM_Erase
(
const       DRV_HANDLE handle,
    DRV_NVM_COMMAND_HANDLE * commandHandle,
uint32_t blockStart,
uint32_t nBlock
);
```

DRV_NVM_EraseWrite Function

Erase and Write blocks of data starting from a specified address in flash memory.

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
void DRV_NVM_EraseWrite(const DRV_HANDLE handle, DRV_NVM_COMMAND_HANDLE * commandHandle, void *
sourceBuffer, uint32_t writeBlockStart, uint32_t nWriteBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It Will be DRV_NVM_COMMAND_HANDLE_INVALID if the request was not queued.

Description

This function combines the step of erasing a page and then writing the row. The application can use this function if it wants to avoid having to explicitly delete a page in order to update the rows contained in the page.

This function schedules a non-blocking operation to erase and write blocks of data into flash memory. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_NVM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only
- if the buffer size is 0
- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_NVM_EVENT_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_NVM_EVENT_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

In order to use this function, the DRV_NVM_ERASE_WRITE_ENABLE must be defined in system_config.h and the drv_nvm_erasewrite.c file must be included in the project.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() must have been called with DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE as a parameter to obtain a valid opened device handle.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = NVM_BASE_ADDRESS_TO_WRITE_TO;
uint32_t    nBlock = 2;
DRV_NVM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myNVMHandle is the handle returned
// by the DRV_NVM_Open function.

// Client registers an event handler with driver

DRV_NVM_EventHandlerSet(myNVMHandle, APP_NVMEventHandler, (uintptr_t)&myAppObj);

DRV_NVM_EraseWrite(myNVMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_NVM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}
```

```
// Event is received when
// the buffer is processed.

void APP_NVMEventHandler(DRV_NVM_EVENT event,
        DRV_NVM_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_NVM_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_NVM_EVENT_COMMAND_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle. If NULL, then buffer handle is not returned.
sourceBuffer	The source buffer containing data to be programmed into NVM Flash
writeBlockStart	Start block address of NVM Flash where the write should begin. This address should be aligned on a DRV_NVM_ROW_SIZE byte boundary.
nWriteBlock	Total number of blocks to be written.

Function

void DRV_NVM_EraseWrite
(
const DRV_HANDLE handle,
DRV_NVM_COMMAND_HANDLE * commandHandle,
void * sourceBuffer,
uint32_t writeBlockStart,
uint32_t nWriteBlock
);

DRV_NVM_EventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
void DRV_NVM_EventHandlerSet(const DRV_HANDLE handle, const void * eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client calls a write or erase function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this

handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any write or erase operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_NVM_COMMAND_HANDLE commandHandle;

// drvNVMHandle is the handle returned
// by the DRV_NVM_Open function.

// Client registers an event handler with driver. This is done once.

DRV_NVM_EventHandlerSet(drvNVMHandle, APP_NVMEventHandler, (uintptr_t)&myAppObj);

DRV_NVM_Read(drvNVMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_NVM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_NVMEventHandler(DRV_NVM_EVENT event,
        DRV_NVM_COMMAND_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_NVM_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_NVM_EVENT_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user

context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_NVM_EventHandlerSet
(
const DRV_HANDLE handle,
const void * eventHandler,
const uintptr_t context
);

c) Client Block Data Functions

DRV_NVM_Tasks Function

Maintains the driver's erase and write state machine and implements its ISR.

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
void DRV_NVM_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal write and erase state machine and implement its ISR for interrupt-driven implementations.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_NVM_Initialize routine must have been called for the specified NVM driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_NVM_Initialize

while (true)
{
    DRV_NVM_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_NVM_Initialize)

Function

void DRV_NVM_Tasks
(
SYS_MODULE_OBJ object
);

d) Status Functions

DRV_NVM_AddressGet Function

Returns the NVM media start address

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
uintptr_t DRV_NVM_AddressGet(const DRV_HANDLE handle);
```

Returns

Start address of the NVM Media if the handle is valid otherwise NULL.

Description

This function returns the NVM Media start address.

Remarks

None.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
uintptr_t startAddress;
startAddress = DRV_NVM_AddressGet(drvNVMHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

uintptr_t DRV_NVM_AddressGet

(

const DRV_HANDLE handle

);

DRV_NVM_CommandStatus Function

Gets the current status of the command.

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
DRV_NVM_COMMAND_STATUS DRV_NVM_CommandStatus(const DRV_HANDLE handle, const DRV_NVM_COMMAND_HANDLE
commandHandle);
```

Returns

A DRV_NVM_COMMAND_STATUS value describing the current status of the command. Returns DRV_NVM_COMMAND_HANDLE_INVALID if the client handle or the command handle is not valid.

Description

This routine gets the current status of the command. The application must use this routine where the status of a scheduled command needs to polled on. The function may return DRV_NVM_COMMAND_HANDLE_INVALID in a case where the command handle has expired. A command

handle expires when the internal buffer object is re-assigned to another erase or write request. It is recommended that this function be called regularly in order to track the command status correctly.

The application can alternatively register an event handler to receive write or erase operation completion events.

Remarks

This routine will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_NVM_Initialize() routine must have been called.

The DRV_NVM_Open() must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE                  handle;         // Returned from DRV_NVM_Open
DRV_NVM_COMMAND_HANDLE      commandHandle;
DRV_NVM_COMMAND_STATUS      status;

status = DRV_NVM_CommandStatus(handle, commandHandle);
if(status == DRV_NVM_COMMAND_COMPLETED)
{
    // Operation Done
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_NVM_COMMAND_STATUS DRV_NVM_CommandStatus
(
const DRV_HANDLE handle,
const DRV_NVM_COMMAND_HANDLE commandHandle
);

DRV_NVM_GeometryGet Function

Returns the geometry of the device.

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_NVM_GeometryGet(const DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Pointer to structure which holds the media geometry information.

Description

This API gives the following geometrical details of the NVM Flash:

- Media Property
- Number of Read/Write/Erase regions in the flash device
- Number of Blocks and their size in each region of the device

Remarks

None.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
SYS_FS_MEDIA_GEOMETRY * nvmFlashGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

nvmFlashGeometry = DRV_NVM_GeometryGet(nvmOpenHandle1);

readBlockSize  = nvmFlashGeometry->geometryTable->blockSize;
nReadBlocks = nvmFlashGeometry->geometryTable->numBlocks;
nReadRegions = nvmFlashGeometry->numReadRegions;

writeBlockSize  = (nvmFlashGeometry->geometryTable +1)->blockSize;
eraseBlockSize  = (nvmFlashGeometry->geometryTable +2)->blockSize;

totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY * DRV_NVM_GeometryGet

(

const DRV_HANDLE handle

);

e) Miscellaneous Functions

DRV_NVM_IsAttached Function

Returns the physical attach status of the NVM.

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
bool DRV_NVM_IsAttached(const DRV_HANDLE handle);
```

Returns

Returns false if the handle is invalid otherwise returns true.

Description

This function returns the physical attach status of the NVM.

Remarks

None.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// The NVM media is always attached and so the below
// always returns true.

bool isNVMAttached;
isNVMAttached = DRV_NVM_isAttached(drvNVMHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_NVM_IsAttached
(
const DRV_HANDLE handle
);

DRV_NVM_IsWriteProtected Function

Returns the write protect status of the NVM.

Implementation: Static/Dynamic

File

drv_nvm.h

C

```
bool DRV_NVM_IsWriteProtected(const DRV_HANDLE handle);
```

Returns

Always returns false.

Description

This function returns the physical attach status of the NVM. This function always returns false.

Remarks

None.

Preconditions

The DRV_NVM_Initialize() routine must have been called for the specified NVM driver instance.

The DRV_NVM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// The NVM media is treated as always writeable.
bool isWriteProtected;
isWriteProtected = DRV_NVM_IsWriteProtected(drvNVMHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_NVM_IsWriteProtected
(
const DRV_HANDLE handle
);

f) Data Types and Constants

DRV_NVM_INDEX_0 Macro

NVM driver index definitions

File

drv_nvm.h

C

```
#define DRV_NVM_INDEX_0 0
```

Description

Driver NVM Module Index reference

These constants provide NVM driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_NVM_Initialize and DRV_NVM_Open routines to identify the driver instance in use.

DRV_NVM_INIT Structure

Defines the data required to initialize or reinitialize the NVM driver

File

drv_nvm.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  NVM_MODULE_ID nvmID;
  INT_SOURCE interruptSource;
  uint32_t mediaStartAddress;
  const SYS_FS_MEDIA_GEOMETRY * nvmMediaGeometry;
} DRV_NVM_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
NVM_MODULE_ID nvmID;	Identifies NVM hardware module (PLIB-level) ID
INT_SOURCE interruptSource;	Interrupt Source for Write Interrupt
uint32_t mediaStartAddress;	NVM Media start address. The driver treats this address as • block 0 address for read, write and erase operations.
const SYS_FS_MEDIA_GEOMETRY * nvmMediaGeometry;	NVM Media geometry object.

Description

NVM Driver Initialization Data

This data type defines the data required to initialize or reinitialize the NVM driver.

Remarks

Not all initialization features are available for all devices. Please refer to the specific device data sheet to determine availability.

DRV_NVM_INDEX_1 Macro

File

drv_nvm.h

C

```
#define DRV_NVM_INDEX_1 1
```

Description

This is macro DRV_NVM_INDEX_1.

DRV_NVM_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_nvm.h

C

```
typedef enum {
  DRV_NVM_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_NVM_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR
} DRV_NVM_EVENT;
```

Members

Members	Description
DRV_NVM_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE	Operation has been completed successfully.
DRV_NVM_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR	There was an error during the operation

Description

NVM Driver Events

This enumeration identifies the possible events that can result from a Write or Erase request caused by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_NVM_EventHandlerSet function when a request is completed.

DRV_NVM_EVENT_HANDLER Type

Pointer to a NVM Driver Event handler function

File

drv_nvm.h

C

```
typedef SYS_FS_MEDIA_EVENT_HANDLER DRV_NVM_EVENT_HANDLER;
```

Returns

None.

Description

NVM Driver Event Handler Function Pointer

This data type defines the required function signature for the NVM event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_NVM_EVENT_COMMAND_COMPLETE, it means that the write or a erase operation was completed successfully.

If the event is DRV_NVM_EVENT_COMMAND_ERROR, it means that the scheduled operation was not completed successfully.

The context parameter contains the handle to the client context, provided at the time the event handling function was registered using the DRV_NVM_EventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write/erase request.

The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations within this function.

Example

```
void APP_MyNvmEventHandler
(
    DRV_NVM_EVENT event,
    DRV_NVM_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
```

```
    {
        case DRV_NVM_EVENT_COMMAND_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_NVM_EVENT_COMMAND_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write/Erase requests
context	Value identifying the context of the application that registered the event handling function

DRV_NVM_COMMAND_HANDLE Type

Handle identifying commands queued in the driver.

File

drv_nvm.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_NVM_COMMAND_HANDLE;
```

Description

NVM Driver command handle.

A command handle is returned by a call to the Read, Write or Erase functions. This handle allows the application to track the completion of the operation. This command handle is also returned to the client along with the event that has occurred with respect to the command. This allows the application to connect the event to a specific command in case where multiple commands are queued.

The command handle associated with the command request expires when the client has been notified of the completion of the command (after event handler function that notifies the client returns) or after the command has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_NVM_COMMAND_STATUS Enumeration

Specifies the status of the command for the read, write and erase operations.

File

drv_nvm.h

C

```
typedef enum {
  DRV_NVM_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED,
  DRV_NVM_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED,
  DRV_NVM_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS,
  DRV_NVM_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN
} DRV_NVM_COMMAND_STATUS;
```

Members

Members	Description
DRV_NVM_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED	Done OK and ready
DRV_NVM_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED	Scheduled but not started

DRV_NVM_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS	Currently being in transfer
DRV_NVM_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN	Unknown Command

Description

NVM Driver Command Status

NVM Driver command Status

This type specifies the status of the command for the read, write and erase operations.

Remarks

None.

DRV_NVM_COMMAND_HANDLE_INVALID Macro

This value defines the NVM Driver's Invalid Command Handle.

File

drv_nvm.h

C

```
#define DRV_NVM_COMMAND_HANDLE_INVALID SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID
```

Description

NVM Driver Invalid Command Handle.

This value defines the NVM Driver Invalid Command Handle. This value is returned by read/write/erase routines when the command request was not accepted.

Remarks

None.

Files

Files

Name	Description
drv_nvm.h	NVM Driver Interface Definition
drv_nvm_config_template.h	NVM driver configuration definitions.

Description

This section lists the source and header files used by the NVM Driver Library.

drv_nvm.h

NVM Driver Interface Definition

Enumerations

	Name	Description
	DRV_NVM_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.
	DRV_NVM_EVENT	Identifies the possible events that can result from a request.

Functions

	Name	Description
	DRV_NVM_AddressGet	Returns the NVM media start address **Implementation:** Static/Dynamic
	DRV_NVM_Close	Closes an opened-instance of the NVM driver **Implementation:** Static/Dynamic
	DRV_NVM_CommandStatus	Gets the current status of the command. **Implementation:** Static/Dynamic

	DRV_NVM_Deinitialize	Deinitializes the specified instance of the NVM driver module **Implementation:** Static/Dynamic
	DRV_NVM_Erase	Erase the specified number of blocks of the Flash memory. **Implementation:** Static/Dynamic
	DRV_NVM_EraseWrite	Erase and Write blocks of data starting from a specified address in flash memory. **Implementation:** Static/Dynamic
	DRV_NVM_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed. **Implementation:** Static/Dynamic
	DRV_NVM_GeometryGet	Returns the geometry of the device. **Implementation:** Static/Dynamic
	DRV_NVM_Initialize	Initializes the NVM instance for the specified driver index **Implementation:** Static/Dynamic
	DRV_NVM_IsAttached	Returns the physical attach status of the NVM. **Implementation:** Static/Dynamic
	DRV_NVM_IsWriteProtected	Returns the write protect status of the NVM. **Implementation:** Static/Dynamic
	DRV_NVM_Open	Opens the specified NVM driver instance and returns a handle to it **Implementation:** Static/Dynamic
	DRV_NVM_Read	Reads blocks of data from the specified address in memory. **Implementation:** Static/Dynamic
	DRV_NVM_Status	Gets the current status of the NVM driver module. **Implementation:** Static/Dynamic
	DRV_NVM_Tasks	Maintains the driver's erase and write state machine and implements its ISR. **Implementation:** Static/Dynamic
	DRV_NVM_Write	Writes blocks of data starting from the specified address in flash memory. **Implementation:** Static/Dynamic

Macros

	Name	Description
	DRV_NVM_COMMAND_HANDLE_INVALID	This value defines the NVM Driver's Invalid Command Handle.
	DRV_NVM_INDEX_0	NVM driver index definitions
	DRV_NVM_INDEX_1	This is macro DRV_NVM_INDEX_1.
	DRV_NVM_PAGE_SIZE	Specifies the NVM Driver Program Page Size in bytes.
	DRV_NVM_ROW_SIZE	Specifies the NVM Driver Program Row Size in bytes.

Structures

	Name	Description
	DRV_NVM_INIT	Defines the data required to initialize or reinitialize the NVM driver

Types

	Name	Description
	DRV_NVM_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_NVM_EVENT_HANDLER	Pointer to a NVM Driver Event handler function

Description

NVM Driver Interface Definition

The NVM driver provides a simple interface to manage the Non Volatile Flash Memory on Microchip microcontrollers. This file defines the interface definition for the NVM driver.

File Name

drv_nvm.h

Company

Microchip Technology Inc.

drv_nvm_config_template.h

NVM driver configuration definitions.

Macros

	Name	Description
	DRV_NVM_BUFFER_OBJECT_NUMBER	Selects the maximum number of buffer objects
	DRV_NVM_CLIENTS_NUMBER	Selects the maximum number of clients
	DRV_NVM_DISABLE_ERROR_CHECK	Disables the error checks in the driver.
	DRV_NVM_ERASE_WRITE_ENABLE	Enables support for NVM Driver Erase Write Feature.
	DRV_NVM_INSTANCES_NUMBER	Selects the maximum number of Driver instances that can be supported by the dynamic driver.
	DRV_NVM_INTERRUPT_MODE	Macro specifies operation of the driver to be in the interrupt mode or polled mode
	DRV_NVM_MEDIA_SIZE	Specifies the NVM Media size.
	DRV_NVM_MEDIA_START_ADDRESS	Specifies the NVM Media start address.
	DRV_NVM_SYS_FS_REGISTER	Register to use with the File system

Description

NVM Driver Configuration Template Header file.

This template file describes all the mandatory and optional configuration macros that are needed for building the NVM driver. Do not include this file in source code.

File Name

drv_nvm_config_template.h

Company

Microchip Technology Inc.

Output Compare Driver Library

This section describes the Output Compare Driver Library.

Introduction

The Output Compare Static Driver provides a high-level interface to manage the Output Compare module on the Microchip family of microcontrollers.

Description

Through the MHC, this driver provides APIs for the following:

- Initializing the module
- Enabling/Disabling of the output compare
- Starting/Stopping of the output compare
- Fault checking

Library Interface

Functions

	Name	Description
	DRV_OC_Disable	Disables the Output Compare instance for the specified driver index. **Implementation:** Static
	DRV_OC_Enable	Enables the Output Compare for the specified driver index. **Implementation:** Static
	DRV_OC_FaultHasOccurred	Checks if a Fault has occurred for the specified driver index. **Implementation:** Static
	DRV_OC_Initialize	Initializes the Comparator instance for the specified driver index. **Implementation:** Static

	DRV_OC_Start	Starts the Comparator instance for the specified driver index. **Implementation:** Static
	DRV_OC_Stop	Stops the Output Compare instance for the specified driver index. **Implementation:** Static

Description

This section describes the Application Programming Interface (API) functions of the Output Compare Driver Library.

Functions

DRV_OC_Disable Function

Disables the Output Compare instance for the specified driver index.

Implementation: Static

File

help_drv_oc.h

C

```
void DRV_OC_Disable();
```

Returns

None.

Description

This routine disables the Output Compare for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Remarks

None.

Preconditions

DRV_OC_Initialize has been called.

Function

void DRV_OC_Disable(void)

DRV_OC_Enable Function

Enables the Output Compare for the specified driver index.

Implementation: Static

File

help_drv_oc.h

C

```
void DRV_OC_Enable();
```

Returns

None.

Description

This routine enables the Output Compare for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Remarks

None.

Preconditions

DRV_OC_Initialize has been called.

Function

void DRV_OC_Enable(void)

DRV_OC_FaultHasOccurred Function

Checks if a Fault has occurred for the specified driver index.

Implementation: Static

File

help_drv_oc.h

C

```
bool DRV_OC_FaultHasOccurred();
```

Returns

Boolean

- 1 - A Fault has occurred
- 0 - A Fault has not occurred

Description

This routine checks whether or not a Fault has occurred for the specified driver index. The initialization routine is specified by the MHC parameters.

Remarks

None.

Preconditions

DRV_OC_Initialize has been called.

Function

bool DRV_OC_FaultHasOccurred(void)

DRV_OC_Initialize Function

Initializes the Comparator instance for the specified driver index.

Implementation: Static

File

help_drv_oc.h

C

```
void DRV_OC_Initialize();
```

Returns

None.

Description

This routine initializes the Output Compare driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters. The driver instance index is independent of the Output Compare module ID. For example, driver instance 0 can be assigned to Output Compare 1.

Remarks

This routine must be called before any other Comparator routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Function

void DRV_OC_Initialize(void)

DRV_OC_Start Function

Starts the Comparator instance for the specified driver index.

Implementation: Static

File

help_drv_oc.h

C

```
void DRV_OC_Start();
```

Returns

None.

Description

This routine starts the Output Compare for the specified driver instance.

Remarks

None.

Preconditions

DRV_OC_Initialize has been called.

Function

void DRV_OC_Start(void)

DRV_OC_Stop Function

Stops the Output Compare instance for the specified driver index.

Implementation: Static

File

help_drv_oc.h

C

```
void DRV_OC_Stop();
```

Returns

None.

Description

This routine stops the Output Compare for the specified driver instance.

Remarks

None.

Preconditions

DRV_OC_Initialize has been called.

Function

void DRV_OC_Stop(void)

Parallel Master Port (PMP) Driver Library

This section describes the Parallel Master Port Driver Library.

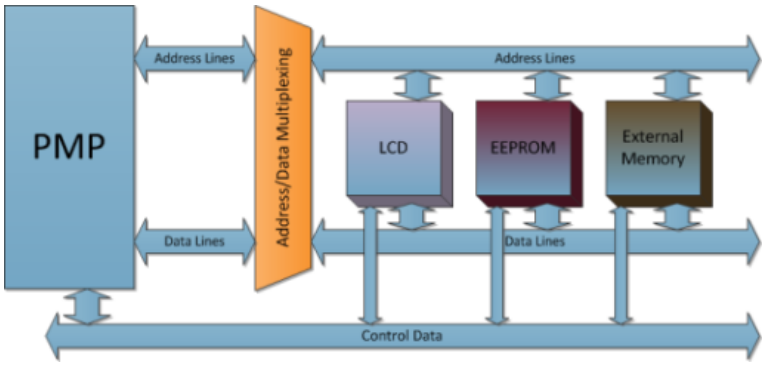
Introduction

This library provides an interface to manage the Parallel Master Port (PMP) module on Microchip family of microcontrollers in different modes of operation.

Description

The Parallel Master Port (PMP) is a parallel 8-bit/16-bit I/O module specifically designed to communicate with a wide variety of parallel devices such as communications peripherals, LCDs, external memory devices and microcontrollers. Because the interfaces to parallel peripherals vary significantly, the PMP module is highly configurable.

The following figure shows a generic block diagram, which illustrates the ways the PMP module can be used:



The PMP module can be used in different modes. Master and Slave are the two modes that can have additional sub-modes, depending on the different microcontroller families.

Master Mode: In Master mode, the PMP module can provide a 8-bit or 16-bit data bus, up to 16 bits of address, and all of the necessary control signals to operate a variety of external parallel devices such as memory devices, peripherals and slave microcontrollers. The PMP master modes provide a simple interface for reading and writing data, but not executing program instructions from external devices, such as SRAM or Flash memories.

Slave Mode: Slave mode only supports 8-bit data and the module control pins are automatically dedicated when this mode is selected.

Using the Library

This topic describes the basic architecture of the PMP Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_pmp.h`

The interface to the PMP Driver library is defined in the `drv_pmp.h` header file. This file is included by the `drv.h` file. Any C language source (`.c`) file that uses the PMP Driver Library should include `drv.h`.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

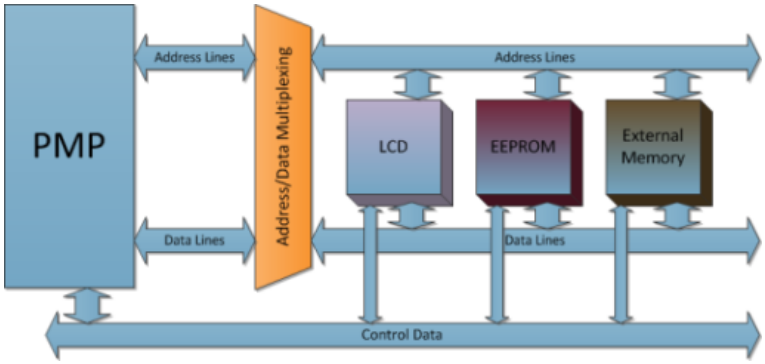
This library provides a low-level abstraction of the Parallel Master Port (PMP) module on Microchip's microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

Hardware Abstraction Model Description

Depending on the device,the PMP module provides interface routines to interact with external peripherals such as LCD, EEPROM, Flash memory, etc., as shown in the following diagram. The diagram shows the PMP module acting as a master. The PMP module can be easily configured to act as a slave. The address and data lines can be multiplexed to suit the application. The address and data buffers are up to 2-byte (16-bit) buffers for data transmitted or received by the parallel interface to the PMP bus over the data and address lines synchronized with control logic including the read and write strobe.

The desired timing wait states to suit different peripheral timings can also be programmed using the PMP module.



PMP Hardware Abstraction Model Diagram

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the PMP module.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks, and status functions.
Client Interaction Functions	Provides open, close, client status and client mode configuration functions.
Client Transfer Functions	Provides interface for data transfer in master and slave mode.
Miscellaneous	Provides driver miscellaneous functions, version identification functions, etc.

How the Library Works

This section describes how the PMP Driver Library operates.

Description

Before the driver is ready for use, its should be configured (compile time configuration). Refer to the Configuring the Library section for more details on how to configure the driver.

There are few run-time configuration items that are done during initialization of the driver instance, and a few that are client-specific and are done using dedicated functions.

To use the PMP Driver, initialization and client functions should be invoked in a specific sequence to ensure correct operation.

The following is the sequence in which various routines should be called:

1. Call DRV_PMP_Initialize to initialize the PMP Driver. Note that this may be performed by the MPLAB Harmony system module. The DRV_PMP_Status function may be used to check the status of the initialization.
2. Once initialization for a particular driver instance is done, the client wanting to use the driver can open it using DRV_PMP_Open.
3. The DRV_PMP_ModeConfig function should now be called, which will configure the driver for the exact mode of operation required by that client.
4. After configuring the mode, DRV_PMP_Write and/or DRV_PMP_Read can be called by the user application to Write/Read using the PMP module. Calling these functions does not start the PMP transfer immediately in non-interrupt mode. Instead, all of these transfer tasks are queued in an internal queue. Actual transfer starts only when the PMP Task function is called by the system/user. In interrupt mode, although transfer tasks are queued, the actual transfer starts immediately.
5. PMP Write and Read functions return an ID of that particular transfer, which should be saved by user to get the status of that transfer later.
6. The system will either call DRV_PMP_Tasks from the System Task Service (in a polled environment), or it will be called from the ISR of the PMP.
7. At any time status of the transfer can be obtained by using DRV_PMP_TransferStatus.

Note: Not all modes are available on all devices. Please refer to the specific device data sheet to determine the supported modes.

System Initialization

This section describes initialization and reinitialization features.

Description

Initialization and Reinitialization

The system performs the initialization and the reinitialization of the device driver with settings that affect only the instance of the device that is being initialized or reinitialized. During system initialization each instance of the PMP device will be initialized with the following configuration settings:

Initialization Member	Description
`moduleInit`	System module initialization of the power state.
`pmpId`	PMP hardware module ID (peripheral library-level ID).
`stopInIdle`	Decide whether or not the module should be stopped in Idle mode.
`muxMode`	To select one of the different multiplexing modes possible for PMP module.
`inputBuffer`	Select the type of Input Buffer (TTL or Schmitt Trigger).

polarity	Select polarity of different PMP pins.
ports	Set the pins the user wants to use as port or PMP pins.

The DRV_PMP_Initialize function returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the initialize interface would be used by the other system interfaces, such as DRV_PMP_Reinitialize, DRV_PMP_Deinitialize, DRV_PMP_Status, and DRV_PMP_Tasks.

Example for PMP Initialization Through the DRV_PMP_INIT Structure

```
DRV_PMP_INIT     init;
SYS_MODULE_OBJ   object;
SYS_STATUS       pmpStatus;

// populate the PMP init configuration structure
    init.inputBuffer = PMP_INPUT_BUFFER_TTL;
    init.polarity.addressLatchPolarity = PMP_POLARITY_ACTIVE_HIGH;
    init.polarity.rwStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
    init.polarity.writeEnableStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
    init.polarity.chipselect1Polarity = PMP_POLARITY_ACTIVE_HIGH;
    init.polarity.chipselect2Polarity = PMP_POLARITY_ACTIVE_LOW;
    init.ports.addressPortsMask = PMP_PMA0_PORT | PMP_PMA1_PORT | PMP_PMA2_TO_PMA13_PORTS | PMP_PMA14_PORT;
    init.ports.readWriteStrobe = PORT_ENABLE;
    init.ports.writeEnableStrobe = PORT_ENABLE;
    init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
    init.pmpID            = PMP_ID_0;
    init.stopInIdle       = false;
    init.muxMode          = PMP_MUX_NONE;

object = DRV_PMP_Initialize (DRV_PMP_INDEX_0, (SYS_MODULE_INIT *)&init);

pmpStatus = DRV_PMP_Status(object);

if ( SYS_STATUS_READY != pmpStatus)
{
    // Handle error
}
```

Deinitialization

Once the initialize operation has been called, the deinitialize operation must be called before the initialize operation can be called again. This routine may block if the driver is running in an OS environment that supports blocking operations and the driver requires system resources access. However, the function will never block for hardware PMP access. If the operation requires time to allow the hardware to complete, which will be reported by DRV_PMP_Status.

Status

PMP status is available to query the module state before, during and after initialization, deinitialization, and reinitialization.

Tasks Routine

The DRV_PMP_Tasks function will see the queue status and perform the task of transferring the data accordingly. In the Blocking mode when interrupts are disabled, it will finish one of the tasks completely (that means emptying one space in queue), and then return back. Whereas in Non-Blocking mode, it will return back just after starting one word (8-bit or 16-bit) of transfer (may not be emptying one space in the queue, as that task may not be completely finished).

The DRV_PMP_Tasks function can be called in two ways:

- By the system task service in a polled environment
- By the ISR of the PMP in an interrupt-based system

Example: Polling

```
int main( void )
{
    SYS_MODULE_OBJ object;
    object = DRV_PMP_Initialize( DRV_PMP_INDEX_0, (SYS_MODULE_INIT *) &initConf );

    if( SYS_STATUS_READY != DRV_PMP_Status( object ) )
        return 0;

    while (1)
    {
        DRV_PMP_Tasks (object);
```

```
    }
}
```

Example: Interrupt

```
int main( void )
{
    SYS_MODULE_OBJ object;
    object = DRV_PMP_Initialize( DRV_PMP_INDEX_0, (SYS_MODULE_INIT *) &initConf );

    if( SYS_STATUS_READY != DRV_PMP_Status( object ) )
        return 0;

    while (1);
}

/* Sample interrupt routine not specific to any device family */
void ISR PMPInterrupt(void)
{
    //Call the PMP Tasks routine
    DRV_PMP_Tasks(object);
}
```

Note: A PMP transfer in Blocking mode in an interrupt environment is not supported.

Transfer Operation

This section describes transfer operation.

Description

Once the PMP Driver is open and configured for a client, it is set to start Reading/Writing through DRV_PMP_Read and DRV_PMP_Write. However, these functions will not directly start reading or writing. These will just put the relevant information in a queue in non-interrupt mode and return an ID that can be used later for checking the transfer status. In Interrupt mode, the Read/Write functions will trigger the transfer immediately after storing the transfer information in the queue.

The user must use a buffer pointing to character for data values.

The repeatCount parameter allows the user to repeatedly write the same nBytes of data into the slave devices.

Example:

```
unsigned char myReadBuffer[300], myWriteBuffer[100]; // has to be 'char' arrays
uint32_t deviceAddress, nBytes, repeatCount, i;
uint32_t writeID, readID;
DRV_HANDLE handle;

//initialize, open and configure the driver/client
/* ... */

deviceAddress = 0x0206;
nBytes = 100;
repeatCount = 0x01;
for (i=0; i<nBytes; i++)
{
   myWriteBuffer[i]=i*5+7;
}

/* it will write 100 bytes of data in the location starting from 0x0206 and then it will repeat
 writing the same set of data in next 100 location starting from 0x206+100 for 8 bit data mode
 and 50 location starting from 0x206+50 for 16 bit data mode. */
writeID = DRV_PMP_Write ( handle, deviceAddress, &myWriteBuffer[0], nBytes, repeatCount);

// it will read 300 locations starting from 0x0206 into myReadBuffer
readID = DRV_PMP_Read ( handle, deviceAddress, &myReadBuffer[0], nBytes);
```

Transfer Status

The status of the read/write transfers can be obtained using API DRV_PMP_TransferStatus.

Example:

```
DRV_PMP_TRANSFER_STATUS writeStatus, readStatus;
uint32_t writeID, readID;
```

```
writeStatus = DRV_PMP_TransferStatus( DRV_PMP_INDEX_0, writeID);
readStatus = DRV_PMP_TransferStatus( DRV_PMP_INDEX_0, readID);
```

Client Operation

This section describes general client operation.

Description

General Client Operation

For the application to start using an instance of the module, it must call the DRV_PMP_Open function with a specific intent. This provides the configuration required to open the PMP instance for operation. If the driver is deinitialized using the function DRV_PMP_Deinitialize, the application must call the DRV_PMP_Open function again to set up the instance of the PMP. The function DRV_PMP_Open need not be called again if the system is reinitialized using the DRV_PMP_Reinitialize function.

The PMP driver supports DRV_IO_INTENT_NONBLOCKING, DRV_IO_INTENT_BLOCKING, DRV_IO_INTENT_EXCLUSIVE, and DRV_IO_INTENT_SHARED IO.

Example:

```
DRV_HANDLE handle;

// Open the instance DRV_PMP_INDEX_0 with Non-blocking and Shared intent
handle = DRV_PMP_Open(DRV_PMP_INDEX_0, DRV_IO_INTENT_SHARED | DRV_IO_INTENT_NONBLOCKING);

if( handle == DRV_HANDLE_INVALID )
{
    // Client cannot open the instance.
}
```

The function DRV_PMP_Close closes an already opened instance of the PMP driver, invalidating the handle. DRV_PMP_Open must have been called to obtain a valid opened device handle.

Example:

```
DRV_HANDLE handle;

// Open the instance DRV_PMP_INDEX_0 with Non-blocking and Shared intent
handle = DRV_PMP_Open(DRV_PMP_INDEX_0, DRV_IO_INTENT_SHARED | DRV_IO_INTENT_NONBLOCKING);

/*...*/

DRV_PMP_Close( handle );
```

The client has the option to check the status through the function DRV_PMP_ClientStatus.

Example:

```
DRV_HANDLE handle;

// Open the instance DRV_PMP_INDEX_0 with Non-blocking and Shared intent
handle = DRV_PMP_Open(DRV_PMP_INDEX_0, DRV_IO_INTENT_SHARED | DRV_IO_INTENT_NONBLOCKING);

if ( DRV_PMP_CLIENT_STATUS_OPEN != DRV_PMP_ClientStatus( handle ) )
   return 0;
```

Client Mode Setting

Any client-specific PMP configuration has to be done using a separate function, DRV_PMP_ModeConfig. This function must be called after the client is open using DRV_PMP_Open.

Following are the client-specific configuration parameters the user can set using this function:

Configuration Parameter	Description
pmpMode	Selects the PMP mode (master or slave) to use.
intMode	Selects the interrupt mode to use.
incrementMode	Sets up address for either auto-increment or decrement mode.
endianMode	Sets Little/Big endian mode.
portSize	Specifies the data width (8-bit or 16-bit).
waitStates	Selects the different wait states.
chipSelect	Selects the Chip Select line.

Example:

```
DRV_HANDLE handle;
DRV_PMP_MODE_CONFIG config;

config.chipSelect = PMCS1_AND_PMCS2_AS_CHIP_SELECT;
config.endianMode = LITTLE_ENDIAN;
config.incrementMode = PMP_ADDRESS_AUTO_INCREMENT;
config.intMode =  PMP_INTERRUPT_NONE;
config.pmpMode = PMP_MASTER_READ_WRITE_STROBES_INDEPENDENT; //Master Mode 2
config.portSize =  PMP_DATA_SIZE_8_BITS;
config.waitStates.dataHoldWait = PMP_DATA_HOLD_2;
config.waitStates.dataWait = PMP_DATA_WAIT_THREE;
config.waitStates.strobeWait = PMP_STROBE_WAIT_5;

// Open the instance DRV_PMP_INDEX_0 with Non-blocking and Shared intent
handle = DRV_PMP_Open(DRV_PMP_INDEX_0, DRV_IO_INTENT_SHARED | DRV_IO_INTENT_NONBLOCKING);

// Configure the client
DRV_PMP_ModeConfig ( handle, config );
```

Example Code for Complete Operation

A code example of complete operation is provided in this section.

Description

This example code will write 100 bytes of data twice (i.e., repeat once) in the memory location starting from 0x0206, and then it will be read in the buffer, myReadBuffer. The modes selected for this transfer are:

- Non-blocking
- No Interrupt
- PMP Master Mode 2
- Address Auto-increment
- No Address/Data Lines Multiplexing
- 8-bit data

Example:

```
void main(void)
{
DRV_PMP_INIT     init;
SYS_MODULE_OBJ   object;
SYS_STATUS       pmpStatus;
DRV_HANDLE       handle;
DRV_PMP_MODE_CONFIG config;
unsigned char myReadBuffer[300], myWriteBuffer[100];
uint32_t deviceAddress, nBytes, repeatCount, i;
uint32_t writeID, readID;
DRV_PMP_TRANSFER_STATUS writeStatus=0, readStatus=0;


// populate the PMP init configuration structure
    init.inputBuffer = PMP_INPUT_BUFFER_TTL;
    init.polarity.addressLatchPolarity = PMP_POLARITY_ACTIVE_HIGH;
    init.polarity.rwStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
    init.polarity.writeEnableStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
    init.polarity.chipselect1Polarity = PMP_POLARITY_ACTIVE_HIGH;
    init.polarity.chipselect2Polarity = PMP_POLARITY_ACTIVE_LOW;
    init.ports.addressPortsMask = PMP_PMA0_PORT | PMP_PMA1_PORT | PMP_PMA2_TO_PMA13_PORTS;
    init.ports.readWriteStrobe = PORT_ENABLE;
    init.ports.writeEnableStrobe = PORT_ENABLE;
    init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
    init.pmpID            = PMP_ID_0;
    init.stopInIdle       = false;
    init.muxMode          = PMP_MUX_NONE;

object = DRV_PMP_Initialize (DRV_PMP_INDEX_0, (SYS_MODULE_INIT *)&init);

pmpStatus = DRV_PMP_Status(object);

if ( SYS_STATUS_READY != pmpStatus)
```

```
{
    // Handle error
}

// Open the instance DRV_PMP_INDEX_0 with Non-blocking and Shared intent
handle = DRV_PMP_Open(DRV_PMP_INDEX_0, DRV_IO_INTENT_SHARED | DRV_IO_INTENT_NONBLOCKING);

if( handle == DRV_HANDLE_INVALID )
{
    // Client cannot open the instance.
}

config.chipSelect = PMCS1_AND_PMCS2_AS_CHIP_SELECT;
config.endianMode = LITTLE_ENDIAN;
config.incrementMode = PMP_ADDRESS_AUTO_INCREMENT;
config.intMode =  PMP_INTERRUPT_NONE;
config.pmpMode = PMP_MASTER_READ_WRITE_STROBES_INDEPENDENT; //Master Mode 2
config.portSize =  PMP_DATA_SIZE_8_BITS;
config.waitStates.dataHoldWait = PMP_DATA_HOLD_2;
config.waitStates.dataWait = PMP_DATA_WAIT_THREE;
config.waitStates.strobeWait = PMP_STROBE_WAIT_5;

// Configure the client
DRV_PMP_ModeConfig ( handle, config );

deviceAddress = 0x0206;
nBytes = 100;
repeatCount = 0x01;
for (i=0; i<nBytes; i++)
{
   myWriteBuffer[i]=i*5+7;
}

writeID = DRV_PMP_Write ( handle, deviceAddress, &myWriteBuffer[0], nBytes, repeatCount);
readID = DRV_PMP_Read ( handle, deviceAddress, &myReadBuffer[0], nBytes*2);


while(!((writeStatus == PMP_TRANSFER_FINISHED)&&(readStatus == PMP_TRANSFER_FINISHED)))
{
    DRV_PMP_Tasks (object);

    writeStatus = DRV_PMP_TransferStatus( DRV_PMP_INDEX_0, writeID);
    readStatus = DRV_PMP_TransferStatus( DRV_PMP_INDEX_0, readID);
}

while(1);
}
```

Configuring the Library

Macros

	Name	Description
	DRV_PMP_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_PMP_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_PMP_QUEUE_SIZE	PMP queue size for different instances.

Description

The configuration of the PMP driver is based on the file `drv_pmp_config.h`.

This header file contains the configuration selection for the PMP Driver. Based on the selections made, the PMP Driver may support the selected features. These configuration settings will apply to all instances of the PMP Driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_PMP_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_pmp_config.h

C

```
#define DRV_PMP_CLIENTS_NUMBER 2
```

Description

PMP maximum number of clients

This definition select the maximum number of clients that the PMP driver can support at run time.

Remarks

None.

DRV_PMP_INSTANCES_NUMBER Macro

Selects the maximum number of hardware instances that can be supported by the dynamic driver.

File

drv_pmp_config.h

C

```
#define DRV_PMP_INSTANCES_NUMBER 1
```

Description

PMP hardware instance configuration

This definition selects the maximum number of hardware instances that can be supported by the dynamic driver.

Remarks

None.

DRV_PMP_QUEUE_SIZE Macro

PMP queue size for different instances.

File

drv_pmp_config.h

C

```
#define DRV_PMP_QUEUE_SIZE 8
```

Description

PMP queue size

The PMP queue size for a driver instances should be placed here. If more than one driver instance of PMP is present, then all takes the same queue size.

Remarks

All the transfers (Read/Write) first gets queued and gets completed sequentially when Task API is called in a loop. Therefore, the minimum value of this index should be 1.

Building the Library

This section lists the files that are available in the PMP Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/pmp`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_pmp.h`	This file provides the interface definitions of the PMP driver

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_pmp_dynamic.c`	This file contains the core implementation of the PMP driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The PMP Driver Library depends on the following modules:

- PMP Peripheral Library
- Interrupt System Service Library

Library Interface

a) System Functions

	Name	Description
	DRV_PMP_Deinitialize	Deinitializes the specified instance of the PMP driver module. **Implementation:** Dynamic
	DRV_PMP_Initialize	Initializes the PMP driver. **Implementation:** Static/Dynamic
	DRV_PMP_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings. **Implementation:** Dynamic
	DRV_PMP_Status	Provides the current status of the PMP driver module. **Implementation:** Dynamic
	DRV_PMP_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Dynamic
	DRV_PMP_TimingSet	Sets PMP timing parameters. **Implementation:** Static

b) Client Interaction Functions

	Name	Description
	DRV_PMP_ClientStatus	Gets the current client-specific status of the PMP driver. **Implementation:** Dynamic
	DRV_PMP_Close	Closes an opened instance of the PMP driver. **Implementation:** Dynamic
	DRV_PMP_ModeConfig	Configures the PMP modes. **Implementation:** Static/Dynamic
	DRV_PMP_Open	Opens the specified PMP driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_PMP_Read	Read the data from external device. **Implementation:** Static/Dynamic
	DRV_PMP_Write	Transfers the data from the MCU to the external device. **Implementation:** Static/Dynamic

c) Client Transfer Functions

	Name	Description
	DRV_PMP_TransferStatus	Returns the transfer status. **Implementation:** Dynamic

e) Data Types and Constants

	Name	Description
	DRV_PMP_INDEX_COUNT	Number of valid PMP driver indices.
	DRV_PMP_CHIPX_STROBE_MODE	PMP writeEnable/ReadWrite strobes.
	DRV_PMP_CLIENT_STATUS	PMP client status definitions.
	DRV_PMP_ENDIAN_MODE	PMP Endian modes.
	DRV_PMP_INDEX	PMP driver index definitions.
	DRV_PMP_INIT	Defines the PMP driver initialization data.
	DRV_PMP_MODE_CONFIG	PMP modes configuration.
	DRV_PMP_POLARITY_OBJECT	PMP polarity object.
	DRV_PMP_PORT_CONTROL	PMP port enable/disable definitions.
	DRV_PMP_PORTS	PMP port configuration.
	DRV_PMP_QUEUE_ELEMENT_OBJ	Defines the object for PMP queue element.
	_DRV_PMP_QUEUE_ELEMENT_OBJ	Defines the object for PMP queue element.
	DRV_PMP_TRANSFER_STATUS	Defines the PMP transfer status.
	DRV_PMP_WAIT_STATES	PMP wait states object.
	MAX_NONBUFFERED_BYTE_COUNT	After this number the PMP transfer should be polled to guarantee data transfer
	DRV_PMP_TRANSFER_TYPE	This is type DRV_PMP_TRANSFER_TYPE.
	PMP_QUEUE_ELEMENT_OBJECT	Defines the structure required for maintaining the queue element.

Description

This section describes the Application Programming Interface (API) functions of the PMP Driver.

Refer to each section for a detailed description.

a) System Functions

DRV_PMP_Deinitialize Function

Deinitializes the specified instance of the PMP driver module.

Implementation: Dynamic

File

drv_pmp.h

C

```
void DRV_PMP_Deinitialize(const SYS_MODULE_OBJ pmpDriverObject);
```

Returns

None.

Description

This function deinitializes the specified instance of the PMP driver module, disabling its operation (and any hardware). All internal data is invalidated.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

This function will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_PMP_Status operation. The system has to use DRV_PMP_Status to find out when the module is in the ready state.

Preconditions

The DRV_PMP_Initialize function must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
SYS_MODULE_OBJ      pmpDriverObject;     //  Returned from DRV_PMP_Initialize
SYS_STATUS          status;

DRV_PMP_Deinitialize(pmpDriverObject);

status = DRV_PMP_Status(pmpDriverObject);
if (SYS_MODULE_DEINITIALIZED == status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
pmpDriverObject	Driver object handle, returned from the DRV_PMP_Initialize

Function

void DRV_PMP_Deinitialize (SYS_MODULE_OBJ pmpDriverObject)

DRV_PMP_Initialize Function

Initializes the PMP driver.

Implementation: Static/Dynamic

File

drv_pmp.h

C

```
SYS_MODULE_OBJ DRV_PMP_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init);
```

Returns

If successful, it returns a valid handle to a driver object. Otherwise, it returns SYS_MODULE_OBJ_INVALID. The returned object must be passed as argument to DRV_PMP_Reinitialize, DRV_PMP_Deinitialize, DRV_PMP_Tasks and DRV_PMP_Status routines.

Description

This function initializes the PMP driver, making it ready for clients to open and use it.

Remarks

This function must be called before any other PMP function is called.

This function should only be called once during system initialization unless DRV_PMP_Deinitialize is called to deinitialize the driver instance.

This function will NEVER block for hardware access. If the operation requires time to allow the hardware to reinitialize, it will be reported by the DRV_PMP_Status operation. The system must use DRV_PMP_Status to find out when the driver is in the ready state.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

None.

Example

```
DRV_PMP_INIT    init;
SYS_MODULE_OBJ  objectHandle;

// Populate the initialization structure
init.inputBuffer = PMP_INPUT_BUFFER_TTL;
init.polarity.addressLatchPolarity = PMP_POLARITY_ACTIVE_HIGH;
init.polarity.rwStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
init.polarity.writeEnableStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
init.polarity.chipselect1Polarity = PMP_POLARITY_ACTIVE_HIGH;
init.polarity.chipselect2Polarity = PMP_POLARITY_ACTIVE_LOW;
init.ports.addressPortsMask = PMP_PMA0_PORT | PMP_PMA1_PORT | PMP_PMA2_TO_PMA13_PORTS | PMP_PMA14_PORT;
init.ports.readWriteStrobe = PORT_ENABLE;
init.ports.writeEnableStrobe = PORT_ENABLE;
```

```
init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
init.pmpID            = PMP_ID_0;
init.stopInIdle       = false;
init.muxMode          = PMP_MUX_NONE;

// Do something

objectHandle = DRV_PMP_Initialize(DRV_PMP_INDEX_0,  (SYS_MODULE_INIT*)&init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Index for the driver instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the driver

Function

SYS_MODULE_OBJ DRV_PMP_Initialize(const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT * const init)

DRV_PMP_Reinitialize Function

Reinitializes the driver and refreshes any associated hardware settings.

Implementation: Dynamic

File

drv_pmp.h

C

```
void DRV_PMP_Reinitialize(const SYS_MODULE_OBJ pmpDriverObject, const SYS_MODULE_INIT * const init);
```

Returns

None.

Description

This function reinitializes the driver and refreshes any associated hardware settings using the specified initialization data, but it will not interrupt any ongoing operations.

Remarks

This function can be called multiple times to reinitialize the module.

This operation can be used to refresh any supported hardware registers as specified by the initialization data or to change the power state of the module.

This function will NEVER block for hardware access. If the operation requires time to allow the hardware to re-initialize, it will be reported by the DRV_PMP_Status operation. The system must use DRV_PMP_Status to find out when the driver is in the ready state.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

The DRV_PMP_Initialize function must have been called before calling this function and a valid SYS_MODULE_OBJ must have been returned.

Example

```
DRV_PMP_INIT    init;
SYS_MODULE_OBJ  pmpDriverObject;
SYS_STATUS  pmpStatus;

// Populate the initialization structure
init.inputBuffer = PMP_INPUT_BUFFER_TTL;
init.polarity.addressLatchPolarity = PMP_POLARITY_ACTIVE_HIGH;
init.polarity.rwStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
init.polarity.writeEnableStrobePolarity = PMP_POLARITY_ACTIVE_LOW;
```

```
init.polarity.chipselect1Polarity = PMP_POLARITY_ACTIVE_HIGH;
init.polarity.chipselect2Polarity = PMP_POLARITY_ACTIVE_LOW;
init.ports.addressPortsMask = PMP_PMA0_PORT | PMP_PMA1_PORT | PMP_PMA2_TO_PMA13_PORTS | PMP_PMA14_PORT;
init.ports.readWriteStrobe = PORT_ENABLE;
init.ports.writeEnableStrobe = PORT_ENABLE;
init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
init.pmpID            = PMP_ID_0;
init.stopInIdle       = false;
init.muxMode          = PMP_MUX_NONE;

DRV_PMP_Reinitialize(pmpDriverObject, (SYS_MODULE_INIT*)&init);

pmpStatus = DRV_PMP_Status(pmpDriverObject);
if (SYS_STATUS_BUSY == pmpStatus)
{
    // Check again later to ensure the driver is ready
}
else if (SYS_STATUS_ERROR >= pmpStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
pmpDriverObject	Driver object handle, returned from the DRV_PMP_Initialize

Function

void DRV_PMP_Reinitialize (SYS_MODULE_OBJ pmpDriverObject,
const SYS_MODULE_INIT * const init)

init - Pointer to the initialization data structure

DRV_PMP_Status Function

Provides the current status of the PMP driver module.

Implementation: Dynamic

File

drv_pmp.h

C

```
SYS_STATUS DRV_PMP_Status(const SYS_MODULE_OBJ pmpDriverObject);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system level operation and cannot start another

Description

This function provides the current status of the PMP driver module.

Remarks

Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_STATUS_BUSY - Indicates that the driver is busy with a previous system level operation and cannot start another

SYS_STATUS_ERROR - Indicates that the driver is in an error state

Any value less than SYS_STATUS_ERROR is also an error state.

SYS_MODULE_DEINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR.

This operation can be used to determine when any of the driver's module level operations has completed.

If the status operation returns SYS_STATUS_BUSY, a previous operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This function will NEVER block waiting for hardware.

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The DRV_PMP_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      pmpDriverObject;     // Returned from DRV_PMP_Initialize
SYS_STATUS          status;

status = DRV_PMP_Status(pmpDriverObject);
else if (SYS_STATUS_ERROR >= status)
{
    // Handle error
}
```

Parameters

Parameters	Description
pmpDriverObject	Driver object handle, returned from the DRV_PMP_Initialize routine

Function

SYS_STATUS DRV_PMP_Status (SYS_MODULE_OBJ pmpDriverObject)

DRV_PMP_Tasks Function

Maintains the driver's state machine and implements its ISR.

Implementation: Dynamic

File

drv_pmp.h

C

```
void DRV_PMP_Tasks(SYS_MODULE_OBJ pmpDriverObject);
```

Returns

None.

Description

This function is used to maintain the queue and execute the tasks stored in the queue. It resides in the ISR of the PMP for interrupt-driven implementations.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This function may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_PMP_Initialize function must have been called for the specified PMP driver instance.

Example

```
SYS_MODULE_OBJ      pmpDriverObject;     // Returned from DRV_PMP_Initialize

while (true)
{
    DRV_PMP_Tasks (pmpDriverObject);

    // Do other tasks
}
```

Parameters

Parameters	Description
pmpDriverObject	Object handle for the specified driver instance (returned from DRV_PMP_Initialize)

Function

void DRV_PMP_Tasks (SYS_MODULE_OBJ pmpDriverObject);

DRV_PMP_TimingSet Function

Sets PMP timing parameters.

Implementation: Static

File

drv_pmp.h

C

```
void DRV_PMP_TimingSet(PMP_DATA_WAIT_STATES dataWait, PMP_STROBE_WAIT_STATES strobeWait,
PMP_DATA_HOLD_STATES dataHold);
```

Returns

None.

Description

This function sets the PMP timing parameters.

Remarks

None.

Preconditions

The DRV_PMP_Initialize function must have been called.

Example

```
DRV_PMP0_TimingSet(PMP_DATA_WAIT_THREE,PMP_STROBE_WAIT_6,PMP_DATA_HOLD_4);
```

Parameters

Parameters	Description
dataWait	Data setup to read/write strobe wait states
strobeWait	Read/write strobe wait states
dataHold	Data hold time after read/write strobe wait states

Function

void DRV_PMP_TimingSet(
PMP_DATA_WAIT_STATES dataWait,
PMP_STROBE_WAIT_STATES strobeWait,
PMP_DATA_HOLD_STATES dataHold
)

b) Client Interaction Functions

DRV_PMP_ClientStatus Function

Gets the current client-specific status of the PMP driver.

Implementation: Dynamic

File

drv_pmp.h

C

```
DRV_PMP_CLIENT_STATUS DRV_PMP_ClientStatus(DRV_HANDLE hClient);
```

Returns

A DRV_PMP_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the PMP driver associated with the specified handle.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_PMP_Initialize routine must have been called.

DRV_PMP_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE hClient;  // Returned from DRV_PMP_Open
DRV_PMP_CLIENT_STATUS pmpClientStatus;

pmpClientStatus = DRV_PMP_ClientStatus(hClient);
if(DRV_PMP_CLIENT_STATUS_ERROR >= pmpClientStatus)
{
    // Handle the error
}
```

Parameters

Parameters	Description
hClient	A valid open-instance handle, returned from the driver's open routine

Function

DRV_PMP_CLIENT_STATUS DRV_PMP_ClientStatus (DRV_HANDLE hClient)

DRV_PMP_Close Function

Closes an opened instance of the PMP driver.

Implementation: Dynamic

File

drv_pmp.h

C

```
void DRV_PMP_Close(const DRV_HANDLE hClient);
```

Returns

None

Description

This function closes an opened instance of the PMP driver, invalidating the handle.

Remarks

After calling this function, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_PMP_Open before the caller may use the driver again.

If DRV_IO_INTENT_BLOCKING was requested and the driver was built appropriately to support blocking behavior call may block until the operation is complete.

If DRV_IO_INTENT_NON_BLOCKING request the driver client can call the DRV_PMP_Status operation to find out when the module is in the ready state (the handle is no longer valid).

Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_PMP_Initialize routine must have been called for the specified PMP driver instance.

DRV_PMP_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE hClient;  // Returned from DRV_PMP_Open

DRV_PMP_Close(hClient);
```

Parameters

Parameters	Description
hClient	A valid open instance handle, returned from the driver's open routine

Function

void DRV_PMP_Close (DRV_HANDLE hClient)

DRV_PMP_ModeConfig Function

Configures the PMP modes.

Implementation: Static/Dynamic

File

drv_pmp.h

C

```
void DRV_PMP_ModeConfig(DRV_HANDLE hClient, DRV_PMP_MODE_CONFIG config);
```

Returns

None.

Description

This function configures the modes for client in which it wants to operate. Different master-slave modes, 8/16 data bits selection, address increment/decrement, interrupt mode, wait states, etc., can be configured through this function.

Remarks

This function will NEVER block waiting for hardware. If this API is called more than once for a particular client handle, previous config setting of that client will be overwritten.

Preconditions

Function DRV_PMP_Initialize must have been called. DRV_PMP_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE  hClient;
DRV_PMP_MODE_CONFIG config;

config.chipSelect = PMCS1_AND_PMCS2_AS_CHIP_SELECT;
config.endianMode = LITTLE_ENDIAN;
config.incrementMode = PMP_ADDRESS_AUTO_INCREMENT;
config.intMode =  PMP_INTERRUPT_NONE;
config.pmpMode = PMP_MASTER_READ_WRITE_STROBES_INDEPENDENT;
config.portSize =  PMP_DATA_SIZE_8_BITS;
config.waitStates.dataHoldWait = PMP_DATA_HOLD_2;
config.waitStates.dataWait = PMP_DATA_WAIT_THREE;
config.waitStates.strobeWait = PMP_STROBE_WAIT_5;

DRV_PMP_ModeConfig ( hClient, config );
```

Parameters

Parameters	Description
hClient	Client handle obtained from DRV_PMP_Open API
config	Structure which will have all the required PMP modes configuration

Function

void DRV_PMP_ModeConfig (DRV_HANDLE hClient,
DRV_PMP_MODE_CONFIG config)

DRV_PMP_Open Function

Opens the specified PMP driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_pmp.h

C

```
DRV_HANDLE DRV_PMP_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Returns

If successful, the function returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID.

Description

This function opens the specified PMP driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_PMP_Close routine is called.

This function will NEVER block waiting for hardware.

If the DRV_IO_INTENT_BLOCKING is requested and the driver was built appropriately to support blocking behavior, other client-level operations may block waiting on hardware until they are complete.

If DRV_IO_INTENT_NON_BLOCKING is requested the driver client can call the DRV_PMP_ClientStatus operation to find out when the module is in the ready state.

If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID.

Preconditions

The DRV_PMP_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  hClient;

hClient = DRV_PMP_Open(DRV_PMP_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == hClient)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT ORed together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_PMP_Open (const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT intent)

DRV_PMP_Read Function

Read the data from external device.

Implementation: Static/Dynamic

File

drv_pmp.h

C

```
PMP_QUEUE_ELEMENT_OBJECT* DRV_PMP_Read(DRV_HANDLE hClient, uint32_t address, uint16_t* buffer, uint32_t
```

```
nBytes);
```

Returns

Returns the position number of the queue, where the data element was stored. Returns '0' when there is no place in the queue to store the data.

Description

This function reads the given number of data bytes from the given address of the external device to the MCU buffer through the selected PMP instance. This function should be used for all the master and slave modes. Proper configuration should be done using DRV_PMP_ModeConfig before calling this function.

Preconditions

The DRV_PMP_Initialize routine must have been called. DRV_PMP_Open must have been called to obtain a valid opened device handle. DRV_PMP_ModeConfig must have been called to configure the desired mode

Example

```
DRV_HANDLE hClient;  // Returned from DRV_PMP_Open
uint32_t deviceAddress;
uint32_t nBytes;
unsigned char myBuffer[nBytes];
uint32_t transferID;

transferID = DRV_PMP_Read ( hClient, deviceAddress, &myBuffer, nBytes);
```

Parameters

Parameters	Description
hClient	A valid open-instance handle, returned from the driver's open routine
address	Starting address of the slave device from where data has to be read. It does not have any significance for legacy slave mode and buffer mode. In PMP enhanced slave mode i.e. addressable buffer slave mode, this parameter should be the buffer number to be used.
buffer	Pointer to the buffer into which the data read through the PMP instance will be placed. Even if only one word has to be transferred, pointer should be used.
nBytes	Number of bytes that need to be read through the PMP instance

Function

uint32_t DRV_PMP_Read (DRV_HANDLE hClient,

uint32_t address,

unsigned char* buffer,

uint32_t nBytes)

DRV_PMP_Write Function

Transfers the data from the MCU to the external device.

Implementation: Static/Dynamic

File

drv_pmp.h

C

```
PMP_QUEUE_ELEMENT_OBJECT* DRV_PMP_Write(DRV_HANDLE* hClient, bool address, uint32_t * buffer, uint32_t
nBytes, uint32_t repeatCount);
```

Returns

Returns a 32-bit ID with which status of the transfer can be checked later. Returns '0' when there is no place in the queue to store the data.

Description

This function transfer the given number of data bytes from the MCU buffer location to the defined address of the external device through the selected PMP instance. It repeats the operation n (=repeatCount) number of times as well. This function should be used for all the master and slave modes. Proper configuration should be done using DRV_PMP_ModeConfig before calling this function.

Preconditions

The DRV_PMP_Initialize routine must have been called. DRV_PMP_Open must have been called to obtain a valid opened device handle. DRV_PMP_ModeConfig must have been called to configure the desired mode.

Example

```
DRV_HANDLE hClient;  // Returned from DRV_PMP_Open
uint32_t deviceAddress;
uint32_t nBytes;
unsigned char myBuffer[nBytes];
uint32_t repeatCount;
uint32_t transferID;

transferID = DRV_PMP_MasterWrite ( hClient, deviceAddress, &myBuffer, nBytes, repeatCount);
```

Parameters

Parameters	Description
hClient	A valid open-instance handle, returned from the driver's open routine
address	Starting address of the slave device where data has to be written. It does not have any significance for legacy slave mode and buffer mode. In PMP enhanced slave mode (i.e., addressable buffer slave mode), this parameter should be the buffer number to be used.
buffer	Pointer to MCU Buffer from which the data will be written through the PMP instance. even if only one word has to be transferred, pointer should be used.
nBytes	Total number of bytes that need to be written through the PMP instance
repeatCount	Number of times the data set (nBytes of data) to be repeatedly written. This value should be 0 if user does not want any repetition. If repeatCount is greater than 0, then after writing every nBytes of data, the buffer starts pointing to its first element. Ideally, PMP Address should be in auto increment/decrement mode for repeatCount greater than 0.

Function

uint32_t DRV_PMP_Write (DRV_HANDLE hClient,

uint32_t address,

unsigned char* buffer,

uint32_t nBytes,

uint32_t repeatCount)

c) Client Transfer Functions

DRV_PMP_TransferStatus Function

Returns the transfer status.

Implementation: Dynamic

File

drv_pmp.h

C

```
DRV_PMP_TRANSFER_STATUS DRV_PMP_TransferStatus(PMP_QUEUE_ELEMENT_OBJECT* queueObject);
```

Returns

A DRV_PMP_TRANSFER_STATUS value describing the current status of the transfer.

Description

This function returns the status of a particular transfer whose ID has been specified as input.

Example

```
uint32_8 seqID;
DRV_PMP_TRANSFER_STATUS transferStatus;

transferStatus = DRV_PMP_TransferStatus( DRV_PMP_INDEX_0, seqID);
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened

seqID	A valid ID returned from read/write transfer functions

Function

DRV_PMP_TRANSFER_STATUS DRV_PMP_TransferStatus(DRV_HANDLE hClient)

d) Miscellaneous Functions

e) Data Types and Constants

DRV_PMP_INDEX_COUNT Macro

Number of valid PMP driver indices.

File

drv_pmp.h

C

```
#define DRV_PMP_INDEX_COUNT _PMP_EXISTS
```

Description

PMP Driver Module Index Count

This constant identifies the number of valid PMP driver indices.

Remarks

The value of "_PMP_EXISTS" is derived from device-specific header files defined as part of the peripheral libraries.

DRV_PMP_CHIPX_STROBE_MODE Enumeration

PMP writeEnable/ReadWrite strobes.

File

drv_pmp.h

C

```
typedef enum {
  PMP_RW_STROBE_WITH_ENABLE_STROBE,
  PMP_READ_AND_WRITE_STROBES
} DRV_PMP_CHIPX_STROBE_MODE;
```

Members

Members	Description
PMP_RW_STROBE_WITH_ENABLE_STROBE	One strobe for read/write and another for enable
PMP_READ_AND_WRITE_STROBES	Separate strobes for read and write operations

Description

PMP writeEnable/ReadWrite strobes

This enumeration provides ReadWrite/WriteEnable Strobe definitions.

DRV_PMP_CLIENT_STATUS Enumeration

PMP client status definitions.

File

drv_pmp.h

C

```
typedef enum {
  DRV_PMP_CLIENT_STATUS_INVALID,
  PMP_CLIENT_STATUS_CLOSED,
```

```
  DRV_PMP_CLIENT_STATUS_OPEN
} DRV_PMP_CLIENT_STATUS;
```

Description

PMP Client Status

This enumeration provides various client status possibilities.

DRV_PMP_ENDIAN_MODE Enumeration

PMP Endian modes.

File

drv_pmp.h

C

```
typedef enum {
  LITTLE,
  BIG
} DRV_PMP_ENDIAN_MODE;
```

Members

Members	Description
LITTLE	Little Endian
BIG	Big Endian

Description

PMP Endian modes

This enumeration holds the Endian configuration options.

DRV_PMP_INDEX Enumeration

PMP driver index definitions.

File

drv_pmp.h

C

```
typedef enum {
  DRV_PMP_INDEX_0,
  DRV_PMP_INDEX_1
} DRV_PMP_INDEX;
```

Members

Members	Description
DRV_PMP_INDEX_0	First PMP instance
DRV_PMP_INDEX_1	Second PMP instance (not available for now)

Description

PMP Driver Module Index Numbers

These constants provide PMP driver index definitions.

Remarks

These values should be passed into the DRV_PMP_Initialize and DRV_PMP_Open functions to identify the driver instance in use.

DRV_PMP_INIT Structure

Defines the PMP driver initialization data.

File

drv_pmp.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  PMP_MODULE_ID pmpID;
  bool stopInIdle;
  PMP_MUX_MODE muxMode;
  PMP_INPUT_BUFFER_TYPE inputBuffer;
  DRV_PMP_POLARITY_OBJECT polarity;
  DRV_PMP_PORTS ports;
} DRV_PMP_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	module power state info
PMP_MODULE_ID pmpID;	module PLIB ID
bool stopInIdle;	Stop in Idle enable
PMP_MUX_MODE muxMode;	MUX mode
PMP_INPUT_BUFFER_TYPE inputBuffer;	Input buffer type to be used
DRV_PMP_POLARITY_OBJECT polarity;	Polarity settings
DRV_PMP_PORTS ports;	PMP port settings

Description

PMP Driver Initialize Data

This data type defines data required to initialize or reinitialize the PMP driver.

Remarks

Not all the initialization features are available for all devices.

DRV_PMP_MODE_CONFIG Structure

PMP modes configuration.

File

drv_pmp.h

C

```
typedef struct {
  PMP_OPERATION_MODE pmpMode;
  PMP_INTERRUPT_MODE intMode;
  PMP_INCREMENT_MODE incrementMode;
  DRV_PMP_ENDIAN_MODE endianMode;
  PMP_DATA_SIZE portSize;
  DRV_PMP_WAIT_STATES waitStates;
  PMP_CHIPSELECT_FUNCTION chipSelect;
} DRV_PMP_MODE_CONFIG;
```

Members

Members	Description
PMP_OPERATION_MODE pmpMode;	PMP Usage Mode Type
PMP_INTERRUPT_MODE intMode;	Interrupt mode
PMP_INCREMENT_MODE incrementMode;	should be appropriately selected based on read/write requirements and operation mode setting */ address/buffer increment mode
DRV_PMP_ENDIAN_MODE endianMode;	it does not have any significance in PMP slave mode or 8bit data mode */ Endian modes
PMP_DATA_SIZE portSize;	Data Port Size
DRV_PMP_WAIT_STATES waitStates;	Wait states
PMP_CHIPSELECT_FUNCTION chipSelect;	use this when PLIB is fixed

Description

PMP modes configuration

This data type controls the configuration of PMP modes.

DRV_PMP_POLARITY_OBJECT Structure

PMP polarity object.

File

drv_pmp.h

C

```
typedef struct {
  PMP_POLARITY_LEVEL addressLatchPolarity;
  PMP_POLARITY_LEVEL byteEnablePolarity;
  PMP_POLARITY_LEVEL rwStrobePolarity;
  PMP_POLARITY_LEVEL writeEnableStrobePolarity;
  PMP_POLARITY_LEVEL chipselect1Polarity;
  PMP_POLARITY_LEVEL chipselect2Polarity;
} DRV_PMP_POLARITY_OBJECT;
```

Members

Members	Description
PMP_POLARITY_LEVEL addressLatchPolarity;	Address latch polarity
PMP_POLARITY_LEVEL byteEnablePolarity;	ByteEnable port polarity
PMP_POLARITY_LEVEL rwStrobePolarity;	Read/Write strobe polarity
PMP_POLARITY_LEVEL writeEnableStrobePolarity;	Write/Enable strobe polarity
PMP_POLARITY_LEVEL chipselect1Polarity;	ChipSelect-1 Polarity
PMP_POLARITY_LEVEL chipselect2Polarity;	chipSelect-2 Polarity

Description

PMP polarity object

This structure holds the polarities of different entities to be configured.

DRV_PMP_PORT_CONTROL Enumeration

PMP port enable/disable definitions.

File

drv_pmp.h

C

```
typedef enum {
  PORT_ENABLE,
  PORT_DISABLE
} DRV_PMP_PORT_CONTROL;
```

Members

Members	Description
PORT_ENABLE	Enable the given port
PORT_DISABLE	Disable the given port

Description

PMP port enable/disable.

This enumeration provides port enable/disable values.

DRV_PMP_PORTS Structure

PMP port configuration.

File

drv_pmp.h

C

```
typedef struct {
  PMP_ADDRESS_PORT addressPortsMask;
  PMP_PMBE_PORT byteEnablePort;
  DRV_PMP_PORT_CONTROL readWriteStrobe;
  DRV_PMP_PORT_CONTROL writeEnableStrobe;
} DRV_PMP_PORTS;
```

Members

Members	Description
PMP_ADDRESS_PORT addressPortsMask;	User needs to put the address lines which he wants to use in ORed fashion * Address ports
PMP_PMBE_PORT byteEnablePort;	Byte enable ports
DRV_PMP_PORT_CONTROL readWriteStrobe;	READ/WRITE Strobe PORT
DRV_PMP_PORT_CONTROL writeEnableStrobe;	WRITE/ENABLE strobe port

Description

PMP Ports

This structure holds the ports (including the address ports) to be configured by the application to function as general purpose I/O (GPIO) or part of the PMP.

DRV_PMP_QUEUE_ELEMENT_OBJ Structure

Defines the object for PMP queue element.

File

drv_pmp.h

C

```
typedef struct _DRV_PMP_QUEUE_ELEMENT_OBJ {
  struct _DRV_PMP_CLIENT_OBJ * hClient;
  uint32_t buffer;
  uint16_t* addressBuffer;
  uint32_t nTransfers;
  int32_t nRepeats;
  DRV_PMP_TRANSFER_TYPE type;
} DRV_PMP_QUEUE_ELEMENT_OBJ;
```

Members

Members	Description
struct _DRV_PMP_CLIENT_OBJ * hClient;	handle of the client object returned from open API
uint32_t buffer;	pointer to the buffer holding the transmitted data
uint16_t* addressBuffer;	pointer to the buffer holding the transmitted data
uint32_t nTransfers;	number of bytes to be transferred
int32_t nRepeats;	number of times the data set has to be transferred repeatedly
DRV_PMP_TRANSFER_TYPE type;	PMP Read or Write

Description

PMP Driver Queue Element Object

This defines the object structure for each queue element of PMP. This object gets created for every Read/Write operations APIs.

Remarks

None

DRV_PMP_TRANSFER_STATUS Enumeration

Defines the PMP transfer status.

File

drv_pmp.h

C

```
typedef enum {
  MASTER_8BIT_TRANSFER_IN_PROGRESS = PMP_DATA_SIZE_8_BITS,
  MASTER_16BIT_TRANSFER_IN_PROGRESS = PMP_DATA_SIZE_16_BITS,
  MASTER_8BIT_BUFFER_IN_PROGRESS,
  MASTER_16BIT_BUFFER_IN_PROGRESS,
  MASTER_8BIT_TRANSFER_CONTINUE,
  MASTER_8BIT_BUFFER_CONTINUE,
  QUEUED_BUT_PMP_TRANSFER_NOT_STARTED,
  PMP_TRANSFER_FINISHED
} DRV_PMP_TRANSFER_STATUS;
```

Description

Queue Element Transfer Status

This enumeration defines the PMP transfer status.

DRV_PMP_WAIT_STATES Structure

PMP wait states object.

File

drv_pmp.h

C

```
typedef struct {
  PMP_DATA_HOLD_STATES dataHoldWait;
  PMP_STROBE_WAIT_STATES strobeWait;
  PMP_DATA_WAIT_STATES dataWait;
} DRV_PMP_WAIT_STATES;
```

Members

Members	Description
PMP_DATA_HOLD_STATES dataHoldWait;	data hold wait states
PMP_STROBE_WAIT_STATES strobeWait;	read/write strobe wait states
PMP_DATA_WAIT_STATES dataWait;	data wait strobe wait sates

Description

PMP wait states object

This structure holds the different wait states to be configured. Refer to the PMP PLIB help document for the possible values and meaning of the different wait states.

MAX_NONBUFFERED_BYTE_COUNT Macro

File

drv_pmp.h

C

```
#define MAX_NONBUFFERED_BYTE_COUNT 4 /*************************************************************************
                                      After this number the PMP transfer should be polled to guarantee data
                                      transfer
                                      *************************************************************************

*/
```

Description

After this number the PMP transfer should be polled to guarantee data transfer

DRV_PMP_TRANSFER_TYPE Enumeration

File

drv_pmp.h

C

```
typedef enum {
  ADDRESS,
  READ,
  WRITE,
  BUFFERED_WRITE
} DRV_PMP_TRANSFER_TYPE;
```

Members

Members	Description
ADDRESS	PMP Address needs to be updated
READ	PMP Read Transfer
WRITE	PMP Write Transfer
BUFFERED_WRITE	PMP Array Write Transfer

Description

This is type DRV_PMP_TRANSFER_TYPE.

PMP_QUEUE_ELEMENT_OBJECT Structure

Defines the structure required for maintaining the queue element.

File

drv_pmp.h

C

```
typedef struct {
  DRV_PMP_QUEUE_ELEMENT_OBJ data;
  DRV_PMP_TRANSFER_STATUS eTransferStatus;
  uint32_t nTransfersDone;
} PMP_QUEUE_ELEMENT_OBJECT;
```

Members

Members	Description
DRV_PMP_QUEUE_ELEMENT_OBJ data;	The PMP Q Element
DRV_PMP_TRANSFER_STATUS eTransferStatus;	Flag to indicate that the element is in use
uint32_t nTransfersDone;	sequence id

Description

Queue Element Object

This defines the structure required for maintaining the queue element.

Remarks

None

Files

Files

Name	Description
drv_pmp.h	Parallel Master Port (PMP) device driver interface file.
drv_pmp_config.h	PMP driver configuration definitions template

Description

This section lists the source and header files used by the PMP Driver Library.

drv_pmp.h

Parallel Master Port (PMP) device driver interface file.

Enumerations

	Name	Description
	DRV_PMP_CHIPX_STROBE_MODE	PMP writeEnable/ReadWrite strobes.
	DRV_PMP_CLIENT_STATUS	PMP client status definitions.
	DRV_PMP_ENDIAN_MODE	PMP Endian modes.
	DRV_PMP_INDEX	PMP driver index definitions.
	DRV_PMP_PORT_CONTROL	PMP port enable/disable definitions.
	DRV_PMP_TRANSFER_STATUS	Defines the PMP transfer status.
	DRV_PMP_TRANSFER_TYPE	This is type DRV_PMP_TRANSFER_TYPE.

Functions

	Name	Description
	DRV_PMP_ClientStatus	Gets the current client-specific status of the PMP driver. **Implementation:** Dynamic
	DRV_PMP_Close	Closes an opened instance of the PMP driver. **Implementation:** Dynamic
	DRV_PMP_Deinitialize	Deinitializes the specified instance of the PMP driver module. **Implementation:** Dynamic
	DRV_PMP_Initialize	Initializes the PMP driver. **Implementation:** Static/Dynamic
	DRV_PMP_ModeConfig	Configures the PMP modes. **Implementation:** Static/Dynamic
	DRV_PMP_Open	Opens the specified PMP driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_PMP_Read	Read the data from external device. **Implementation:** Static/Dynamic
	DRV_PMP_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings. **Implementation:** Dynamic
	DRV_PMP_Status	Provides the current status of the PMP driver module. **Implementation:** Dynamic
	DRV_PMP_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Dynamic
	DRV_PMP_TimingSet	Sets PMP timing parameters. **Implementation:** Static
	DRV_PMP_TransferStatus	Returns the transfer status. **Implementation:** Dynamic
	DRV_PMP_Write	Transfers the data from the MCU to the external device. **Implementation:** Static/Dynamic

Macros

	Name	Description
	DRV_PMP_INDEX_COUNT	Number of valid PMP driver indices.
	MAX_NONBUFFERED_BYTE_COUNT	After this number the PMP transfer should be polled to guarantee data transfer

Structures

	Name	Description
	_DRV_PMP_QUEUE_ELEMENT_OBJ	Defines the object for PMP queue element.
	DRV_PMP_INIT	Defines the PMP driver initialization data.
	DRV_PMP_MODE_CONFIG	PMP modes configuration.
	DRV_PMP_POLARITY_OBJECT	PMP polarity object.
	DRV_PMP_PORTS	PMP port configuration.
	DRV_PMP_QUEUE_ELEMENT_OBJ	Defines the object for PMP queue element.
	DRV_PMP_WAIT_STATES	PMP wait states object.
	PMP_QUEUE_ELEMENT_OBJECT	Defines the structure required for maintaining the queue element.

Description

PMP Device Driver Interface

The PMP device driver provides a simple interface to manage the Parallel Master and Slave ports. This file defines the interface definitions and prototypes for the PMP driver.

File Name

drv_pmp.h

Company

Microchip Technology Inc.

drv_pmp_config.h

PMP driver configuration definitions template

Macros

	Name	Description
	DRV_PMP_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_PMP_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_PMP_QUEUE_SIZE	PMP queue size for different instances.

Description

PMP Driver Configuration Definitions for the Template Version

These definitions statically define the driver's mode of operation.

File Name

drv_pmp_config_template.h

Company

Microchip Technology Inc.

RTCC Driver Library

This section describes the RTCC Driver Library.

Introduction

The Real-Time Clock Calendar (RTCC) Static Driver provides a high-level interface to manage the RTCC module on the Microchip family of microcontrollers.

Description

Through the MHC, this driver provides APIs for the following:

- Initializing the module
- Starting/Stopping the RTCC
- Status functions to yield the date/time
- Status functions to yield the alarm date/time
- Clock output control

Library Interface

System Interaction Functions

	Name	Description
	DRV_RTCC_AlarmDateGet	Gets the Alarm Date of the RTCC. **Implementation:** Static
	DRV_RTCC_AlarmTimeGet	Gets the Alarm Time of the RTCC. **Implementation:** Static

	DRV_RTCC_ClockOutput	Enables Clock Output for the RTCC. **Implementation:** Static
	DRV_RTCC_DateGet	Gets the Date of the RTCC. **Implementation:** Static
	DRV_RTCC_Initialize	Initializes the RTCC instance for the specified driver index. **Implementation:** Static
	DRV_RTCC_Start	Starts the RTCC. **Implementation:** Static
	DRV_RTCC_Stop	Stops the RTCC. **Implementation:** Static
	DRV_RTCC_TimeGet	Gets the time of the RTCC. **Implementation:** Static

Description

This section describes the Application Programming Interface (API) functions of the RTCC Driver Library.

System Interaction Functions

DRV_RTCC_AlarmDateGet Function

Gets the Alarm Date of the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
uint32_t DRV_RTCC_AlarmDateGet();
```

Returns

uint32_t alarm date value

Description

This routine gets the RTCC alarm date.

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

uint32_t DRV_RTCC_AlarmDateGet(void)

DRV_RTCC_AlarmTimeGet Function

Gets the Alarm Time of the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
uint32_t DRV_RTCC_AlarmTimeGet();
```

Returns

uint32_t alarm time value

Description

This routine gets the RTCC alarm time.

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

uint32_t DRV_RTCC_AlarmTimeGet(void)

DRV_RTCC_ClockOutput Function

Enables Clock Output for the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
void DRV_RTCC_ClockOutput();
```

Returns

None.

Description

This routine enables the clock output for the RTCC

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

void DRV_RTCC_ClockOutput(void)

DRV_RTCC_DateGet Function

Gets the Date of the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
uint32_t DRV_RTCC_DateGet();
```

Returns

uint32_t date value

Description

This routine gets the RTCC date.

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

uint32_t DRV_RTCC_DateGet(void)

DRV_RTCC_Initialize Function

Initializes the RTCC instance for the specified driver index.

Implementation: Static

File

help_drv_rtcc.h

C

```
void DRV_RTCC_Initialize();
```

Returns

None.

Description

This routine initializes the RTCC driver instance for the specified driver instance, making it ready for clients to use it. The initialization routine is specified by the MHC parameters.

Remarks

This routine must be called before any other RTCC routine is called. This routine should only be called once during system initialization.

Preconditions

None.

Function

void DRV_RTCC_Initialize(void)

DRV_RTCC_Start Function

Starts the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
void DRV_RTCC_Start();
```

Returns

None.

Description

This routine starts the RTCC, making it ready for clients to use it.

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

void DRV_RTCC_Start(void)

DRV_RTCC_Stop Function

Stops the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
void DRV_RTCC_Stop();
```

Returns

None.

Description

This routine stops the RTCC.

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

void DRV_RTCC_Stop(void)

DRV_RTCC_TimeGet Function

Gets the time of the RTCC.

Implementation: Static

File

help_drv_rtcc.h

C

```
uint32_t DRV_RTCC_TimeGet();
```

Returns

uint32_t time value

Description

This routine gets the RTCC time.

Remarks

None.

Preconditions

DRV_RTCC_Initialize has been called.

Function

uint32_t DRV_RTCC_TimeGet(void)

Secure Digital (SD) Card Driver Library

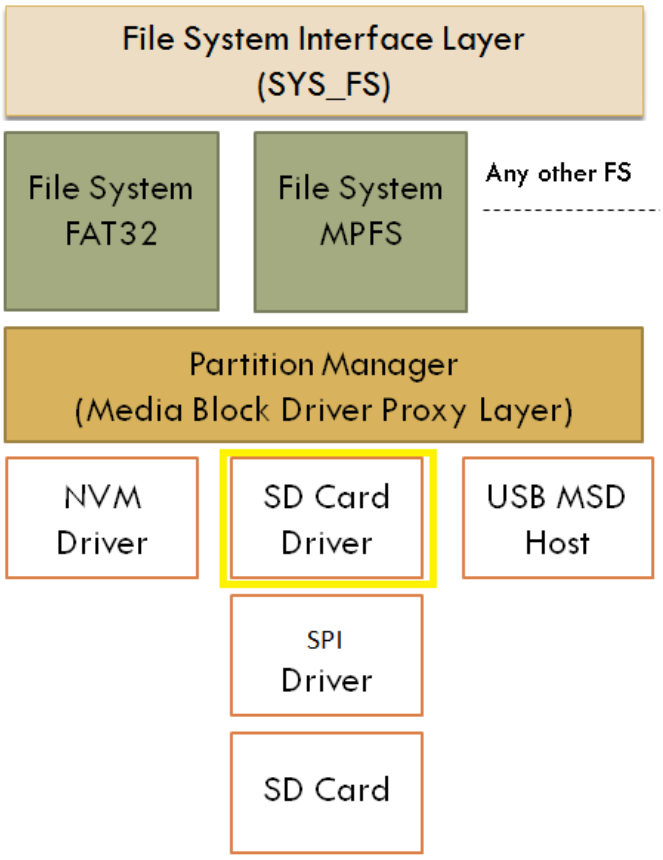
This section describes the Secure Digital (SD) Card Driver Library.

Introduction

The SD Card driver provides the necessary interfaces to interact with an SD card. It provides the necessary abstraction for the higher layer.

Description

A SD Card is a non-volatile memory (Flash memory) card designed to provide high-capacity memory in a small size. Its applications include digital video camcorders, digital cameras, handheld computers, audio players, and mobile phones.



Using the Library

This topic describes the basic architecture of the SD Card Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_sdcard.h`

The interface to the SD Card Driver library is defined in the `drv_sdcard.h` header file. This file is included by the `drv.h` file. Any C language source (`.c`) file that uses the SD Card Driver library should include `drv.h`.

Please refer to the What is MPLAB Harmony? section for how the Driver interacts with the framework.

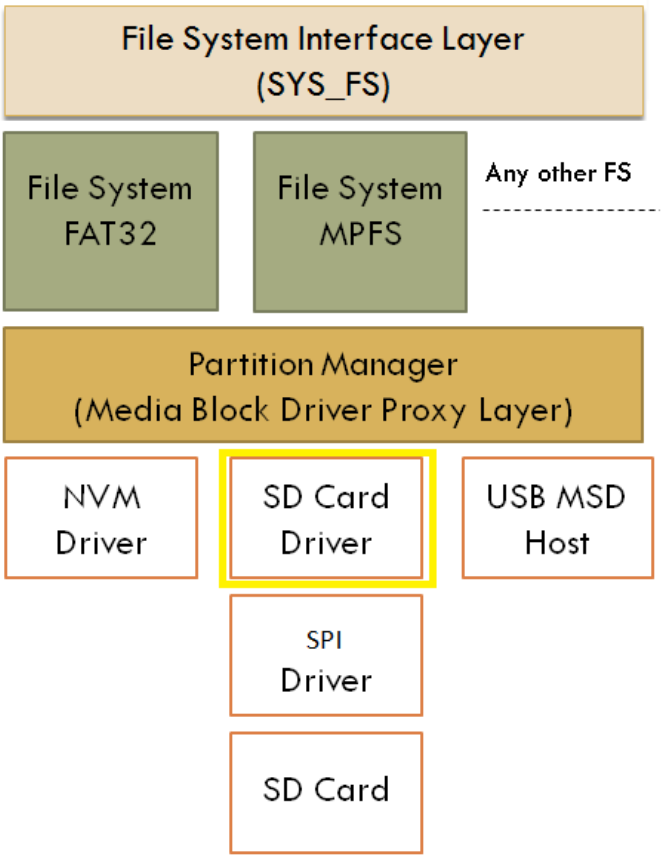
Abstraction Model

This library provides a low-level abstraction of the SD Card Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The SD Card driver comes in the layer below the Partition Manager in the MPLAB Harmony file system architecture and it uses the SPI Driver to interact with the SD card.

SD Card Driver Software Abstraction Block Diagram



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the SD Card module.

Library Interface Section	Description
System Level Functions	Includes functions for initialize the module.
Client Level Functions	Functions to open and close a client.
Operation Functions	Functions for read and write operations
Module Information Functions	Functions for information about the module.
Version Information Functions	Functions to get the software version.

How the Library Works

This section describes how the SD Card Driver Library operates.

Description

Note: Not all modes are available on all devices. Please refer to the specific device data sheet to determine the supported modes.

The library provides interfaces that support:

- Driver Initialization Functionality
- Client Block Data Functionality
- Client Access Functionality

SD Card Driver Initialization

This section provides information for system initialization and reinitialization.

Description

The system performs the initialization and the reinitialization of the device driver with settings that affect only the instance of the device that is being initialized or reinitialized. During system initialization each instance of the SD Card module would be initialized with the following configuration settings (either passed dynamically at run time using DRV_SDCARD_INIT or by using initialization overrides) that are supported by the specific SD Card device hardware:

- SPI Peripheral ID: Identifies the SPI Peripheral ID to be used for the SD Card Driver
- SPI Index: SPI Driver Index
- SD Card frequency: SD Card communication speed
- SPI Clock source: Peripheral clock used by the SPI
- Write-Protect Port: Port used to check if the SD Card is write protected
- Write-Protect Pin: Pin used to check if the SD Card is write protected
- Chip Select Port: Port used for the SPI Chip Select
- Chip Select Pin: Pin used for the SPI Chip Select

The DRV_SDCARD_Initialize function returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the initialize interface would be used by the other system interfaces, such as DRV_SDCARD_Deinitialize, DRV_SDCARD_Status, and DRV_SDCARD_Tasks.

Note: The system initialization and the reinitialization settings, only affect the instance of the peripheral that is being initialized or reinitialized.

Example:

```
const DRV_SDCARD_INIT drvSDCardInit =
{
.spiId = SPI_ID_2,
.spiIndex = 0,
.sdcardSpeedHz = 20000000,
.spiClk = CLK_BUS_PERIPHERAL_2,
.writeProtectPort = PORT_CHANNEL_F,
.writeProtectBitPosition = PORTS_BIT_POS_1,
.chipSelectPort = PORT_CHANNEL_B,
.chipSelectBitPosition = PORTS_BIT_POS_14,
};

void SYS_Initialize (void *data)
{
.
.
sysObj.drvSDCard = DRV_SDCARD_Initialize(DRV_SDCARD_INDEX_0,(SYS_MODULE_INIT *)&drvSDCardInit);
.
.
}
```

Tasks Routine

The system will call DRV_SDCARD_Tasks, from system task service to maintain the driver's state machine.

Client Access Operation

This section provides information for general client operation.

Description

General Client Operation

For the application to start using an instance of the module, it must call the DRV_SDCARD_Open function. This provides the configuration required to open the SD Card instance for operation. If the driver is deinitialized using the function DRV_SDCARD_Deinitialize, the application must call the DRV_SDCARD_Open function again to set up the instance of the SDCARD.

For the various options available for I/O INTENT please refer to Data Types and Constants in the Library Interface section.

Example:

```
DRV_HANDLE handle;
handle = DRV_SDCARD_Open(DRV_SDCARD_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
```

```
if (DRV_HANDLE_INVALID == handle)
{
// Unable to open the driver
}
```

Client Block Data Operation

This topic provides information on client block data operation.

Description

The SDCARD Driver provides a block interface to access the SD Card. The interface provides functionality to read from and write to the SD Card.

Reading Data from the SD Card:

The following steps outline the sequence to be followed for reading data from the SD Card:

1. The system should have completed necessary initialization and DRV_SDCARD_Tasks should either be running in a polled environment, or in an interrupt environment.
2. The driver should have been opened with the necessary intent.
3. Invoke the DRV_SDCARD_Read function and pass the pointer where the data is to be stored, block start address and the number of blocks of data to be read.
4. The client should validate the command handle returned by the DRV_SDCARD_Read function. DRV_SDCARD_COMMAND_HANDLE_INVALID value indicates that an error has occurred which the client needs to handle.
5. If the request was successful then the client can check the status of the request by invoking the DRV_SDCARD_CommandStatus and passing the command handle returned by the read request. Alternately the client could use the event handler for notifications from the driver.
6. The client will be able to close itself by calling the DRV_SDCARD_Close.

Example:

```
// This code shows how to read data from the SD Card
DRV_HANDLE sdcardHandle;
DRV_SDCARD_COMMAND_HANDLE sdcardCommandHandle;
DRV_SDCARD_COMMAND_STATUS commandStatus;
uint8_t readBuf[512];
uint32_t blockAddress;
uint32_t nBlocks;

/* Initialize the block start address and the number of blocks to be read */
blockAddress = 0;
nBlocks = 1;

DRV_SDCARD_Read(sdcardHandle, &sdcardCommandHandle, (uint8_t *)readBuf, blockAddress, nBlocks);
if(DRV_SDCARD_COMMAND_HANDLE_INVALID == sdcardCommandHandle)
{
    /* Failed to queue the read request. Handle the error. */
}
// Wait until the command completes. This should not
// be a while loop if part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.
commandStatus = DRV_SDCARD_CommandStatus(sdcardHandle, sdcardCommandHandle);
if(DRV_SDCARD_COMMAND_COMPLETED == commandStatus)
{
    /* Read completed */
}
else if (DRV_SDCARD_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Read Failed */
}
```

Writing Data to the SD Card:

The following steps outline the sequence to be followed for writing data to the SD Card:

1. The system should have completed necessary initialization and DRV_SDCARD_Tasks should either be running in a polled environment, or in an interrupt environment.
2. The driver should have been opened with the necessary intent.
3. Invoke the DRV_SDCARD_Write function and pass the pointer to the data to be written, block start address and the number of blocks of data to be written.
4. The client should validate the command handle returned by the DRV_SDCARD_Write function. DRV_SDCARD_COMMAND_HANDLE_INVALID value indicates that an error has occurred which the client needs to handle.
5. If the request was successful then the client can check the status of the request by invoking the DRV_SDCARD_CommandStatus and passing the command handle returned by the write request. Alternately, the client could use the event handler for notifications from the driver.

6. The client will be able to close itself by calling the DRV_SDCARD_Close.

Example:

```
// This code shows how to write data to the SD Card
DRV_HANDLE sdcardHandle;
DRV_SDCARD_COMMAND_HANDLE sdcardCommandHandle;
DRV_SDCARD_COMMAND_STATUS commandStatus;
uint8_t writeBuf[512];
uint32_t blockAddress;
uint32_t nBlocks;

/* Initialize the block start address and the number of blocks to be written */
blockAddress = 0;
nBlocks = 1;
/* Populate writeBuf with the data to be written */

DRV_SDCARD_Write(sdcardHandle, &sdcardCommandHandle, (uint8_t *)writeBuf, blockAddress, nBlocks);
if(DRV_SDCARD_COMMAND_HANDLE_INVALID == sdcardCommandHandle)
{
    /* Failed to queue the write request. Handle the error. */
}
// Wait until the command completes. This should not
// be a while loop if part of cooperative multi-tasking
// routine. In that case, it should be invoked in task
// state machine.
commandStatus = DRV_SDCARD_CommandStatus(sdcardHandle, sdcardCommandHandle);
if(DRV_SDCARD_COMMAND_COMPLETED == commandStatus)
{
    /* Write completed */
}
else if (DRV_SDCARD_COMMAND_ERROR_UNKNOWN == commandStatus)
{
    /* Write Failed */
}
```

Configuring the Library

Macros

	Name	Description
	DRV_SDCARD_CLIENTS_NUMBER	Selects the miximum number of clients
	DRV_SDCARD_INDEX_MAX	SD Card Static Index selection
	DRV_SDCARD_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver
	DRV_SDCARD_POWER_STATE	Defines an override of the power state of the SD Card driver.
	DRV_SDCARD_SYS_FS_REGISTER	Register to use with the File system
	DRV_SDCARD_ENABLE_WRITE_PROTECT_CHECK	Enable SD Card write protect check.

Description

The configuration of the SD Card Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the SD Card Driver. Based on the selections made, the SD Card Driver may support the selected features. These configuration settings will apply to all instances of the SD Card.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_SDCARD_CLIENTS_NUMBER Macro

Selects the miximum number of clients

File

drv_sdcard_config_template.h

C

```
#define DRV_SDCARD_CLIENTS_NUMBER 1
```

Description

SD Card Maximum Number of Clients

This definition select the maximum number of clients that the SD Card driver can support at run time. Not defining it means using a single client.

Remarks

None.

DRV_SDCARD_INDEX_MAX Macro

SD Card Static Index selection

File

drv_sdcard_config_template.h

C

```
#define DRV_SDCARD_INDEX_MAX 1
```

Description

SD Card Static Index Selection

SD Card Static Index selection for the driver object reference

Remarks

This index is required to make a reference to the driver object

DRV_SDCARD_INSTANCES_NUMBER Macro

Selects the maximum number of hardware instances that can be supported by the dynamic driver

File

drv_sdcard_config_template.h

C

```
#define DRV_SDCARD_INSTANCES_NUMBER 1
```

Description

SD Card hardware instance configuration

This definition selects the maximum number of hardware instances that can be supported by the dynamic driver. Not defining it means using a static driver.

Remarks

None

DRV_SDCARD_POWER_STATE Macro

Defines an override of the power state of the SD Card driver.

File

drv_sdcard_config_template.h

C

```
#define DRV_SDCARD_POWER_STATE SYS_MODULE_POWER_IDLE_STOP
```

Description

SD Card power state configuration

Defines an override of the power state of the SD Card driver.

Remarks

This feature may not be available in the device or the SD Card module selected.

DRV_SDCARD_SYS_FS_REGISTER Macro

Register to use with the File system

File

drv_sdcard_config_template.h

C

```
#define DRV_SDCARD_SYS_FS_REGISTER
```

Description

SDCARD Driver Register with File System

Specifying this macro enables the SDCARD driver to register its services with the SYS FS.

Remarks

This macro is optional and should be specified only if the SDCARD driver is to be used with the File System.

DRV_SDCARD_ENABLE_WRITE_PROTECT_CHECK Macro

Enable SD Card write protect check.

File

drv_sdcard_config_template.h

C

```
#define DRV_SDCARD_ENABLE_WRITE_PROTECT_CHECK
```

Description

SDCARD Driver Enable Write Protect Check

Specifying this macro enables the SDCARD driver to check whether the SD card is write protected.

Remarks

None

Building the Library

This section lists the files that are available in the SD Card Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/sdcard`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_sdcard.h`	This file provides the interface definitions of the SD Card driver

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_sdcard.c`	This file contains the core implementation of the SD Card driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The SD Card Driver Library depends on the following modules:

- SPI Driver Library
- Clock System Service Library
- Interrupt System Service Library
- Ports System Service Library
- Timer System Service Library
- Timer Driver Library

Library Interface

a) System Level Functions

	Name	Description
	DRV_SDCARD_Initialize	Initializes the SD Card driver. **Implementation:** Dynamic
	DRV_SDCARD_Deinitialize	Deinitializes the specified instance of the SD Card driver module. **Implementation:** Dynamic
	DRV_SDCARD_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings. **Implementation:** Dynamic
	DRV_SDCARD_Status	Provides the current status of the SD Card driver module. **Implementation:** Dynamic
	DRV_SDCARD_Tasks	Maintains the driver's state machine. **Implementation:** Dynamic

b) Client Level Functions

	Name	Description
	DRV_SDCARD_Close	Closes an opened-instance of the SD Card driver. **Implementation:** Dynamic
	DRV_SDCARD_Open	Opens the specified SD Card driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_SDCARD_Read	Reads blocks of data from the specified block address of the SD Card.
	DRV_SDCARD_Write	Writes blocks of data starting at the specified address of the SD Card.
	DRV_SDCARD_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

c) Status Functions

	Name	Description
	DRV_SDCARD_IsAttached	Returns the physical attach status of the SD Card.
	DRV_SDCARD_IsWriteProtected	Returns the write protect status of the SDCARD.
	DRV_SDCARD_CommandStatus	Gets the current status of the command.
	DRV_SDCARD_GeometryGet	Returns the geometry of the device.

d) Data Types and Constants

	Name	Description
	DRV_SDCARD_INDEX_0	SD Card driver index definitions
	DRV_SDCARD_INDEX_COUNT	Number of valid SD Card driver indices
	DRV_SDCARD_INIT	Contains all the data necessary to initialize the SD Card device
	_DRV_SDCARD_INIT	Contains all the data necessary to initialize the SD Card device
	SDCARD_DETECTION_LOGIC	Defines the different system events
	SDCARD_MAX_LIMIT	Maximum allowed SD card instances
	DRV_SDCARD_INDEX_1	This is macro DRV_SDCARD_INDEX_1.
	DRV_SDCARD_INDEX_2	This is macro DRV_SDCARD_INDEX_2.
	DRV_SDCARD_INDEX_3	This is macro DRV_SDCARD_INDEX_3.

	DRV_SDCARD_COMMAND_HANDLE_INVALID	SDCARD Driver's Invalid Command Handle.
	DRV_SDCARD_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_SDCARD_COMMAND_STATUS	Identifies the possible events that can result from a request.
	DRV_SDCARD_EVENT	Identifies the possible events that can result from a request.
	DRV_SDCARD_EVENT_HANDLER	Pointer to a SDCARDDriver Event handler function

Description

This section describes the Application Programming Interface (API) functions of the SD Card Driver.

Refer to each section for a detailed description.

a) System Level Functions

DRV_SDCARD_Initialize Function

Initializes the SD Card driver.

Implementation: Dynamic

File

drv_sdcard.h

C

```
SYS_MODULE_OBJ DRV_SDCARD_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT *const init);
```

Returns

If successful, returns a valid handle to a driver object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the SD Card driver, making it ready for clients to open and use the driver.

Remarks

This routine must be called before any other SD Card routine is called.

This routine should only be called once during system initialization unless DRV_SDCARD_Deinitialize is called to deinitialize the driver instance.

This routine will NEVER block for hardware access. If the operation requires time to allow the hardware to reinitialize, it will be reported by the DRV_SDCARD_Status operation. The system must use DRV_SDCARD_Status to find out when the driver is in the ready state.

Preconditions

None.

Example

```
DRV_SDCARD_INIT     init;
SYS_MODULE_OBJ      objectHandle;

// Populate the SD Card initialization structure

objectHandle = DRV_SDCARD_Initialize(DRV_SDCARD_INDEX_0, (SYS_MODULE_INIT*)&init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Index for the driver instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_SDCARD_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init
);

DRV_SDCARD_Deinitialize Function

Deinitializes the specified instance of the SD Card driver module.

Implementation: Dynamic

File

drv_sdcard.h

C

```
void DRV_SDCARD_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SD Card driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

This routine will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_SDCARD_Status operation. The system has to use DRV_SDCARD_Status to check if the de-initialization is complete.

Preconditions

Function DRV_SDCARD_Initialize must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
SYS_MODULE_OBJ      objectHandle;     //  Returned from DRV_SDCARD_Initialize
SYS_STATUS          status;

DRV_SDCARD_Deinitialize(objectHandle);

status = DRV_SDCARD_Status(objectHandle);
if (SYS_MODULE_UNINITIALIZED == status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SDCARD_Initialize routine.

Function

void DRV_SDCARD_Deinitialize
(
SYS_MODULE_OBJ object
);

DRV_SDCARD_Reinitialize Function

Reinitializes the driver and refreshes any associated hardware settings.

Implementation: Dynamic

File

drv_sdcard.h

C

```
void DRV_SDCARD_Reinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
```

Returns

None

Description

This routine reinitializes the driver and refreshes any associated hardware settings using the given initialization data, but it will not interrupt any ongoing operations.

Remarks

This function can be called multiple times to reinitialize the module.

This operation can be used to refresh any supported hardware registers as specified by the initialization data or to change the power state of the module.

This routine will NEVER block for hardware access. If the operation requires time to allow the hardware to reinitialize, it will be reported by the DRV_SDCARD_Status operation. The system must use DRV_SDCARD_Status to find out when the driver is in the ready state.

Preconditions

Function DRV_SDCARD_Initialize must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
DRV_SDCARD_INIT     init;
SYS_MODULE_OBJ      objectHandle;  //  Returned from DRV_SDCARD_Initialize

// Update the required fields of the SD Card initialization structure

DRV_SDCARD_Reinitialize (objectHandle, (SYS_MODULE_INIT*)&init);
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SDCARD_Initialize routine
init	Pointer to the initialization data structure

Function

void DRV_SDCARD_Reinitialize

(

SYS_MODULE_OBJ object,

const SYS_MODULE_INIT * const init

);

DRV_SDCARD_Status Function

Provides the current status of the SD Card driver module.

Implementation: Dynamic

File

drv_sdcard.h

C

```
SYS_STATUS DRV_SDCARD_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system level operation and cannot start another

Note Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_STATUS_BUSY - Indicates that the driver is busy with a previous system level operation and cannot start another

SYS_STATUS_ERROR - Indicates that the driver is in an error state

Description

This routine provides the current status of the SD Card driver module.

Remarks

Any value less than SYS_STATUS_ERROR is also an error state.

SYS_MODULE_DEINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR

This operation can be used to determine when any of the driver's module level operations has completed.

If the status operation returns SYS_STATUS_BUSY, then a previous operation has not yet completed. If the status operation returns SYS_STATUS_READY, then it indicates that all previous operations have completed.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This routine will NEVER block waiting for hardware.

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

Function DRV_SDCARD_Initialize must have been called before calling this

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SDCARD_Initialize
SYS_STATUS          status;

status = DRV_SDCARD_Status(object);

if (SYS_MODULE_UNINITIALIZED == status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
else if (SYS_STATUS_ERROR >= status)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SDCARD_Initialize routine

Function

SYS_STATUS DRV_SDCARD_Status

(

SYS_MODULE_OBJ object

);

DRV_SDCARD_Tasks Function

Maintains the driver's state machine.

Implementation: Dynamic

File

drv_sdcard.h

C

```
void DRV_SDCARD_Tasks(SYS_MODULE_OBJ object);
```

Returns

None

Description

This routine is used to maintain the driver's internal state machine.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_SDCARD_Initialize routine must have been called for the specified SDCARD driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SDCARD_Initialize

while (true)
{
    DRV_SDCARD_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_SDCARD_Initialize)

Function

void DRV_SDCARD_Tasks

(

SYS_MODULE_OBJ object

);

b) Client Level Functions

DRV_SDCARD_Close Function

Closes an opened-instance of the SD Card driver.

Implementation: Dynamic

File

drv_sdcard.h

C

```
void DRV_SDCARD_Close(DRV_HANDLE handle);
```

Returns

None

Description

This routine closes an opened-instance of the SD Card driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_SDCARD_Open before the caller may use the driver again.

If DRV_IO_INTENT_BLOCKING was requested and the driver was built appropriately to support blocking behavior call may block until the operation is complete.

If DRV_IO_INTENT_NON_BLOCKING request the driver client can call the DRV_SDCARD_Status operation to find out when the module is in the ready state (the handle is no longer valid).

Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_SDCARD_Initialize routine must have been called for the specified SD Card driver instance.

DRV_SDCARD_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_SDCARD_Open

DRV_SDCARD_Close (handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_SDCARD_Close
(
DRV_HANDLE handle
);

DRV_SDCARD_Open Function

Opens the specified SD Card driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_sdcard.h

C

```
DRV_HANDLE DRV_SDCARD_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID.

Description

This routine opens the specified SD Card driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_SDCARD_Close routine is called.

This routine will NEVER block waiting for hardware.

If the DRV_IO_INTENT_BLOCKING is requested and the driver was built appropriately to support blocking behavior, then other client-level operations may block waiting on hardware until they are complete.

If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID.

Preconditions

Function DRV_SDCARD_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_SDCARD_Open (DRV_SDCARD_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);

if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_SDCARD_Open
(
const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT intent

);

DRV_SDCARD_Read Function

Reads blocks of data from the specified block address of the SD Card.

File

drv_sdcard.h

C

```
void DRV_SDCARD_Read(DRV_HANDLE handle, DRV_SDCARD_COMMAND_HANDLE * commandHandle, void * targetBuffer,
uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_SDCARD_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking read operation for reading blocks of data from the SD Card. The function returns with a valid buffer handle in the commandHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SDCARD_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the driver handle is invalid
- if the target buffer pointer is NULL
- if the number of blocks to be read is zero or more than the actual number of blocks available
- if a buffer object could not be allocated to the request
- if the client opened the driver in write only mode

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SDCARD_EVENT_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_SDCARD_EVENT_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

None.

Preconditions

The DRV_SDCARD_Initialize routine must have been called for the specified SDCARD driver instance.

DRV_SDCARD_Open must have been called with DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE as the ioIntent to obtain a valid opened device handle.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = 0x00;
uint32_t    nBlock = 2;
DRV_SDCARD_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySDCARDHandle is the handle returned
// by the DRV_SDCARD_Open function.

DRV_SDCARD_Read(mySDCARDHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SDCARD_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}
else
{
    // Read Successful
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

commandHandle	Pointer to an argument that will contain the return buffer handle
targetBuffer	Buffer into which the data read from the SD Card will be placed
blockStart	Start block address of the SD Card from where the read should begin.
nBlock	Total number of blocks to be read.

Function

void DRV_SDCARD_Read
(
const DRV_HANDLE handle,
DRV_SDCARD_COMMAND_HANDLE * commandHandle,
void * targetBuffer,
uint32_t blockStart,
uint32_t nBlock
);

DRV_SDCARD_Write Function

Writes blocks of data starting at the specified address of the SD Card.

File

drv_sdcard.h

C

```
void DRV_SDCARD_Write(DRV_HANDLE handle, DRV_SDCARD_COMMAND_HANDLE * commandHandle, void * sourceBuffer,
uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_SDCARD_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data to the SD Card. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SDCARD_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer object could not be allocated to the request
- if the source buffer pointer is NULL
- if the client opened the driver for read only
- if the number of blocks to be written is either zero or more than the number of blocks actually available
- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SDCARD_EVENT_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_SDCARD_EVENT_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

None.

Preconditions

The DRV_SDCARD_Initialize() routine must have been called for the specified SDCARD driver instance.

DRV_SDCARD_Open() routine must have been called to obtain a valid opened device handle. DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified as a parameter to this routine.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = 0x00;
uint32_t    nBlock = 2;
DRV_SDCARD_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;
```

```
// mySDCARDHandle is the handle returned
// by the DRV_SDCARD_Open function.

// Client registers an event handler with driver

DRV_SDCARD_EventHandlerSet(mySDCARDHandle, APP_SDCARDEventHandler, (uintptr_t)&myAppObj);

DRV_SDCARD_Write(mySDCARDHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SDCARD_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_SDCARDEventHandler(DRV_SDCARD_EVENT event,
        DRV_SDCARD_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SDCARD_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SDCARD_EVENT_COMMAND_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
sourceBuffer	The source buffer containing data to be programmed to the SD Card.
blockStart	Start block address of SD Card where the writes should begin.
nBlock	Total number of blocks to be written.

Function

```
void DRV_SDCARD_Write
(
const       DRV_HANDLE handle,
    DRV_SDCARD_COMMAND_HANDLE * commandHandle,
void * sourceBuffer,
uint32_t blockStart,
uint32_t nBlock
);
```

DRV_SDCARD_EventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

File

drv_sdcard.h

C

```
void DRV_SDCARD_EventHandlerSet(const DRV_HANDLE handle, const void * eventHandler, const uintptr_t
context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client queues a request for a read or a write operation, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any read or write operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_SDCARD_Initialize() routine must have been called for the specified SDCARD driver instance.

The DRV_SDCARD_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_SDCARD_COMMAND_HANDLE commandHandle;

// drvSDCARDHandle is the handle returned
// by the DRV_SDCARD_Open function.

// Client registers an event handler with driver. This is done once.

DRV_SDCARD_EventHandlerSet(drvSDCARDHandle, APP_SDCARDEventHandler, (uintptr_t)&myAppObj);

DRV_SDCARD_Read(drvSDCARDHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SDCARD_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_SDCARDEventHandler(DRV_SDCARD_EVENT event,
        DRV_SDCARD_COMMAND_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_SDCARD_EVENT_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SDCARD_EVENT_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
```

```
            break;
        }
    }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_SDCARD_EventHandlerSet

(

const DRV_HANDLE handle,

const void * eventHandler,

const uintptr_t context

);

c) Status Functions

DRV_SDCARD_IsAttached Function

Returns the physical attach status of the SD Card.

File

drv_sdcard.h

C

```
bool DRV_SDCARD_IsAttached(const DRV_HANDLE handle);
```

Returns

Returns false if the handle is invalid otherwise returns the attach status of the SD Card. Returns true if the SD Card is attached and initialized by the SDCARD driver otherwise returns false.

Description

This function returns the physical attach status of the SD Card.

Remarks

None.

Preconditions

The DRV_SDCARD_Initialize() routine must have been called for the specified SDCARD driver instance.

The DRV_SDCARD_Open() routine must have been called to obtain a valid opened device handle.

Example

```
bool isSDCARDAttached;
isSDCARDAttached = DRV_SDCARD_isAttached(drvSDCARDHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_SDCARD_IsAttached

(

const DRV_HANDLE handle

);

DRV_SDCARD_IsWriteProtected Function

Returns the write protect status of the SDCARD.

File

drv_sdcard.h

C

```
bool DRV_SDCARD_IsWriteProtected(const DRV_HANDLE handle);
```

Returns

Returns true if the attached SD Card is write protected. Returns false if the handle is not valid, or if the SD Card is not write protected.

Description

This function returns true if the SD Card is write protected otherwise it returns false.

Remarks

None.

Preconditions

The DRV_SDCARD_Initialize() routine must have been called for the specified SDCARD driver instance.

The DRV_SDCARD_Open() routine must have been called to obtain a valid opened device handle.

Example

```
bool isWriteProtected;
isWriteProtected = DRV_SDCARD_IsWriteProtected(drvSDCARDHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_SDCARD_IsWriteProtected

(

const DRV_HANDLE handle

);

DRV_SDCARD_CommandStatus Function

Gets the current status of the command.

File

drv_sdcard.h

C

```
DRV_SDCARD_COMMAND_STATUS DRV_SDCARD_CommandStatus(const DRV_HANDLE handle, const DRV_SDCARD_COMMAND_HANDLE
commandHandle);
```

Returns

A DRV_SDCARD_COMMAND_STATUS value describing the current status of the command. Returns DRV_SDCARD_COMMAND_HANDLE_INVALID if the client handle or the command handle is not valid.

Description

This routine gets the current status of the command. The application must use this routine where the status of a scheduled command needs to be polled on. The function may return DRV_SDCARD_COMMAND_HANDLE_INVALID in a case where the command handle has expired. A command handle expires when the internal buffer object is re-assigned to another read or write request. It is recommended that this function be called regularly in order to track the command status correctly.

The application can alternatively register an event handler to receive read or write operation completion events.

Remarks

This routine will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_SDCARD_Initialize() routine must have been called.

The DRV_SDCARD_Open() must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE                      handle;         // Returned from DRV_SDCARD_Open
DRV_SDCARD_COMMAND_HANDLE       commandHandle;
DRV_SDCARD_COMMAND_STATUS       status;

status = DRV_SDCARD_CommandStatus(handle, commandHandle);
if(status == DRV_SDCARD_COMMAND_COMPLETED)
{
    // Operation Done
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_SDCARD_COMMAND_STATUS DRV_SDCARD_CommandStatus

(

const DRV_HANDLE handle,

const DRV_SDCARD_COMMAND_HANDLE commandHandle

);

DRV_SDCARD_GeometryGet Function

Returns the geometry of the device.

File

drv_sdcard.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_SDCARD_GeometryGet(const DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Pointer to structure which holds the media geometry information.

Description

This API gives the following geometrical details of the SD Card.

- Media Property
- Number of Read/Write/Erase regions in the SD Card
- Number of Blocks and their size in each region of the device

Remarks

None.

Preconditions

The DRV_SDCARD_Initialize() routine must have been called for the specified SDCARD driver instance.

The DRV_SDCARD_Open() routine must have been called to obtain a valid opened device handle.

Example

```
SYS_FS_MEDIA_GEOMETRY * SDCARDGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalSize;

SDCARDGeometry = DRV_SDCARD_GeometryGet(SDCARDOpenHandle1);
```

```
readBlockSize  = SDCARDGeometry->geometryTable->blockSize;
nReadBlocks = SDCARDGeometry->geometryTable->numBlocks;
nReadRegions = SDCARDGeometry->numReadRegions;

writeBlockSize  = (SDCARDGeometry->geometryTable +1)->blockSize;
eraseBlockSize  = (SDCARDGeometry->geometryTable +2)->blockSize;

totalSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY * DRV_SDCARD_GeometryGet
(
const DRV_HANDLE handle
);

d) Data Types and Constants

DRV_SDCARD_INDEX_0 Macro

SD Card driver index definitions

File

drv_sdcard.h

C

```
#define DRV_SDCARD_INDEX_0 0
```

Description

SD Card Driver Module Index Numbers
These constants provide SD Card driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.
These values should be passed into the DRV_SDCARD_Initialize and DRV_SDCARD_Open routines to identify the driver instance in use.

DRV_SDCARD_INDEX_COUNT Macro

Number of valid SD Card driver indices

File

drv_sdcard.h

C

```
#define DRV_SDCARD_INDEX_COUNT DRV_SDCARD_INDEX_MAX
```

Description

SD Card Driver Module Index Count
This constant identifies number of valid SD Card driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.
This value is derived from part-specific header files defined as part of the peripheral libraries.

DRV_SDCARD_INIT Structure

Contains all the data necessary to initialize the SD Card device

File

drv_sdcard.h

C

```
typedef struct _DRV_SDCARD_INIT {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX spiIndex;
  SPI_MODULE_ID spiId;
  CLK_BUSES_PERIPHERAL spiClk;
  uint32_t sdcardSpeedHz;
  SDCARD_DETECTION_LOGIC sdCardPinActiveLogic;
  PORTS_CHANNEL cardDetectPort;
  PORTS_BIT_POS cardDetectBitPosition;
  PORTS_CHANNEL writeProtectPort;
  PORTS_BIT_POS writeProtectBitPosition;
  PORTS_CHANNEL chipSelectPort;
  PORTS_BIT_POS chipSelectBitPosition;
} DRV_SDCARD_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX spiIndex;	SPI driver index
SPI_MODULE_ID spiId;	Identifies peripheral (PLIB-level) ID
CLK_BUSES_PERIPHERAL spiClk;	Peripheral clock used by the SPI
uint32_t sdcardSpeedHz;	SD card communication speed
SDCARD_DETECTION_LOGIC sdCardPinActiveLogic;	SD Card Pin Detection Logic
PORTS_CHANNEL cardDetectPort;	Card detect port
PORTS_BIT_POS cardDetectBitPosition;	Card detect pin
PORTS_CHANNEL writeProtectPort;	Write protect port
PORTS_BIT_POS writeProtectBitPosition;	Write protect pin
PORTS_CHANNEL chipSelectPort;	Chip select port
PORTS_BIT_POS chipSelectBitPosition;	Chip select pin

Description

SD Card Device Driver Initialization Data

This structure contains all the data necessary to initialize the SD Card device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_SDCARD_Initialize routine.

SDCARD_DETECTION_LOGIC Enumeration

Defines the different system events

File

drv_sdcard.h

C

```
typedef enum {
  SDCARD_DETECTION_LOGIC_ACTIVE_LOW,
  SDCARD_DETECTION_LOGIC_ACTIVE_HIGH
} SDCARD_DETECTION_LOGIC;
```

Members

Members	Description
SDCARD_DETECTION_LOGIC_ACTIVE_LOW	The media event is SD Card attach

SDCARD_DETECTION_LOGIC_ACTIVE_HIGH	The media event is SD Card detach

Description

System events

This enum defines different system events.

Remarks

None.

SDCARD_MAX_LIMIT Macro

Maximum allowed SD card instances

File

drv_sdcard.h

C

```
#define SDCARD_MAX_LIMIT 2
```

Description

SD Card Driver Maximum allowed limit

This constant identifies number of valid SD Card driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from part-specific header files defined as part of the peripheral libraries.

DRV_SDCARD_INDEX_1 Macro

File

drv_sdcard.h

C

```
#define DRV_SDCARD_INDEX_1 1
```

Description

This is macro DRV_SDCARD_INDEX_1.

DRV_SDCARD_INDEX_2 Macro

File

drv_sdcard.h

C

```
#define DRV_SDCARD_INDEX_2 2
```

Description

This is macro DRV_SDCARD_INDEX_2.

DRV_SDCARD_INDEX_3 Macro

File

drv_sdcard.h

C

```
#define DRV_SDCARD_INDEX_3 3
```

Description

This is macro DRV_SDCARD_INDEX_3.

DRV_SDCARD_COMMAND_HANDLE_INVALID Macro

SDCARD Driver's Invalid Command Handle.

File

drv_sdcard.h

C

```
#define DRV_SDCARD_COMMAND_HANDLE_INVALID SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID
```

Description

SDCARD Driver Invalid Command Handle.

This value defines the SDCARD Driver Invalid Command Handle. This value is returned by read or write routines when the command request was not accepted.

Remarks

None.

DRV_SDCARD_COMMAND_HANDLE Type

Handle identifying commands queued in the driver.

File

drv_sdcard.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_SDCARD_COMMAND_HANDLE;
```

Description

SDCARD Driver command handle.

A command handle is returned by a call to the Read or Write functions. This handle allows the application to track the completion of the operation. This command handle is also returned to the client along with the event that has occurred with respect to the command. This allows the application to connect the event to a specific command in case where multiple commands are queued.

The command handle associated with the command request expires when the client has been notified of the completion of the command (after event handler function that notifies the client returns) or after the command has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_SDCARD_COMMAND_STATUS Enumeration

Identifies the possible events that can result from a request.

File

drv_sdcard.h

C

```
typedef enum {
  DRV_SDCARD_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED,
  DRV_SDCARD_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED,
  DRV_SDCARD_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS,
  DRV_SDCARD_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN
} DRV_SDCARD_COMMAND_STATUS;
```

Members

Members	Description
DRV_SDCARD_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED	Done OK and ready
DRV_SDCARD_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED	Scheduled but not started

DRV_SDCARD_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS	Currently being in transfer
DRV_SDCARD_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN	Unknown Command

Description

SDCARD Driver Events

This enumeration identifies the possible events that can result from a read or a write request made by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_SDCARD_EventHandlerSet function when a request is completed.

DRV_SDCARD_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_sdcard.h

C

```
typedef enum {
  DRV_SDCARD_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_SDCARD_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR
} DRV_SDCARD_EVENT;
```

Members

Members	Description
DRV_SDCARD_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE	Operation has been completed successfully.
DRV_SDCARD_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR	There was an error during the operation

Description

SDCARD Driver Events

This enumeration identifies the possible events that can result from a read or a write request issued by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_SDCARD_EventHandlerSet function when a request is completed.

DRV_SDCARD_EVENT_HANDLER Type

Pointer to a SDCARDDriver Event handler function

File

drv_sdcard.h

C

```
typedef SYS_FS_MEDIA_EVENT_HANDLER DRV_SDCARD_EVENT_HANDLER;
```

Returns

None.

Description

SDCARD Driver Event Handler Function Pointer

This data type defines the required function signature for the SDCARD event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_SDCARD_EVENT_COMMAND_COMPLETE, it means that the write or a erase operation was completed successfully.

If the event is DRV_SDCARD_EVENT_COMMAND_ERROR, it means that the scheduled operation was not completed successfully.

The context parameter contains the handle to the client context, provided at the time the event handling function was registered using the DRV_SDCARD_EventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write/erase request.

Example

```
void APP_MySDCARDEventHandler
(
    DRV_SDCARD_EVENT event,
    DRV_SDCARD_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_SDCARD_EVENT_COMMAND_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_SDCARD_EVENT_COMMAND_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write requests
context	Value identifying the context of the application that registered the event handling function

Files

Files

Name	Description
drv_sdcard.h	SD Card Device Driver Interface File
drv_sdcard_config_template.h	SD Card driver configuration definitions template

Description

This section lists the source and header files used by the SD Card Driver Library.

drv_sdcard.h

SD Card Device Driver Interface File

Enumerations

	Name	Description
	DRV_SDCARD_COMMAND_STATUS	Identifies the possible events that can result from a request.
	DRV_SDCARD_EVENT	Identifies the possible events that can result from a request.
	SDCARD_DETECTION_LOGIC	Defines the different system events

Functions

	Name	Description
	DRV_SDCARD_Close	Closes an opened-instance of the SD Card driver. **Implementation:** Dynamic
	DRV_SDCARD_CommandStatus	Gets the current status of the command.
	DRV_SDCARD_Deinitialize	Deinitializes the specified instance of the SD Card driver module. **Implementation:** Dynamic
	DRV_SDCARD_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.
	DRV_SDCARD_GeometryGet	Returns the geometry of the device.
	DRV_SDCARD_Initialize	Initializes the SD Card driver. **Implementation:** Dynamic
	DRV_SDCARD_IsAttached	Returns the physical attach status of the SD Card.
	DRV_SDCARD_IsWriteProtected	Returns the write protect status of the SDCARD.
	DRV_SDCARD_Open	Opens the specified SD Card driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_SDCARD_Read	Reads blocks of data from the specified block address of the SD Card.
	DRV_SDCARD_Reinitialize	Reinitializes the driver and refreshes any associated hardware settings. **Implementation:** Dynamic
	DRV_SDCARD_Status	Provides the current status of the SD Card driver module. **Implementation:** Dynamic
	DRV_SDCARD_Tasks	Maintains the driver's state machine. **Implementation:** Dynamic
	DRV_SDCARD_Write	Writes blocks of data starting at the specified address of the SD Card.

Macros

	Name	Description
	DRV_SDCARD_COMMAND_HANDLE_INVALID	SDCARD Driver's Invalid Command Handle.
	DRV_SDCARD_INDEX_0	SD Card driver index definitions
	DRV_SDCARD_INDEX_1	This is macro DRV_SDCARD_INDEX_1.
	DRV_SDCARD_INDEX_2	This is macro DRV_SDCARD_INDEX_2.
	DRV_SDCARD_INDEX_3	This is macro DRV_SDCARD_INDEX_3.
	DRV_SDCARD_INDEX_COUNT	Number of valid SD Card driver indices
	SDCARD_MAX_LIMIT	Maximum allowed SD card instances

Structures

	Name	Description
	_DRV_SDCARD_INIT	Contains all the data necessary to initialize the SD Card device
	DRV_SDCARD_INIT	Contains all the data necessary to initialize the SD Card device

Types

	Name	Description
	DRV_SDCARD_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_SDCARD_EVENT_HANDLER	Pointer to a SDCARDDriver Event handler function

Description

SD Card Device Driver Interface

The SD Card device driver provides a simple interface to manage the "SD Card" peripheral. This file defines the interface definitions and prototypes for the SD Card driver.

File Name

drv_sdcard.h

Company

Microchip Technology Inc.

drv_sdcard_config_template.h

SD Card driver configuration definitions template

Macros

	Name	Description
	DRV_SDCARD_CLIENTS_NUMBER	Selects the miximum number of clients
	DRV_SDCARD_ENABLE_WRITE_PROTECT_CHECK	Enable SD Card write protect check.
	DRV_SDCARD_INDEX_MAX	SD Card Static Index selection
	DRV_SDCARD_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver
	DRV_SDCARD_POWER_STATE	Defines an override of the power state of the SD Card driver.
	DRV_SDCARD_SYS_FS_REGISTER	Register to use with the File system

Description

SD Card Driver Configuration Definitions for the template version

These definitions statically define the driver's mode of operation.

File Name

drv_sdcard_config_template.h

Company

Microchip Technology Inc.

SPI Driver Library

This section describes the Serial Peripheral Interface (SPI) Driver Library.

Introduction

This library provides an interface to manage the Serial Peripheral Interface (SPI) module on the Microchip family of microcontrollers in different modes of operation.

Description

The SPI module is a full duplex synchronous serial interface useful for communicating with other peripherals or microcontrollers in master/slave relationship and it can transfer data over short distances at high speeds. The peripheral devices may be serial EEPROMs, shift registers, display drivers, analog-to-digital converters, etc. The SPI module is compatible with Motorola's SPI and SIOP interfaces.

During data transfer devices can work either in master or in Slave mode. The source of synchronization is the system clock, which is generated by the master. The SPI module allows one or more slave devices to be connected to a single master device via the same bus.

The SPI serial interface consists of four pins, which are further sub-divided into data and control lines:

Data Lines:

• MOSI – Master Data Output, Slave Data Input

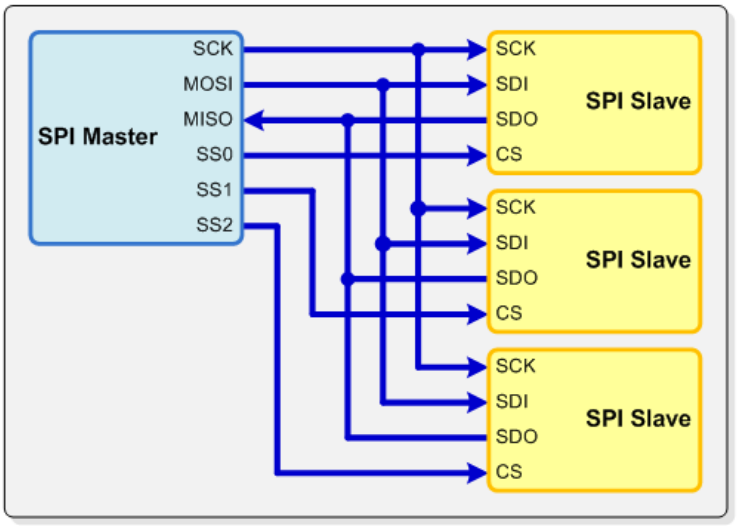
• MISO – Master Data Input, Slave Data Output

Control Lines:

• SCLK – Serial Clock

• /SS – Slave Select (no addressing)

SPI Master-Slave Relationship



The SPI module can be configured to operate using two, three, or four pins. In the 3-pin mode, the Slave Select line is not used. In the 2-pin mode, both the MOSI and /SS lines are not used.

Note: Third-party trademarks are property of their respective owners. Refer to the MPLAB Harmony *Software License Agreement* for complete licensing information. A copy of this agreement is available in the `<install-dir>/doc` folder of your MPLAB Harmony installation.

Using the Library

This topic describes the basic architecture of the SPI Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_spi.h`

The interface to the SPI Driver library is defined in the `drv_spi.h` header file. Any C language source (`.c`) file that uses the SPI Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the Driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the SPI Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

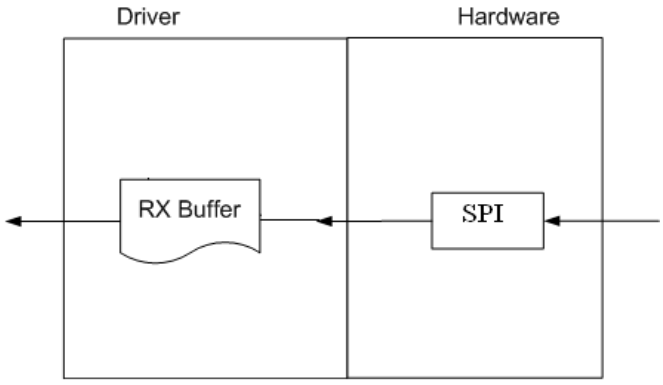
Description

Different types of SPIs are available on Microchip microcontrollers. Some have an internal buffer mechanism and some do not. The buffer depth varies across part families. The SPI driver abstracts out these differences and provides a unified model for data transfer across different types of SPIs available.

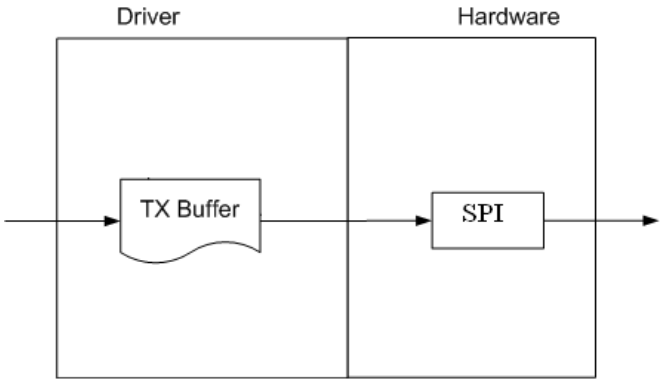
Both transmitter and receiver provides a buffer in the driver which transmits and receives data to/from the hardware. The SPI driver provides a set of interfaces to perform the read and the write.

The following diagrams illustrate the model used by the SPI driver for transmitter and receiver.

Receiver Abstraction Model



Transmitter Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the SPI module.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open, close, status and other setup functions.
Data Transfer Functions	Provides data transfer functions available in the configuration.
Miscellaneous	Provides driver miscellaneous functions, data transfer status function, version identification functions etc.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

Note: Not all modes are available on all devices, please refer to the specific device data sheet to determine the modes that are supported for your device.

System Access

System Initialization and Reinitialization

The system performs the initialization and the reinitialization of the device driver with settings that affect only the instance of the device that is being initialized or reinitialized. During system initialization each instance of the SPI module would be initialized with the following configuration

settings (either passed dynamically at run time using DRV_SPI_INIT or by using Initialization Overrides) that are supported by the specific SPI device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section
- The actual peripheral ID enumerated as the PLIB level module ID (e.g., SPI_ID_2)
- Defining the respective interrupt sources for TX, RX, and Error Interrupt

The DRV_SPI_Initialize API returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the Initialize interface would be used by the other system interfaces like DRV_SPI_Deinitialize, DRV_SPI_Status, and DRV_SPI_Tasks.

Note: The system initialization and the reinitialization settings, only affect the instance of the peripheral that is being initialized or reinitialized.

Example:

```
DRV_SPI_INIT                spiInitData;
SYS_MODULE_OBJ              objectHandle;

    spiInitData.moduleInit.value    = SYS_MODULE_POWER_RUN_FULL;
    spiInitData.spiId               = SPI_ID_1;
    spiInitData.taskMode            = DRV_SPI_TASK_MODE_POLLED;
    spiInitData.spiMode             = DRV_SPI_MODE_MASTER;
    spiInitData.spiProtocolType     = DRV_SPI_PROTOCOL_TYPE_STANDARD;
    spiInitData.commWidth           = SPI_COMMUNICATION_WIDTH_8BITS;
    spiInitData.baudRate            = 5000;
    spiInitData.bufferType          = DRV_SPI_BUFFER_TYPE_STANDARD;
                                      // It is highly recommended to set this to
                                      // DRV_SPI_BUFFER_TYPE_ENHANCED for hardware
                                      // that supports it
    spiInitData.inputSamplePhase    = SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE;
    spiInitData.clockMode           = DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_RISE;
    spiInitData.txInterruptSource   = INT_SOURCE_SPI_1_TRANSMIT;
    spiInitData.rxInterruptSource   = INT_SOURCE_SPI_1_RECEIVE;
    spiInitData.errInterruptSource  = INT_SOURCE_SPI_1_ERROR;
    spiInitData.queueSize = 10;
    spiInitData.jobQueueReserveSize = 1;

    objectHandle = DRV_SPI_Initialize(DRV_SPI_INDEX_1, (SYS_MODULE_INIT*)&spiInitData);
    if (SYS_MODULE_OBJ_INVALID == objectHandle)
   {
    // Handle error
  }
```

Tasks Routine

The system will either call DRV_SPI_Tasks, from System Task Service (in a polled environment) or DRV_SPI_Tasks will be called from the ISR of the SPI.

Client Access

General Client Operation

For the application to start using an instance of the module, it must call the DRV_SPI_Open function. This provides the configuration required to open the SPI instance for operation. If the driver is deinitialized using the function DRV_SPI_Deinitialize, the application must call the DRV_SPI_Open function again to set up the instance of the SPI.

For the various options available for IO_INTENT, please refer to **Data Types and Constants** in the Library Interface section.

After a client instance is opened, DRV_SPI_ClientConfigure can be called to set a client-specific bps, OperationStarting and OperationEnded callbacks. The OperationStarting callback will be called before the first bit is put onto the SPI bus, allowing for the slave select line to be toggled to active. The OperationEnded callback will be called after the last bit is received, allowing for the slave select line to be toggled to inactive. These two callbacks will be called from the ISR, if the SPI driver is operating in ISR mode, care should be taken that they do the minimum needed. For example, OSAL calls make cause exceptions in ISR context.

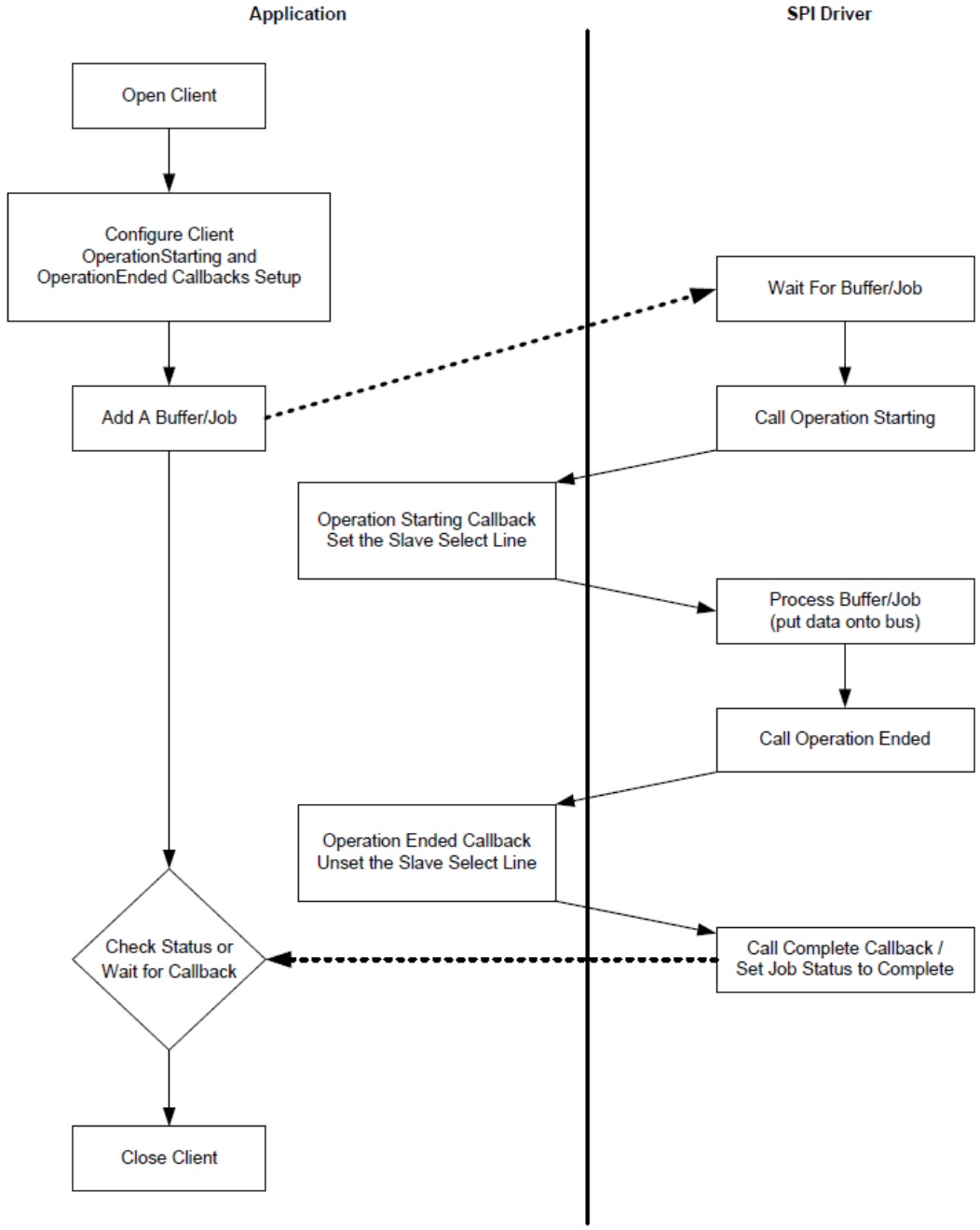
Example:

```
DRV_HANDLE handle;

// Configure the instance DRV_SPI_INDEX_1 with the configuration
handle = DRV_SPI_Open(DRV_SPI_INDEX_1, DRV_IO_INTENT_READWRITE);

if(handle == DRV_HANDLE_INVALID)
{
    // Client cannot open the instance.
}
```

Client Transfer - Core

Client basic functionality provides a extremely basic interface for the driver operation.

The following diagram illustrates the byte/word model used for the data transfer.



Note: It is not necessary to close and reopen the client between multiple transfers.

Client Data Transfer Functionality

Applications using the SPI byte/word functionality, need to perform the following:

1. The system should have completed necessary initialization and the DRV_SPI_Tasks should either be running in polled environment, or in an interrupt environment.
2. Open the driver using DRV_SPI_Open with the necessary intent.
3. Optionally configure the client with DRV_SPI_ClientConfigure to set up OperationStarting and OperationEnded callbacks to handle selecting and deselecting the slave select pin.
4. Add a buffer using the DRV_SPI_BufferAddRead/DRV_SPI_BufferAddWrite/DRV_SPI_BufferAddWriteRead functions. An optional callback can be provided that will be called when the buffer/job is complete.

5. Check for the current transfer status using DRV_SPI_BufferStatus until the transfer progress is DRV_SPI_BUFFER_EVENT_COMPLETE, or wait for the callback to be called. If the SPI driver is configured in Polled more, ensure that DRV_SPI_Tasks is called regularly to handle the buffer/job.
6. The client will be able to close the driver using DRV_SPI_Close when required.

Example:

```
SYS_MODULE_OBJ spiObject;

int main( void )
{
    while ( 1 )
    {
        appTask ();
        DRV_SPI_Tasks(spiObject);
    }
}
void appTask ()
{
    #define MY_BUFFER_SIZE   5
    DRV_HANDLE          handle;    // Returned from DRV_SPI_Open
    char          myBuffer[MY_BUFFER_SIZE] = { 11, 22, 33, 44, 55};
    unsigned int        numBytes;
    DRV_SPI_BUFFER_HANDLE bufHandle;

    // Preinitialize myBuffer with MY_BUFFER_SIZE bytes of valid data.
    while( 1 )
    {
        switch( state )
        {
            case APP_STATE_INIT:
                /* Initialize the SPI Driver */
                spiObject = DRV_SPI_Initialize( DRV_SPI_INDEX_1,
                                                ( SYS_MODULE_INIT * )
                                                  &initConf_1 );

                /* Check for the System Status */
                if( SYS_STATUS_READY != DRV_SPI_Status( spiObject ) )
                    return 0;

                /* Open the Driver */
                handle = DRV_SPI_Open( DRV_SPI_INDEX_1,
                                       DRV_IO_INTENT_EXCLUSIVE );

                /* Enable/Activate the CS */

                /* Update the state to transfer data */
                state = APP_STATE_DATA_PUT;
            break;

            case APP_STATE_DATA_PUT:
                bufHandle = DRV_SPI_BufferAddWrite ( handle, myBuffer,
                                                     5, NULL, NULL );
                /* Update the state to status check */
                state = APP_STATE_DATA_CHECK;
            break;

            case APP_STATE_DATA_CHECK:
                /* Check for the successful data transfer */
                if( DRV_SPI_BUFFER_EVENT_COMPLETE &
                    DRV_SPI_BufferStatus( bufhandle ) )
                {
                    /* Do this repeatedly */
                    state = APP_STATE_DATA_PUT;
                }

            break;
            default:
            break;
        }
    }
```

Configuring the Library

Miscellaneous Configuration

	Name	Description
	DRV_SPI_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver .
	DRV_SPI_CLIENTS_NUMBER	Selects the maximum number of clients.

System Configuration

	Name	Description
	DRV_SPI_16BIT	Controls the compilation of 16 Bit mode
	DRV_SPI_32BIT	Controls the compilation of 32 Bit mode
	DRV_SPI_8BIT	Controls the compilation of 8 Bit mode
	DRV_SPI_DMA	Controls the compilation of DMA support
	DRV_SPI_DMA_DUMMY_BUFFER_SIZE	Controls the size of DMA dummy buffer
	DRV_SPI_DMA_TXFER_SIZE	Controls the size of DMA transfers
	DRV_SPI_EBM	Controls the compilation of Enhanced Buffer Mode mode
	DRV_SPI_ELEMENTS_PER_QUEUE	Controls the number of elements that are allocated.
	DRV_SPI_ISR	Controls the compilation of ISR mode
	DRV_SPI_MASTER	Controls the compilation of master mode
	DRV_SPI_POLLED	Controls the compilation of Polled mode
	DRV_SPI_RM	Controls the compilation of Standard Buffer mode
	DRV_SPI_SLAVE	Controls the compilation of slave mode

Description

The configuration of the SPI driver is based on the file `system_config.h`.

This header file contains the configuration selection for the SPI driver. Based on the selections made, the SPI driver may support the selected features. These configuration settings will apply to all instances of the SPI driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

System Configuration

DRV_SPI_16BIT Macro

Controls the compilation of 16 Bit mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_16BIT 1
```

Description

SPI 16 Bit Mode Enable

This definition controls whether or not 16 Bit mode functionality is built as part of the driver. With it set to 1 then 16 Bit mode will be compiled and commWidth = SPI_COMMUNICATION_WIDTH_16BITS will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert. With this set the BufferAdd functions will only accept buffer sizes of multiples of 2 (16 bit words)

Remarks

Optional definition

DRV_SPI_32BIT Macro

Controls the compilation of 32 Bit mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_32BIT 1
```

Description

SPI 32 Bit Mode Enable

This definition controls whether or not 32 Bit mode functionality is built as part of the driver. With it set to 1 then 32 Bit mode will be compiled and commWidth = SPI_COMMUNICATION_WIDTH_32BITS will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert. With this set the BufferAdd functions will only accept buffer sizes of multiples of 4 (32 bit words)

Remarks

Optional definition

DRV_SPI_8BIT Macro

Controls the compilation of 8 Bit mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_8BIT 1
```

Description

SPI 8 Bit Mode Enable

This definition controls whether or not 8 Bit mode functionality is built as part of the driver. With it set to 1 then 8 Bit mode will be compiled and commWidth = SPI_COMMUNICATION_WIDTH_8BITS will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert.

Remarks

Optional definition

DRV_SPI_DMA Macro

Controls the compilation of DMA support

File

drv_spi_config_template.h

C

```
#define DRV_SPI_DMA 1
```

Description

SPI DMA Enable

This definition controls whether or not DMA functionality is built as part of the driver. With it set to 1 then DMA will be compiled.

Remarks

Optional definition

DRV_SPI_DMA_DUMMY_BUFFER_SIZE Macro

Controls the size of DMA dummy buffer

File

drv_spi_config_template.h

C

```
#define DRV_SPI_DMA_DUMMY_BUFFER_SIZE 256
```

Description

SPI DMA Dummy Buffer Size

This controls the size of the buffer the SPI driver uses to give to the DMA service when it is to send and receive invalid data on the bus. This occurs when the number of bytes to be read are different than the number of bytes transmitted.

Remarks

Optional definition

DRV_SPI_DMA_TXFER_SIZE Macro

Controls the size of DMA transfers

File

drv_spi_config_template.h

C

```
#define DRV_SPI_DMA_TXFER_SIZE 256
```

Description

SPI DMA Transfer Size

This definition controls the maximum number of bytes to transfer per DMA transfer.

Remarks

Optional definition

DRV_SPI_EBM Macro

Controls the compilation of Enhanced Buffer Mode mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_EBM 1
```

Description

SPI Enhanced Buffer Mode Enable (Hardware FIFO)

This definition controls whether or not Enhanced Buffer mode functionality is built as part of the driver. With it set to 1 then enhanced buffer mode will be compiled and bufferType = DRV_SPI_BUFFER_TYPE_ENHANCED will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert. This mode is not available on all PIC32s. Trying to use this mode on PICMX3XX/4XX will cause compile time warnings and errors.

Remarks

Optional definition

DRV_SPI_ELEMENTS_PER_QUEUE Macro

Controls the number of elements that are allocated.

File

drv_spi_config_template.h

C

```
#define DRV_SPI_ELEMENTS_PER_QUEUE 10
```

Description

SPI Buffer Queue Depth

This definition along with DRV_SPI_INSTANCES_NUMBER and DRV_SPI_CLIENT_NUMBER controls how many buffer queue elements are created.

Remarks

Optional definition

DRV_SPI_ISR Macro

Controls the compilation of ISR mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_ISR 1
```

Description

SPI ISR Mode Enable

This definition controls whether or not ISR mode functionality is built as part of the driver. With it set to 1 then ISR mode will be compiled and taskMode = DRV_SPI_TASK_MODE_ISR will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert

Remarks

Optional definition

DRV_SPI_MASTER Macro

Controls the compilation of master mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_MASTER 1
```

Description

SPI Master Mode Enable

This definition controls whether or not master mode functionality is built as part of the driver. With it set to 1 then master mode will be compiled and spiMode = DRV_SPI_MODE_MASTER will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert

Remarks

Optional definition

DRV_SPI_POLLED Macro

Controls the compilation of Polled mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_POLLED 1
```

Description

SPI Polled Mode Enable

This definition controls whether or not polled mode functionality is built as part of the driver. With it set to 1 then polled mode will be compiled and taskMode = DRV_SPI_TASK_MODE_POLLED will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert

Remarks

Optional definition

DRV_SPI_RM Macro

Controls the compilation of Standard Buffer mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_RM 1
```

Description

SPI Standard Buffer Mode Enable

This definition controls whether or not Standard Buffer mode functionality is built as part of the driver. With it set to 1 then standard buffer mode will be compiled and bufferType = DRV_SPI_BUFFER_TYPE_STANDARD will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert. This mode is available on all PIC32s

Remarks

Optional definition

DRV_SPI_SLAVE Macro

Controls the compilation of slave mode

File

drv_spi_config_template.h

C

```
#define DRV_SPI_SLAVE 1
```

Description

SPI Slave Mode Enable

This definition controls whether or not slave mode functionality is built as part of the driver. With it set to 1 then slave mode will be compiled and spiMode = DRV_SPI_MODE_SLAVE will be accepted by SPI_DRV_Initialize(). With it set to 0 SPI_DRV_Initialize() will cause an assert

Remarks

Optional definition

Miscellaneous Configuration

DRV_SPI_INSTANCES_NUMBER Macro

Selects the maximum number of hardware instances that can be supported by the dynamic driver .

File

drv_spi_config_template.h

C

```
#define DRV_SPI_INSTANCES_NUMBER 1
```

Description

SPI hardware instance configuration

This definition selects the maximum number of hardware instances that can be supported by the dynamic driver.

Remarks

Mandatory definition

DRV_SPI_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_spi_config_template.h

C

```
#define DRV_SPI_CLIENTS_NUMBER 1
```

Description

SPI maximum number of clients

This definition selects the maximum number of clients that the SPI driver can support at run time.

Remarks

Mandatory definition

Building the Library

This section lists the files that are available in the SPI Driver Library.

Description

This section list the files that are available in the \src folder of the SPI Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/spi`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_spi.h	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_spi.c	Basic SPI Driver implementation file.
/src/dynamic/drv_spi_api.c	Functions used by the driver API.
/src/drv_spi_sys_queue_fifo.c	Queue implementation used by the SPI Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The SPI Driver Library depends on the following modules:

- Clock System Service Library

Optional Dependencies

- DMA System Service Library (used when operating in DMA mode)
- Interrupt System Service Library (used when task is running in Interrupt mode)

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_SPI_Initialize	Initializes the SPI instance for the specified driver index. **Implementation:** Static/Dynamic

	DRV_SPI_Deinitialize	Deinitializes the specified instance of the SPI driver module. **Implementation:** Static/Dynamic
	DRV_SPI_Status	Provides the current status of the SPI driver module. **Implementation:** Static/Dynamic
	DRV_SPI_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Static/Dynamic

b) Client Setup Functions

	Name	Description
	DRV_SPI_Close	Closes an opened instance of the SPI driver. **Implementation:** Static/Dynamic
	DRV_SPI_Open	Opens the specified SPI driver instance and returns a handle to it. **Implementation:** Static/Dynamic
	DRV_SPI_ClientConfigure	Configures a SPI client with specific data. **Implementation:** Static/Dynamic

c) Data Transfer Functions

	Name	Description
	DRV_SPI_BufferStatus	Returns the transmitter and receiver transfer status. **Implementation:** Static/Dynamic
	DRV_SPI_BufferAddRead	Registers a buffer for a read operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPI_BufferAddWrite	Registers a buffer for a write operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPI_BufferAddWriteRead	Registers a buffer for a read and write operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPI_BufferAddRead2	Registers a buffer for a read operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPI_BufferAddWrite2	Registers a buffer for a write operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPI_BufferAddWriteRead2	Registers a buffer for a read and write operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPIn_ReceiverBufferIsFull	Returns the receive buffer status. 'n' represents the instance of the SPI driver used. **Implementation:** Static
	DRV_SPIn_TransmitterBufferIsFull	Returns the transmit buffer status. 'n' represents the instance of the SPI driver used. **Implementation:** Static

Description

This section describes the API functions of the SPI Driver library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_SPI_Initialize Function

Initializes the SPI instance for the specified driver index.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
SYS_MODULE_OBJ DRV_SPI_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

- If successful - returns a valid handle to a driver instance object

- If unsuccessful - returns SYS_MODULE_OBJ_INVALID

Description

This routine initializes the SPI driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the 'init' parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the SPI module ID. For example, driver instance 0 can be assigned to SPI2. If the driver is built statically, then some of the initialization parameters are overridden by configuration macros. Refer to the description of the DRV_SPI_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This routine must be called before any other SPI routine is called.

This routine should only be called once during system initialization unless DRV_SPI_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

None.

Example

```
DRV_SPI_INIT        init;
SYS_MODULE_OBJ      objectHandle;

// Populate the SPI initialization structure
init.spiId = SPI_ID_1,
init.taskMode = DRV_SPI_TASK_MODE_ISR,
init.spiMode = DRV_SPI_MODE_MASTER,
init.allowIdleRun = false,
init.spiProtocolType = DRV_SPI_PROTOCOL_TYPE_STANDARD,
init.commWidth = SPI_COMMUNICATION_WIDTH_8BITS,
init.baudClockSource = SPI_BAUD_RATE_PBCLK_CLOCK;
init.spiClk = CLK_BUS_PERIPHERAL_2,
init.baudRate = 10000000,
init.bufferType = DRV_SPI_BUFFER_TYPE_ENHANCED,
init.clockMode = DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL,
init.inputSamplePhase = SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE,
init.txInterruptSource = INT_SOURCE_SPI_1_TRANSMIT,
init.rxInterruptSource = INT_SOURCE_SPI_1_RECEIVE,
init.errInterruptSource = INT_SOURCE_SPI_1_ERROR,
init.dummyByteValue = 0xFF,
init.queueSize = 10,
init.jobQueueReserveSize = 1,

objectHandle = DRV_SPI_Initialize(DRV_SPI_INDEX_1, (SYS_MODULE_INIT*)usartInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized. Please note this is not the SPI id. The hardware SPI id is set in the initialization structure. This is the index of the driver index to use.
init	Pointer to a data structure containing any data necessary to initialize the driver. If this pointer is NULL, the driver uses the static initialization override macros for each member of the initialization data structure.

Function

SYS_MODULE_OBJ DRV_SPI_Initialize(const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init)

DRV_SPI_Deinitialize Function

Deinitializes the specified instance of the SPI driver module.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
void DRV_SPI_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SPI driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again.

This function will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_SPI_Status operation. The system has to use DRV_SPI_Status to find out when the module is in the ready state.

Preconditions

Function DRV_SPI_Initialize must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_SPI_Initialize
SYS_STATUS          status;

DRV_SPI_Deinitialize ( object );

status = DRV_SPI_Status( object );
if( SYS_MODULE_UNINITIALIZED == status )
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_SPI_Initialize

Function

void DRV_SPI_Deinitialize (SYS_MODULE_OBJ object)

DRV_SPI_Status Function

Provides the current status of the SPI driver module.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
SYS_STATUS DRV_SPI_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that the driver is busy with a previous

system level operation and cannot start another

Description

This function provides the current status of the SPI driver module.

Remarks

Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_MODULE_UNINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR.

This function can be used to determine when any of the driver's module level operations has completed.

If the status operation returns SYS_STATUS_BUSY, the previous operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This function will NEVER block waiting for hardware.

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The DRV_SPI_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SPI_Initialize
SYS_STATUS          status;

status = DRV_SPI_Status( object );
if( SYS_STATUS_READY != status )
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_SPI_Initialize

Function

SYS_STATUS DRV_SPI_Status (SYS_MODULE_OBJ object)

DRV_SPI_Tasks Function

Maintains the driver's state machine and implements its ISR.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
void DRV_SPI_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal state machine and implement its transmit ISR for interrupt-driven implementations. In polling mode, this function should be called from the SYS_Tasks() function. In interrupt mode, this function should be called in the transmit interrupt service routine of the USART that is associated with this USART driver hardware instance.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This function may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SPI_Initialize

while( true )
{
    DRV_SPI_Tasks ( object );

    // Do other tasks
```

```
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_SPI_Initialize)

Function

void DRV_SPI_Tasks (SYS_MODULE_OBJ object);

b) Client Setup Functions

DRV_SPI_Close Function

Closes an opened instance of the SPI driver.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
void DRV_SPI_Close(DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened instance of the SPI driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_SPI_Open before the caller may use the driver again. This function is thread safe in a RTOS application.

Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_SPI_Open

DRV_SPI_Close ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_SPI_Close (DRV_HANDLE handle)

DRV_SPI_Open Function

Opens the specified SPI driver instance and returns a handle to it.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
DRV_HANDLE DRV_SPI_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. An error can occur when the following is true:

- if the number of client objects allocated via DRV_SPI_INSTANCES_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid

Description

This routine opens the specified SPI driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

If ioIntent is DRV_IO_INTENT_READ, the client will only be read from the driver. If ioIntent is DRV_IO_INTENT_WRITE, the client will only be able to write to the driver. If the ioIntent in DRV_IO_INTENT_READWRITE, the client will be able to do both, read and write.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_SPI_Close routine is called. This routine will NEVER block waiting for hardware. If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

The DRV_SPI_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_SPI_Open( DRV_SPI_INDEX_0, DRV_IO_INTENT_EXCLUSIVE );

if( DRV_HANDLE_INVALID == handle )
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Index of the driver initialized with DRV_SPI_Initialize(). Please note this is not the SPI ID.
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT ORed together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_SPI_Open (const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT ioIntent)

DRV_SPI_ClientConfigure Function

Configures a SPI client with specific data.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
int32_t DRV_SPI_ClientConfigure(DRV_HANDLE handle, const DRV_SPI_CLIENT_DATA * cfgData);
```

Returns

- If successful - the routing will return greater than or equal to zero
- If an error occurs - the return value is negative

Description

This routine takes a DRV_SPI_CLIENT_DATA structure and sets client specific options. Whenever a new SPI job is started these values will be used. Passing in NULL will reset the client back to configuration parameters passed to driver initialization. A zero in any of the structure elements

will reset that specific configuration back to the driver default.

Preconditions

The DRV_SPI_Open function must have been called before calling this function.

Parameters

Parameters	Description
handle	handle of the client returned by DRV_SPI_Open.
cfgData	Client-specific configuration data.

Function

int32_t DRV_SPI_ClientConfigure (DRV_HANDLE handle,
const DRV_SPI_CLIENT_DATA * cfgData)

c) Data Transfer Functions

DRV_SPI_BufferStatus Function

Returns the transmitter and receiver transfer status.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_EVENT DRV_SPI_BufferStatus(DRV_SPI_BUFFER_HANDLE bufferHandle);
```

Returns

A DRV_SPI_BUFFER_STATUS value describing the current status of the transfer.

Description

This returns the transmitter and receiver transfer status.

Remarks

The returned status may contain a value with more than one of the bits specified in the DRV_SPI_BUFFER_STATUS enumeration set. The caller should perform an AND with the bit of interest and verify if the result is non-zero (as shown in the example) to verify the desired status bit.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

DRV_SPI_BufferAddmust have been called to obtain the buffer handle associated with that transfer.

Example

```
// Buffer handle returned from the data transfer function
DRV_SPI_BUFFER_HANDLE bufferHandle;

if(DRV_SPI_BufferStatus(bufferHandle) == DRV_SPI_BUFFER_EVENT_COMPLETE)
{
    // All transmitter data has been sent.
}
```

Parameters

Parameters	Description
bufferHandle	A valid buffer handle, returned from the driver's data transfer routine

Function

DRV_SPI_BUFFER_EVENT DRV_SPI_BufferStatus (DRV_SPI_BUFFER_HANDLE bufferHandle)

DRV_SPI_BufferAddRead Function

Registers a buffer for a read operation. Actual transfer will happen in the Task function.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddRead(DRV_HANDLE handle, void * rxBuffer, size_t size,
DRV_SPI_BUFFER_EVENT_HANDLER completeCB, void * context);
```

Returns

If the buffer add request is successful, a valid buffer handle is returned. If request is not queued up, DRV_SPI_BUFFER_HANDLE_INVALID is returned.

Description

Registers a buffer for a read operation. Actual transfer will happen in the Task function. The status of this operation can be monitored using DRV_SPI_BufferStatus function. A optional callback can also be provided that will be called when the operation is complete.

Remarks

This API will be deprecated soon, so avoid using it. Use "DRV_SPI_BufferAddRead2" instead of it.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SPI_Open call.

Example

```
DRV_HANDLE      handle;    // Returned from DRV_SPI_Open
char   myBuffer[MY_BUFFER_SIZE], state = 0;
DRV_SPI_BUFFER_HANDLE bufferHandle;

switch ( state )
{
    case 0:
        bufferHandle = DRV_SPI_BufferAddRead( handle, myBuffer, 10, NULL, NULL );
        if(bufferHandle != DRV_SPI_BUFFER_HANDLE_INVALID )
        {
            state++;
        }
        break;
    case 1:
        if(DRV_SPI_BufferStatus(bufferHandle) == DRV_SPI_BUFFER_EVENT_COMPLETE)
        {
            state++;
            // All transmitter data has been sent successfully.
        }
        break;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
rxBuffer	The buffer to which the data should be written to.
size	Number of bytes to be read from the SPI bus.
completeCB	Pointer to a function to be called when this queued operation is complete.
context	unused by the driver but this is passed to the callback when it is called.

Function

DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddRead (DRV_HANDLE handle, void *rxBuffer,

size_t size, DRV_SPI_BUFFER_EVENT_HANDLER completeCB,

void * context)

DRV_SPI_BufferAddWrite Function

Registers a buffer for a write operation. Actual transfer will happen in the Task function.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWrite(DRV_HANDLE handle, void * txBuffer, size_t size,
DRV_SPI_BUFFER_EVENT_HANDLER completeCB, void * context);
```

Returns

If the buffer add request is successful, a valid buffer handle is returned. If request is not queued up, DRV_SPI_BUFFER_HANDLE_INVALID is returned.

Description

Registers a buffer for a write operation. Actual transfer will happen in the Task function. The status of this operation can be monitored using DRV_SPI_BufferStatus function. A optional callback can also be provided that will be called when the operation is complete.

Remarks

This API will be deprecated soon, so avoid using it. Use "DRV_SPI_BufferAddWrite2" instead of it.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SPI_Open call.

Example

```
DRV_HANDLE      handle;    // Returned from DRV_SPI_Open
char   myBuffer[MY_BUFFER_SIZE], state = 0;
DRV_SPI_BUFFER_HANDLE bufferHandle;

switch ( state )
{
    case 0:
        bufferHandle = DRV_SPI_BufferAddWrite( handle, myBuffer, 10, NULL, NULL );
        if(bufferHandle != DRV_SPI_BUFFER_HANDLE_INVALID )
        {
            state++;
        }
        break;
    case 1:
        if(DRV_SPI_BufferStatus(bufferHandle) == DRV_SPI_BUFFER_EVENT_COMPLETE)
        {
            state++;
            // All transmitter data has been sent successfully.
        }
        break;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
txBuffer	The buffer which hold the data.
size	Number of bytes to be written to the SPI bus.
completeCB	Pointer to a function to be called when this queued operation is complete
context	unused by the driver but this is passed to the callback when it is called

Function

DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWrite (DRV_HANDLE handle, void *txBuffer,

size_t size, DRV_SPI_BUFFER_EVENT_HANDLER completeCB,
void * context)

DRV_SPI_BufferAddWriteRead Function

Registers a buffer for a read and write operation. Actual transfer will happen in the Task function.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWriteRead(DRV_HANDLE handle, void * txBuffer, size_t txSize, void *
rxBuffer, size_t rxSize, DRV_SPI_BUFFER_EVENT_HANDLER completeCB, void * context);
```

Returns

If the buffer add request is successful, a valid buffer handle is returned. If request is not queued up, DRV_SPI_BUFFER_HANDLE_INVALID is returned.

Description

Registers a buffer for a read and write operation. Actual transfer will happen in the Task function. The status of this operation can be monitored using DRV_SPI_BufferStatus function. A optional callback can also be provided that will be called when the operation is complete.

Remarks

This API will be deprecated soon, so avoid using it. Use "DRV_SPI_BufferAddWriteRead2" instead of it.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      handle;    // Returned from DRV_SPI_Open
char   myReadBuffer[MY_BUFFER_SIZE], myWriteBuffer[MY_BUFFER_SIZE], state = 0;
DRV_SPI_BUFFER_HANDLE bufferHandle;

switch ( state )
{
    case 0:
        bufferHandle = DRV_SPI_BufferAddWriteRead( handle, myWriteBuffer, 10, myReadBuffer, 10, NULL, NULL
);
        if(bufferHandle != DRV_SPI_BUFFER_HANDLE_INVALID )
        {
            state++;
        }
        break;
    case 1:
        if(DRV_SPI_BufferStatus(bufferHandle) == DRV_SPI_BUFFER_EVENT_COMPLETE)
        {
            state++;
            // All transmitter data has been sent successfully.
        }
        break;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
txBuffer	The buffer which hold the data.
txSize	Number of bytes to be written to the SPI bus.
rxBuffer	The buffer to which the data should be written to.
rxSize	Number of bytes to be read from the SPI bus
completeCB	Pointer to a function to be called when this queued operation is complete
context	unused by the driver but this is passed to the callback when it is called

Function

DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWriteRead(DRV_HANDLE handle,
void *txBuffer, void *rxBuffer, size_t size,)

DRV_SPI_BufferAddRead2 Function

Registers a buffer for a read operation. Actual transfer will happen in the Task function.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddRead2(DRV_HANDLE handle, void * rxBuffer, size_t size,
DRV_SPI_BUFFER_EVENT_HANDLER completeCB, void * context, DRV_SPI_BUFFER_HANDLE * jobHandle);
```

Returns

If the buffer add request is successful, a valid buffer handle is returned. If request is not queued up, DRV_SPI_BUFFER_HANDLE_INVALID is returned.

Description

Registers a buffer for a read operation. Actual transfer will happen in the Task function. The status of this operation can be monitored using DRV_SPI_BufferStatus function. A optional callback can also be provided that will be called when the operation is complete.

Remarks

None.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SPI_Open call.

Example

```
DRV_HANDLE      handle;    // Returned from DRV_SPI_Open
char   myBuffer[MY_BUFFER_SIZE], state = 0;
DRV_SPI_BUFFER_HANDLE bufferHandle, bufferHandle2;

switch ( state )
{
    case 0:
        bufferHandle = DRV_SPI_BufferAddRead2( handle, myBuffer, 10, NULL, NULL, &bufferHandle2 );
        if(bufferHandle2 != DRV_SPI_BUFFER_HANDLE_INVALID )
        {
            state++;
        }
        break;
    case 1:
        if(DRV_SPI_BufferStatus(bufferHandle2) == DRV_SPI_BUFFER_EVENT_COMPLETE)
        {
            state++;
            // All transmitter data has been sent successfully.
        }
        break;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
rxBuffer	The buffer to which the data should be written to.
size	Number of bytes to be read from the SPI bus.
completeCB	Pointer to a function to be called when this queued operation is complete
context	unused by the driver but this is passed to the callback when it is called

jobHandle	pointer to the buffer handle, this will be set before the function returns and can be used in the ISR callback.

Function

DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddRead2 (DRV_HANDLE handle, void *rxBuffer,

size_t size, DRV_SPI_BUFFER_EVENT_HANDLER completeCB,

void * context, DRV_SPI_BUFFER_HANDLE * jobHandle)

DRV_SPI_BufferAddWrite2 Function

Registers a buffer for a write operation. Actual transfer will happen in the Task function.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWrite2(DRV_HANDLE handle, void * txBuffer, size_t size,
DRV_SPI_BUFFER_EVENT_HANDLER completeCB, void * context, DRV_SPI_BUFFER_HANDLE * jobHandle);
```

Returns

If the buffer add request is successful, a valid buffer handle is returned. If request is not queued up, DRV_SPI_BUFFER_HANDLE_INVALID is returned.

Description

Registers a buffer for a write operation. Actual transfer will happen in the Task function. The status of this operation can be monitored using DRV_SPI_BufferStatus function. A optional callback can also be provided that will be called when the operation is complete.

Remarks

None.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SPI_Open call.

Example

```
DRV_HANDLE      handle;    // Returned from DRV_SPI_Open
char   myBuffer[MY_BUFFER_SIZE], state = 0;
DRV_SPI_BUFFER_HANDLE bufferHandle, bufferHandle2;

switch ( state )
{
    case 0:
        bufferHandle = DRV_SPI_BufferAddWrite2( handle, myBuffer, 10, NULL, NULL, &bufferHandle2 );
        if(bufferHandle2 != DRV_SPI_BUFFER_HANDLE_INVALID )
        {
            state++;
        }
        break;
    case 1:
        if(DRV_SPI_BufferStatus(bufferHandle2) == DRV_SPI_BUFFER_EVENT_COMPLETE)
        {
            state++;
            // All transmitter data has been sent successfully.
        }
        break;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
txBuffer	The buffer which hold the data.

size	Number of bytes to be written to the SPI bus.
completeCB	Pointer to a function to be called when this queued operation is complete
context	unused by the driver but this is passed to the callback when it is called
jobHandle	pointer to the buffer handle, this will be set before the function returns and can be used in the ISR callback.

Function

DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWrite2 (DRV_HANDLE handle, void *txBuffer,

size_t size, DRV_SPI_BUFFER_EVENT_HANDLER completeCB,

void * context, DRV_SPI_BUFFER_HANDLE * jobHandle)

DRV_SPI_BufferAddWriteRead2 Function

Registers a buffer for a read and write operation. Actual transfer will happen in the Task function.

Implementation: Static/Dynamic

File

drv_spi.h

C

```
DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWriteRead2(DRV_HANDLE handle, void * txBuffer, size_t txSize, void *
rxBuffer, size_t rxSize, DRV_SPI_BUFFER_EVENT_HANDLER completeCB, void * context, DRV_SPI_BUFFER_HANDLE *
jobHandle);
```

Returns

If the buffer add request is successful, a valid buffer handle is returned. If request is not queued up, DRV_SPI_BUFFER_HANDLE_INVALID is returned.

Description

Registers a buffer for a read and write operation. Actual transfer will happen in the Task function. The status of this operation can be monitored using DRV_SPI_BufferStatus function. A optional callback can also be provided that will be called when the operation is complete.

Remarks

None.

Preconditions

The DRV_SPI_Initialize routine must have been called for the specified SPI driver instance.

DRV_SPI_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      handle;    // Returned from DRV_SPI_Open
char   myReadBuffer[MY_BUFFER_SIZE], myWriteBuffer[MY_BUFFER_SIZE], state = 0;
DRV_SPI_BUFFER_HANDLE bufferHandle, bufferHandle2;

switch ( state )
{
    case 0:
        bufferHandle = DRV_SPI_BufferAddWriteRead2( handle, myWriteBuffer, 10, myReadBuffer, 10, NULL,
NULL, &bufferHandle2 );
        if(bufferHandle2 != DRV_SPI_BUFFER_HANDLE_INVALID )
        {
            state++;
        }
        break;
    case 1:
        if(DRV_SPI_BufferStatus(bufferHandle2) == DRV_SPI_BUFFER_EVENT_COMPLETE)
        {
            state++;
            // All transmitter data has been sent successfully.
        }
        break;
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
txBuffer	The buffer which hold the data.
txSize	Number of bytes to be written to the SPI bus.
rxBuffer	The buffer to which the data should be written to.
rxSize	Number of bytes to be read from the SPI bus
completeCB	Pointer to a function to be called when this queued operation is complete
context	unused by the driver but this is passed to the callback when it is called
jobHandle	pointer to the buffer handle, this will be set before the function returns and can be used in the ISR callback.

Function

DRV_SPI_BUFFER_HANDLE DRV_SPI_BufferAddWriteRead2(DRV_HANDLE handle,
void *txBuffer, void *rxBuffer, size_t size,
DRV_SPI_BUFFER_EVENT_HANDLER completeCB,
void * context, DRV_SPI_BUFFER_HANDLE * jobHandle)

DRV_SPIn_ReceiverBufferIsFull Function

Returns the receive buffer status. 'n' represents the instance of the SPI driver used.

Implementation: Static

File

drv_spi.h

C

```
bool DRV_SPIn_ReceiverBufferIsFull();
```

Returns

Receive Buffer Status

- 1 - Full
- 0 - Empty

Description

This function returns the receive buffer status (full/empty).

Remarks

None.

Preconditions

None.

Example

```
bool rxBufStat;
// Using instance 1 of SPI driver, that is n = 1
rxBufStat = DRV_SPI1_ReceiverBufferIsFull();

if (rxBufStat)
{
...
}
```

Function

bool DRV_SPIn_ReceiverBufferIsFull(void)

DRV_SPIn_TransmitterBufferIsFull Function

Returns the transmit buffer status. 'n' represents the instance of the SPI driver used.

Implementation: Static

File

drv_spi.h

C

```
bool DRV_SPIn_TransmitterBufferIsFull();
```

Returns

Transmit Buffer Status

- 1 - Full
- 0 - Empty

Description

This function returns the transmit buffer status (full/empty).

Remarks

None.

Preconditions

None.

Example

```
bool txBufStat;
// Using instance 1 of SPI driver, that is n = 1
txBufStat = DRV_SPI1_TransmitterBufferIsFull();

if (txBufStat)
{
...
}
```

Function

bool DRV_SPIn_TransmitterBufferIsFull(void)

d) Miscellaneous Functions

e) Data Types and Constants

Files

Files

Name	Description
drv_spi.h	SPI device driver interface file.
drv_spi_config_template.h	SPI Driver configuration definitions template.

Description

This section lists the source and header files used by the SPI Driver Library.

drv_spi.h

SPI device driver interface file.

Functions

	Name	Description
	DRV_SPI_BufferAddRead	Registers a buffer for a read operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPI_BufferAddRead2	Registers a buffer for a read operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic

	DRV_SPI_BufferAddWrite	Registers a buffer for a write operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPI_BufferAddWrite2	Registers a buffer for a write operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPI_BufferAddWriteRead	Registers a buffer for a read and write operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPI_BufferAddWriteRead2	Registers a buffer for a read and write operation. Actual transfer will happen in the Task function. **Implementation:** Static/Dynamic
	DRV_SPI_BufferStatus	Returns the transmitter and receiver transfer status. **Implementation:** Static/Dynamic
	DRV_SPI_ClientConfigure	Configures a SPI client with specific data. **Implementation:** Static/Dynamic
	DRV_SPI_Close	Closes an opened instance of the SPI driver. **Implementation:** Static/Dynamic
	DRV_SPI_Deinitialize	Deinitializes the specified instance of the SPI driver module. **Implementation:** Static/Dynamic
	DRV_SPI_Initialize	Initializes the SPI instance for the specified driver index. **Implementation:** Static/Dynamic
	DRV_SPI_Open	Opens the specified SPI driver instance and returns a handle to it. **Implementation:** Static/Dynamic
	DRV_SPI_Status	Provides the current status of the SPI driver module. **Implementation:** Static/Dynamic
	DRV_SPI_Tasks	Maintains the driver's state machine and implements its ISR. **Implementation:** Static/Dynamic
	DRV_SPIn_ReceiverBufferIsFull	Returns the receive buffer status. 'n' represents the instance of the SPI driver used. **Implementation:** Static
	DRV_SPIn_TransmitterBufferIsFull	Returns the transmit buffer status. 'n' represents the instance of the SPI driver used. **Implementation:** Static

Description

SPI Driver Interface

The SPI driver provides a simple interface to manage the SPI module. This file defines the interface definitions and prototypes for the SPI driver.

File Name

drv_spi.h

Company

Microchip Technology Inc.

drv_spi_config_template.h

SPI Driver configuration definitions template.

Macros

	Name	Description
	DRV_SPI_16BIT	Controls the compilation of 16 Bit mode
	DRV_SPI_32BIT	Controls the compilation of 32 Bit mode
	DRV_SPI_8BIT	Controls the compilation of 8 Bit mode
	DRV_SPI_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_SPI_DMA	Controls the compilation of DMA support
	DRV_SPI_DMA_DUMMY_BUFFER_SIZE	Controls the size of DMA dummy buffer
	DRV_SPI_DMA_TXFER_SIZE	Controls the size of DMA transfers
	DRV_SPI_EBM	Controls the compilation of Enhanced Buffer Mode mode
	DRV_SPI_ELEMENTS_PER_QUEUE	Controls the number of elements that are allocated.
	DRV_SPI_INSTANCES_NUMBER	Selects the maximum number of hardware instances that can be supported by the dynamic driver .
	DRV_SPI_ISR	Controls the compilation of ISR mode

	DRV_SPI_MASTER	Controls the compilation of master mode
	DRV_SPI_POLLED	Controls the compilation of Polled mode
	DRV_SPI_RM	Controls the compilation of Standard Buffer mode
	DRV_SPI_SLAVE	Controls the compilation of slave mode

Description

SPI Driver Configuration Definitions for the Template Version

These definitions statically define the driver's mode of operation.

File Name

drv_spi_config_template.h

Company

Microchip Technology Inc.

SPI Flash Driver Library

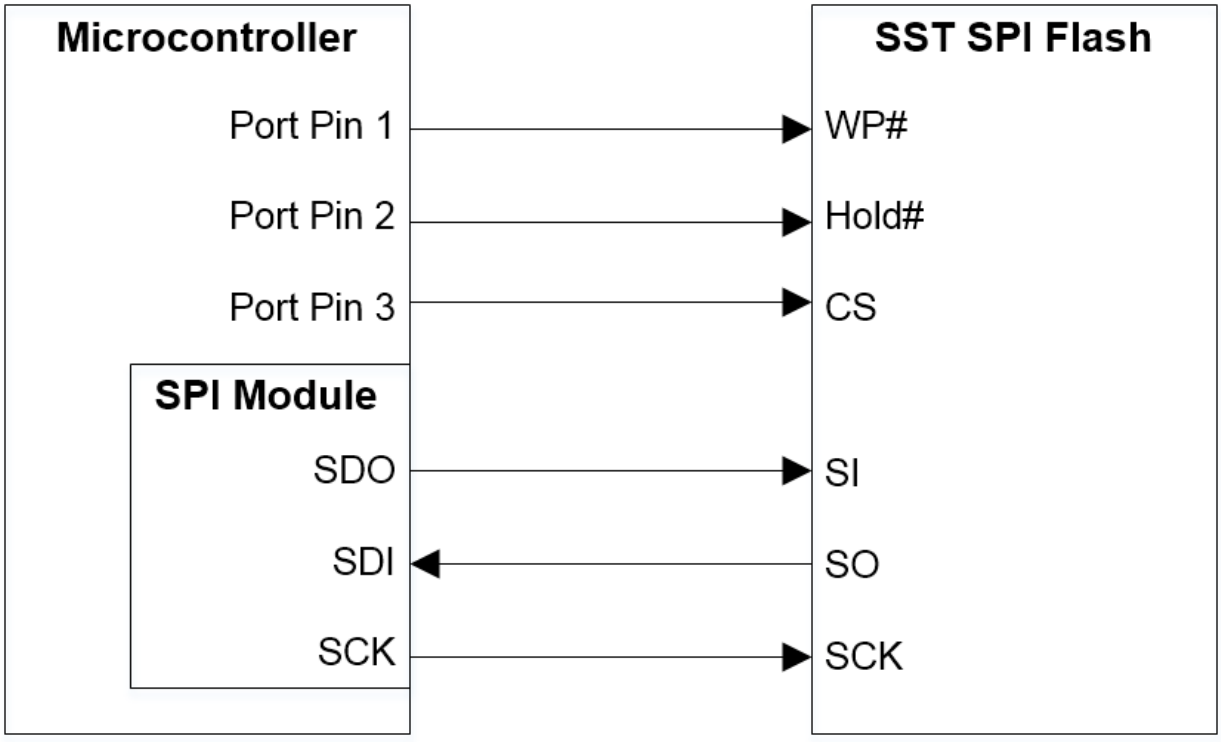
This section describes the Serial Peripheral Interface (SPI) Flash Driver Library.

Introduction

This library provides an interface to manage the SST SPI Flash modules (SST25VF020B, SST25VF016B, and SST25VF064C) in different modes of operation.

Description

The SPI Flash Driver uses SPI interface to establish the communication between SST Flash and Microchip microcontrollers. The SPI module of the controller works as a Master device and the Flash module works as a Slave. The following diagram shows the pin connections that are required to make the driver operational:



The SPI Flash Driver is dynamic in nature, so single instance of it can support multiple clients that want to use the same Flash. Multiple instances of the driver can be used when multiple Flash devices are required to be part of the system. The SPI Driver, which is used by the SPI Flash Driver, can be configured for use in either Polled or Interrupt mode.

Using the Library

This topic describes the basic architecture of the SPI Flash Driver Library and provides information and examples on its use.

Description

Interface Header Files: `drv_sst25vf016b.h`, `drv_sst25vf020b.h`, or `drv_sst25vf064c.h`

The interface to the SPI Flash Driver Library is defined in the header file. Any C language source (`.c`) file that uses the SPI Flash Driver library should include this header.

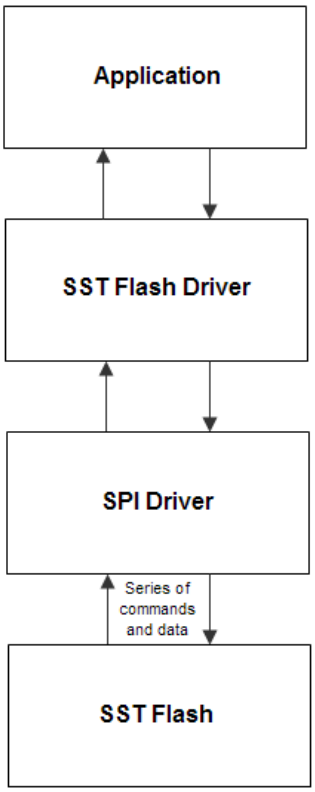
Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the SPI Flash Driver Library with a convenient C language interface. This topic describes how that abstraction is modeled in software.

Description

The SST SPI Flash needs a specific set of commands to be given on its SPI interface along with the required address and data to do different operations. This driver abstracts these requirements and provide simple APIs that can be used to perform Erase, Write, and Read operations. The SPI Driver is used for this purpose. The following layered diagram depicts the communication between different modules.



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the SPI Flash module.

Library Interface Section	Description
System Functions	These functions are accessed by the MPLAB Harmony System module and allow the driver to be initialized, deinitialized, and maintained.
Core Client Functions	These functions allow the application client to open and close the driver.
Block Operation Functions	These functions enable the Flash module to be erased, written, and read (to/from).
Media Interface Functions	These functions provide media status and the Flash geometry.

How the Library Works

The library provides interfaces to support:

- System Initialization/Deinitialization
- Opening the Driver
- Block Operations

System Initialization and Deinitialization

Provides information on initializing the system.

Description

System Initialization and Deinitialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization each instance of the SST Flash module would be initialized with the following configuration settings (either passed dynamically at run-time using DRV_SST25VF020B_INIT, DRV_SST25VF016B_INIT, or DRV_SST25VF064C_INIT, or by using Initialization Overrides) that are supported or used by the specific SST Flash device hardware:

- Device requested power state: one of the System Module Power States. For specific details please refer to **Data Types and Constants** in the Library Interface section
- The SPI Driver Module Index which is intended to be used to communicate with SST Flash (e.g., DRV_SPI_INDEX_0)
- Port Pins of the microcontroller to be used for Chip Select, Write Protection, and Hold operations on the SST Flash device
- Maximum Buffer Queue Size for that instance of the SST Flash Driver

Using the SST25VF020B as an example, the DRV_SST25VF020B_Initialize function returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the Initialize interface would be used by the other system interfaces like DRV_SST25VF020B_Deinitialize, DRV_SST25VF020B_Status, and DRV_SST25VF020B_Tasks.

Note: The system initialization and the deinitialization settings, only affect the instance of the peripheral that is being initialized or deinitialized.

Example:

```
// This code example shows the initialization of the SST25VF020B SPI Flash
// Driver. SPI driver index 0 is used for the purpose. Pin numbers 1, 2,
// and 3 of PORTB are configured for the Hold pin, Write Protection pin, and
// the Chip Select pin, respectively. The maximum buffer queue size is set to 5.

   DRV_SST25VF020B_INIT   SST25VF020BInitData;
   SYS_MODULE_OBJ      objectHandle;

   SST25VF020BInitData.moduleInit.value        = SYS_MODULE_POWER_RUN_FULL;
   SST25VF020BInitData.spiDriverModuleIndex    = DRV_SPI_INDEX_0;
   SST25VF020BInitData.holdPortChannel         = PORT_CHANNEL_B;
   SST25VF020BInitData.holdBitPosition         = PORTS_BIT_POS_1;
   SST25VF020BInitData.writeProtectPortChannel = PORT_CHANNEL_B;
   SST25VF020BInitData.writeProtectBitPosition = PORTS_BIT_POS_2;
   SST25VF020BInitData.chipSelectPortChannel   = PORT_CHANNEL_F;
   SST25VF020BInitData.chipSelectBitPosition   = PORTS_BIT_POS_2;
   SST25VF020BInitData.queueSize               = 5;

   objectHandle = DRV_SST25VF020B_Initialize(DRV_SST25VF020B_INDEX_0,
                                  (SYS_MODULE_INIT*)SST25VF020BInitData);
   if (SYS_MODULE_OBJ_INVALID == objectHandle)
   {
       // Handle error
   }
```

Tasks Routine

The system will either call DRV_SST25VF020B_Tasks, from SYS_Tasks (in a polled environment) or DRV_SST25VF020B_Tasks will be called from the ISR of the SPI module in use.

Opening the Driver

Provides information on opening the driver.

Description

To use the SST Flash driver, the application must open the driver. Using the SST25VF020B as an example, this is done by calling the DRV_SST25VF020B_Open function. Calling this function with DRV_IO_INTENT_NONBLOCKING will cause the driver to be opened in non blocking mode. Then DRV_SST25VF020B_BlockErase, DRV_SST25VF020B_BlockWrite and DRV_SST25VF020B_BlockRead functions when called by this client will be non-blocking.

The client can also open the driver in Read-only mode (DRV_IO_INTENT_READ), Write-only mode (DRV_IO_INTENT_WRITE), and Exclusive mode (DRV_IO_INTENT_EXCLUSIVE). If the driver has been opened exclusively by a client, it cannot be opened again by another client.

If successful, the DRV_SST25VF020B_Open function will return a handle to the driver. This handle records the association between the client and the driver instance that was opened. The DRV_SST25VF020B_Open function may return DRV_HANDLE_INVALID in the situation where the driver is not ready to be opened. When this occurs, the application can try opening the driver again. Note that the open function may return an invalid handle in other (error) cases as well.

The following code shows an example of the driver being opened in different modes.

```
DRV_HANDLE sstHandle1, sstHandle2;

/* Client 1 opens the SST driver in non blocking mode */
sstHandle1 = DRV_SST25VF020B_Open(DRV_SST25VF020B_INDEX_0, DRV_IO_INTENT_NONBLOCKING);

/* Check if the handle is valid */
if(DRV_HANDLE_INVALID == sstHandle1)
{
    /* The driver was not opened successfully. The client
     * can try opening it again */
}

/* Client 2 opens the SST driver in Exclusive Write only mode */
sstHandle2 = DRV_SST25VF020B_Open(DRV_SST25VF020B_INDEX_0, DRV_IO_INTENT_WRITE | DRV_IO_INTENT_EXCLUSIVE);

/* Check if the handle is valid */
if(DRV_HANDLE_INVALID == sstHandle2)
{
    /* The driver was not opened successfully. The client
     * can try opening it again */
}
```

Block Operations

Provides information on block operations.

Description

This driver provides simple client interfaces to Erase, Write, and Read the SST flash in blocks. A block is the unit to represent minimum amount of data that can be erased, written, or read. Block size may differ for Erase, Write, and Read operations. Using the SST25VF020B as an example, the DRV_SST25VF020B_GeometryGet function can be used to determine the different block sizes for the driver.

The DRV_SST25VF020B_BlockErase, DRV_SST25VF020B_BlockWrite, and DRV_SST25VF020B_BlockRead functions are used to erase, write, and read the data to/from SST SPI Flash. These functions are always non-blocking. All of these functions follow a standard queue model to read, write, and erase. When any of these functions are called (i.e., a block request is made), the request is queued. The size of the queue is determined by the queueSize member of the DRV_SST25VF020B_INIT data structure. All of the requests in the queue are executed by the DRV_SST25VF020B_Tasks function one-by-one.

When the driver adds a request to the queue, it returns a buffer handle. This handle allows the client to track the request as it progresses through the queue. The buffer handle expires when the event associated with the buffer completes. The driver provides driver events (DRV_SST25VF020B_BLOCK_EVENT) that indicate termination of the buffer requests.

The following steps can be performed for a simple Block Data Operation:

1. The system should have completed necessary initialization of the SPI Driver and the SST Flash Driver, and the DRV_SST25VF020B_Tasks function should be running in a polled environment.
2. The DRV_SPI_Tasks function should be running in either a polled environment or an interrupt environment.
3. Open the driver using DRV_SST25VF020B_Open with the necessary intent.
4. Set an event handler callback using the function DRV_SST25VF020B_BlockEventHandlerSet.
5. Request for block operations using the functions, DRV_SST25VF020B_BlockErase, DRV_SST25VF020B_BlockWrite, and DRV_SST25VF020B_BlockRead, with the appropriate parameters.
6. Wait for event handler callback to occur and check the status of the block operation using the callback function parameter of type DRV_SST25VF020B_BLOCK_EVENT.
7. The client will be able to close the driver using the function, DRV_SST25VF020B_Close, when required.

Example:

```
/* This code example shows usage of the block operations
 * on the SPI Flash SST25VF020B device */
```

```
DRV_HANDLE sstHandle1;
uint8_t myData1[10], myData2[10];
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE  blockHandle1, blockHandle2, blockHandle3;

/* The driver is opened for read-write in Exclusive mode */
sstHandle1 = DRV_SST25VF020B_Open(DRV_SST25VF020B_INDEX_0,
                                  DRV_IO_INTENT_READWRITE | DRV_IO_INTENT_EXCLUSIVE);

/* Check if the driver was opened successfully */
if(DRV_HANDLE_INVALID == sstHandle1)
{
    /* The driver could not be opened successfully */
}

/* Register a Buffer Event Handler with SST25VF020B  driver.
 * This event handler function will be called whenever
 * there is a buffer event.  An application defined
 * context can also be specified. This is returned when
 * the event handler is called.
 * */
DRV_SST25VF020B_BlockEventHandlerSet(sstHandle1,
        APP_SSTBufferEventHandler, NULL);

/* Request for all the three block operations one by one */

/* first block API to erase 1 block of the flash starting from address 0x0, each block is of 4kbyte */
DRV_SST25VF020B_BlockErase(sstHandle1, &blockHandle1, 0x0, 1);
/* 2nd block API to write myData1 in the first 10 locations of the flash */
DRV_SST25VF020B_BlockWrite(sstHandle1, &blockHandle2, &myData1[0], 0x0, 10);
/* 3rd block API to read the first 10 locations of the flash into myData2 */
DRV_SST25VF020B_BlockRead(sstHandle1, &blockHandle3, &myData2[0], 0x0, 10);

/* This is the Driver Event Handler */

void APP_SSTBufferEventHandler(DRV_SST25VF020B_BLOCK_EVENT event,
        DRV_SST25VF020B_BLOCK_COMMAND_HANDLE blockHandle, uintptr_t contextHandle)
{
    switch(event)
    {
        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE:
    if ( blockHandle == blockHandle3)
    {
            /* This means the data was read */
    /* Do data verification/processing */
    }
            break;
        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR:
            /* Error handling here. */
            break;
        default:
            break;
    }
}
```

Configuring the Library

SST25VF016B Configuration

	Name	Description
	DRV_SST25VF016B_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_SST25VF016B_HARDWARE_HOLD_ENABLE	Specifies if the hardware hold feature is enabled or not.
	DRV_SST25VF016B_HARDWARE_WRITE_PROTECTION_ENABLE	Specifies if the hardware write protect feature is enabled or not.
	DRV_SST25VF016B_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported

	DRV_SST25VF016B_MODE	Determines whether the driver is implemented as static or dynamic
	DRV_SST25VF016B_QUEUE_DEPTH_COMBINED	Number of entries of queues in all instances of the driver.

SST25VF020B Configuration

	Name	Description
	DRV_SST25VF020B_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_SST25VF020B_HARDWARE_HOLD_ENABLE	Specifies if the hardware hold feature is enabled or not.
	DRV_SST25VF020B_HARDWARE_WRITE_PROTECTION_ENABLE	Specifies if the hardware write protect feature is enabled or not.
	DRV_SST25VF020B_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_SST25VF020B_MODE	Determines whether the driver is implemented as static or dynamic.
	DRV_SST25VF020B_QUEUE_DEPTH_COMBINED	Number of entries of queues in all instances of the driver.

SST25VF064C Configuration

	Name	Description
	DRV_SST25VF064C_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_SST25VF064C_HARDWARE_HOLD_ENABLE	Specifies whether or not the hardware hold feature is enabled.
	DRV_SST25VF064C_HARDWARE_WRITE_PROTECTION_ENABLE	Specifies whether or not the hardware write protect feature is enabled.
	DRV_SST25VF064C_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_SST25VF064C_MODE	Determines whether the driver is implemented as static or dynamic.
	DRV_SST25VF064C_QUEUE_DEPTH_COMBINED	Number of entries of queues in all instances of the driver.

Description

The SST Flash Driver requires the specification of compile-time configuration macros. These macros define resource usage, feature availability, and dynamic behavior of the driver. These configuration macros should be defined in the `system_config.h` file.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

SST25VF016B Configuration

DRV_SST25VF016B_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_sst25vf016b_config_template.h

C

```
#define DRV_SST25VF016B_CLIENTS_NUMBER 4
```

Description

SST25VF016B Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. This value represents the total number of clients to be supported across all hardware instances. So if SST25VF016B-1 will be accessed by 2 clients and SST25VF016B-2 will accessed by 3 clients, then this number should be 5. It is recommended that this be set exactly equal to the number of expected clients. Client support consumes RAM memory space. If this macro is not defined and the DRV_SST25VF016B_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - single client operation. If this macro is defined and the DRV_SST25VF016B_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - multi client operation.

Remarks

None.

DRV_SST25VF016B_HARDWARE_HOLD_ENABLE Macro

Specifies if the hardware hold feature is enabled or not.

File

drv_sst25vf016b_config_template.h

C

```
#define DRV_SST25VF016B_HARDWARE_HOLD_ENABLE false
```

Description

SST25VF016B Hardware HOLD Support

This macro defines if the hardware hold feature is enabled or not. If hardware hold is enabled, then user must provide a port pin corresponding to HOLD pin on the flash

Remarks

None

DRV_SST25VF016B_HARDWARE_WRITE_PROTECTION_ENABLE Macro

Specifies if the hardware write protect feature is enabled or not.

File

drv_sst25vf016b_config_template.h

C

```
#define DRV_SST25VF016B_HARDWARE_WRITE_PROTECTION_ENABLE false
```

Description

SST25VF016B Hardware Write Protect Support

This macro defines if the hardware Write Protect feature is enabled or not. If hardware write protection is enabled, then user must provide a port pin corresponding to WP pin on the flash

Remarks

None.

DRV_SST25VF016B_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_sst25vf016b_config_template.h

C

```
#define DRV_SST25VF016B_INSTANCES_NUMBER 2
```

Description

SST25VF016B driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of SST25VF016B modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None.

DRV_SST25VF016B_MODE Macro

Determines whether the driver is implemented as static or dynamic

File

drv_sst25vf016b_config_template.h

C

```
#define DRV_SST25VF016B_MODE DYNAMIC
```

Description

SST25VF016B mode

Determines whether the driver is implemented as static or dynamic. Static drivers control the peripheral directly with peripheral library routines.

Remarks

None.

DRV_SST25VF016B_QUEUE_DEPTH_COMBINED Macro

Number of entries of queues in all instances of the driver.

File

drv_sst25vf016b_config_template.h

C

```
#define DRV_SST25VF016B_QUEUE_DEPTH_COMBINED 7
```

Description

SST25VF016B Driver Instance combined queue depth.

This macro defines the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for all the read/write/erase operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build).

A buffer queue will contain buffer queue entries, each related to a BufferAdd request. This configuration macro defines total number of buffer entries that will be available for use between all SST25VF016B driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking erase/write/read requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its buffer queue size.

SST25VF020B Configuration

DRV_SST25VF020B_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_sst25vf020b_config_template.h

C

```
#define DRV_SST25VF020B_CLIENTS_NUMBER 4
```

Description

SST25VF020B Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. This value represents the total number of clients to be supported across all hardware instances. So if SST25VF020B-1 will be accessed by 2 clients and SST25VF020B-2 will accessed by 3 clients, then this number should be 5. It is recommended that this be set exactly equal to the number of expected clients. Client support consumes RAM memory space. If this macro is not defined and the DRV_SST25VF020B_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - single client operation. If this macro is defined and the DRV_SST25VF020B_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - multi client operation.

Remarks

None.

DRV_SST25VF020B_HARDWARE_HOLD_ENABLE Macro

Specifies if the hardware hold feature is enabled or not.

File

drv_sst25vf020b_config_template.h

C

```
#define DRV_SST25VF020B_HARDWARE_HOLD_ENABLE false
```

Description

SST25VF020B Hardware HOLD Support

This macro defines if the hardware hold feature is enabled or not. If hardware hold is enabled, then user must provide a port pin corresponding to HOLD pin on the flash

Remarks

None.

DRV_SST25VF020B_HARDWARE_WRITE_PROTECTION_ENABLE Macro

Specifies if the hardware write protect feature is enabled or not.

File

drv_sst25vf020b_config_template.h

C

```
#define DRV_SST25VF020B_HARDWARE_WRITE_PROTECTION_ENABLE false
```

Description

SST25VF020B Hardware Write Protect Support

This macro defines if the hardware Write Protect feature is enabled or not. If hardware write protection is enabled, then user must provide a port pin corresponding to WP pin on the flash

Remarks

None.

DRV_SST25VF020B_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported.

File

drv_sst25vf020b_config_template.h

C

```
#define DRV_SST25VF020B_INSTANCES_NUMBER 2
```

Description

SST25VF020B driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of SST25VF020B modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None.

DRV_SST25VF020B_MODE Macro

Determines whether the driver is implemented as static or dynamic.

File

drv_sst25vf020b_config_template.h

C

```
#define DRV_SST25VF020B_MODE DYNAMIC
```

Description

SST25VF020B mode

Determines whether the driver is implemented as static or dynamic. Static drivers control the peripheral directly with peripheral library routines.

Remarks

None.

DRV_SST25VF020B_QUEUE_DEPTH_COMBINED Macro

Number of entries of queues in all instances of the driver.

File

drv_sst25vf020b_config_template.h

C

```
#define DRV_SST25VF020B_QUEUE_DEPTH_COMBINED 7
```

Description

SST25VF020B Driver Instance combined queue depth.

This macro defines the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for all the read/write/erase operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build).

A buffer queue will contain buffer queue entries, each related to a BufferAdd request. This configuration macro defines total number of buffer entries that will be available for use between all SST25VF020B driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking erase/write/read requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its buffer queue size.

SST25VF064C Configuration

DRV_SST25VF064C_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_sst25vf064c_config_template.h

C

```
#define DRV_SST25VF064C_CLIENTS_NUMBER 4
```

Description

SST25VF064C Client Count Configuration

Sets up the maximum number of clients that can be connected to any hardware instance. This value represents the total number of clients to be supported across all hardware instances. So if SST25VF064C-1 will be accessed by 2 clients and SST25VF064C-2 will accessed by 3 clients, then this number should be 5. It is recommended that this be set exactly equal to the number of expected clients. Client support consumes RAM memory space. If this macro is not defined and the DRV_SST25VF064C_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - single client operation. If this macro is defined and the DRV_SST25VF064C_INSTANCES_NUMBER macro is not defined, then the driver will be built for static - multi client operation.

Remarks

None.

DRV_SST25VF064C_HARDWARE_HOLD_ENABLE Macro

Specifies whether or not the hardware hold feature is enabled.

File

drv_sst25vf064c_config_template.h

C

```
#define DRV_SST25VF064C_HARDWARE_HOLD_ENABLE false
```

Description

SST25VF064C Hardware HOLD Support

This macro defines whether or not the hardware hold feature is enabled. If hardware hold is enabled, the user must provide a port pin corresponding to the HOLD pin on the Flash device.

Remarks

None.

DRV_SST25VF064C_HARDWARE_WRITE_PROTECTION_ENABLE Macro

Specifies whether or not the hardware write protect feature is enabled.

File

drv_sst25vf064c_config_template.h

C

```
#define DRV_SST25VF064C_HARDWARE_WRITE_PROTECTION_ENABLE false
```

Description

SST25VF064C Hardware Write Protect Support

This macro defines whether or not the hardware Write Protect feature is enabled. If hardware write protection is enabled, the user must provide a port pin corresponding to the WP pin on the Flash device.

Remarks

None.

DRV_SST25VF064C_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported.

File

drv_sst25vf064c_config_template.h

C

```
#define DRV_SST25VF064C_INSTANCES_NUMBER 2
```

Description

SST25VF064C driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of SST25VF064C modules that are needed by the application. Hardware Instance support consumes RAM memory space. If this macro is not defined, then the driver will be built statically.

Remarks

None.

DRV_SST25VF064C_MODE Macro

Determines whether the driver is implemented as static or dynamic.

File

drv_sst25vf064c_config_template.h

C

```
#define DRV_SST25VF064C_MODE DYNAMIC
```

Description

SST25VF064C mode

Determines whether the driver is implemented as static or dynamic. Static drivers control the peripheral directly with peripheral library routines.

Remarks

None.

DRV_SST25VF064C_QUEUE_DEPTH_COMBINED Macro

Number of entries of queues in all instances of the driver.

File

drv_sst25vf064c_config_template.h

C

```
#define DRV_SST25VF064C_QUEUE_DEPTH_COMBINED 7
```

Description

SST25VF064C Driver Instance combined queue depth.

This macro defines the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for all the read/write/erase operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build).

A buffer queue will contain buffer queue entries, each related to a BufferAdd request. This configuration macro defines total number of buffer entries that will be available for use between all SST25VF064C driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking erase/write/read requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. More the number of buffer entries, greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its buffer queue size.

Building the Library

This section lists the files that are available in the SPI Flash Driver Library.

Description

This section list the files that are available in the `/src` folder of the SPI Flash Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/spi_flash`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`sst25vf016b/drv_sst25vf016b.h`	Header file that exports the SST25VF016B driver API.
`sst25vf020b/drv_sst25vf020b.h`	Header file that exports the SST25VF020B driver API.
`sst25vf064c/drv_sst25vf064c.h`	Header file that exports the SST25VF064C driver API.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`sst25vf016b/src/dynamic/drv_sst25vf016b.c`	Basic SPI Flash Driver SST25VF016B implementation file.

sst25vf020b/src/dynamic/drv_sst25vf020b.c	Basic SPI Flash Driver SST25VF020B implementation file.
sst25vf064c/src/dynamic/drv_sst25vf064c.c	Basic SPI Flash Driver SST25VF064C implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
sst25vf020b/src/dynamic/drv_sst25vf020b_erasewrite.c	This file implements an optional BlockEraseWrite feature for the SST25VF020B driver.

Module Dependencies

The SPI Flash Driver Library depends on the following modules:

- SPI Driver Library
- Ports System Service Library

Library Interface

This section describes the API functions of the SPI Flash Driver Library.

Refer to each section for a detailed description.

SST25FV016B API

a) System Functions

	Name	Description
	DRV_SST25VF016B_Initialize	Initializes the SST25VF016B SPI Flash Driver instance for the specified driver index. **Implementation:** Dynamic
	DRV_SST25VF016B_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module. **Implementation:** Dynamic
	DRV_SST25VF016B_Status	Gets the current status of the SPI Flash Driver module. **Implementation:** Dynamic
	DRV_SST25VF016B_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR. **Implementation:** Dynamic

b) Core Client Functions

	Name	Description
	DRV_SST25VF016B_Close	Closes an opened-instance of the SPI Flash driver. **Implementation:** Dynamic
	DRV_SST25VF016B_Open	Opens the specified SPI Flash driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_SST25VF016B_ClientStatus	Gets current client-specific status of the SPI Flash driver. **Implementation:** Dynamic

c) Block Operation Functions

	Name	Description
	DRV_SST25VF016B_BlockErase	Erase the specified number of blocks in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF016B_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed. **Implementation:** Dynamic
	DRV_SST25VF016B_BlockRead	Reads blocks of data starting from the specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF016B_BlockWrite	Write blocks of data starting from a specified address in Flash memory. **Implementation:** Dynamic

d) Media Interface Functions

	Name	Description
	DRV_SST25VF016B_GeometryGet	Returns the geometry of the device. **Implementation:** Dynamic
	DRV_SST25VF016B_MediaIsAttached	Returns the status of the media. **Implementation:** Dynamic

e) Data Types and Constants

	Name	Description
	DRV_SST25VF016B_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.
	DRV_SST25VF016B_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_SST25VF016B_CLIENT_STATUS	Defines the client status. **Implementation:** Dynamic
	DRV_SST25VF016B_EVENT_HANDLER	Pointer to a SST25VF016B SPI Flash Driver Event handler function. **Implementation:** Dynamic
	DRV_SST25VF016B_INIT	Contains all the data necessary to initialize the SPI Flash device. **Implementation:** Dynamic
	DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_SST25VF016B_INDEX_0	SPI Flash driver index definitions
	DRV_SST25VF016B_INDEX_1	This is macro DRV_SST25VF016B_INDEX_1.

Description

This section contains the SST25V016B Flash device API.

a) System Functions

DRV_SST25VF016B_Initialize Function

Initializes the SST25VF016B SPI Flash Driver instance for the specified driver index.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
SYS_MODULE_OBJ DRV_SST25VF016B_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes the SPI Flash driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This function must be called before any other SPI Flash function is called.

This function should only be called once during system initialization unless DRV_SST25VF016B_Deinitialize is called to deinitialize the driver instance.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the SST25VF016B SPI
// Flash Driver. SPI driver index 0 is used for the purpose. Pin numbers 1, 2
// and 3 of port channel B are configured for hold pin, write protection pin
```

```
// and chip select pin respectively. Maximum buffer queue size is set 5.

DRV_SST25VF016B_INIT   SST25VF016BInitData;
SYS_MODULE_OBJ      objectHandle;

SST25VF016BInitData.moduleInit.value      = SYS_MODULE_POWER_RUN_FULL;
SST25VF016BInitData.spiDriverModuleIndex  = DRV_SPI_INDEX_0;
SST25VF016BInitData.holdPortChannel       = PORT_CHANNEL_B;
SST25VF016BInitData.holdBitPosition       = PORTS_BIT_POS_1;
SST25VF016BInitData.writeProtectPortChannel = PORT_CHANNEL_B;
SST25VF016BInitData.writeProtectBitPosition = PORTS_BIT_POS_2;
SST25VF016BInitData.chipSelectPortChannel = PORT_CHANNEL_F;
SST25VF016BInitData.chipSelectBitPosition = PORTS_BIT_POS_2;
SST25VF016BInitData.queueSize = 5;

objectHandle = DRV_SST25VF016B_Initialize(DRV_SST25VF016B_INDEX_0,
                                (SYS_MODULE_INIT*)SST25VF016BInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to a data structure containing data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_SST25VF016B_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

);

DRV_SST25VF016B_Deinitialize Function

Deinitializes the specified instance of the SPI Flash driver module.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SPI Flash Driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This function will NEVER block waiting for hardware.

Preconditions

Function DRV_SST25VF016B_Initialize should have been called before calling this function.

Example

```
// This code snippet shows an example of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_SST25VF016B_Initialize
SYS_STATUS          status;
```

```
DRV_SST25VF016B_Deinitialize(object);

status = DRV_SST25VF016B_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SST25VF016B_Initialize

Function

void DRV_SST25VF016B_Deinitialize(SYS_MODULE_OBJ object)

DRV_SST25VF016B_Status Function

Gets the current status of the SPI Flash Driver module.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
SYS_STATUS DRV_SST25VF016B_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and accept requests for new operations

SYS_STATUS_UNINITIALIZED - Indicates that the driver is not initialized

Description

This function provides the current status of the SPI Flash Driver module.

Remarks

A driver can only be opened when its status is SYS_STATUS_READY.

Preconditions

Function DRV_SST25VF016B_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST25VF016B_Initialize
SYS_STATUS          SST25VF016BStatus;

SST25VF016BStatus = DRV_SST25VF016B_Status(object);
else if (SYS_STATUS_ERROR >= SST25VF016BStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SST25VF016B_Initialize

Function

SYS_STATUS DRV_SST25VF016B_Status(SYS_MODULE_OBJ object)

DRV_SST25VF016B_Tasks Function

Maintains the driver's read, erase, and write state machine and implements its ISR.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the driver's internal state machine and should be called from the system's Tasks function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks function (SYS_Tasks).

Preconditions

The DRV_SST25VF016B_Initialize function must have been called for the specified SPI Flash driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST25VF016B_Initialize

while (true)
{
    DRV_SST25VF016B_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_SST25VF016B_Initialize)

Function

void DRV_SST25VF016B_Tasks (SYS_MODULE_OBJ object);

b) Core Client Functions

DRV_SST25VF016B_Close Function

Closes an opened-instance of the SPI Flash driver.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened-instance of the SPI Flash driver, invalidating the handle.

Remarks

After calling this function, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_SST25VF016B_Open before the caller may use the driver again.

Usually, there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_SST25VF016B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF016B_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_SST25VF016B_Open

DRV_SST25VF016B_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

void DRV_SST25VF016B_Close(DRV_Handle handle);

DRV_SST25VF016B_Open Function

Opens the specified SPI Flash driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
DRV_HANDLE DRV_SST25VF016B_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the function returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_SST25VF016B_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver status is not ready.

The driver status becomes ready inside "DRV_SST25VF016B_Tasks" function. To make the SST Driver status ready and hence successfully "Open" the driver, "Task" routine need to be called periodically.

Description

This function opens the specified SPI Flash driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The driver will always work in Non-Blocking mode even if IO-intent is selected as blocking.

The handle returned is valid until the DRV_SST25VF016B_Close function is called.

This function will NEVER block waiting for hardware.

Preconditions

Function DRV_SST25VF016B_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_SST25VF016B_Open(DRV_SST25VF016B_INDEX_0,
                                                DRV_IO_INTENT_EXCLUSIVE);
```

```
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_SST25VF016B_Open
(
const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT ioIntent
);

DRV_SST25VF016B_ClientStatus Function

Gets current client-specific status of the SPI Flash driver.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
DRV_SST25VF016B_CLIENT_STATUS DRV_SST25VF016B_ClientStatus(const DRV_HANDLE handle);
```

Returns

A DRV_SST25VF016B_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the SPI Flash driver associated with the given handle.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_SST25VF016B_Initialize function must have been called.

DRV_SST25VF016B_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      handle;         // Returned from DRV_SST25VF016B_Open
DRV_SST25VF016B_CLIENT_STATUS     clientStatus;

clientStatus = DRV_SST25VF016B_ClientStatus(handle);
if(DRV_SST25VF016B_CLIENT_STATUS_READY == clientStatus)
{
    // do the tasks
}
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's open

Function

DRV_SST25VF016B_CLIENT_STATUS DRV_SST25VF016B_ClientStatus(DRV_HANDLE handle);

c) Block Operation Functions

DRV_SST25VF016B_BlockErase Function

Erase the specified number of blocks in Flash memory.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_BlockErase(const DRV_HANDLE handle, DRV_SST25VF016B_BLOCK_COMMAND_HANDLE *
commandHandle, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It Will be DRV_BUFFER_HANDLE_INVALID if the request was not queued.

Description

This function schedules a non-blocking erase operation in flash memory. The function returns with a valid erase handle in the commandHandle argument if the erase request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the client opened the driver for read only
- if nBlock is 0
- if the queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF016B_EVENT_ERASE_COMPLETE event if the erase operation was successful or DRV_SST25VF016B_EVENT_ERASE_ERROR event if the erase operation was not successful.

Remarks

Write Protection will be disabled for the complete flash memory region in the beginning by default.

Preconditions

The DRV_SST25VF016B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF016B_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF016B_Open call.

Example

```
// Destination address should be block aligned.
uint32_t blockStart;
uint32_t nBlock;
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF016BHandle is the handle returned
// by the DRV_SST25VF016B_Open function.

// Client registers an event handler with driver

DRV_SST25VF016B_BlockEventHandlerSet(mySST25VF016BHandle,
                APP_SST25VF016BEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF016B_BlockErase( mySST25VF016BHandle, commandHandle,
                                                    blockStart, nBlock );

if(DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer queue is processed.

void APP_SST25VF016BEventHandler(DRV_SST25VF016B_BLOCK_EVENT event,
        DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
```

```
        // contextHandle points to myAppObj.

        switch(event)
        {
            case DRV_SST25VF016B_EVENT_ERASE_COMPLETE:

                // This means the data was transferred.
                break;

            case DRV_SST25VF016B_EVENT_ERASE_ERROR:

                // Error handling here.

                break;

            default:
                break;
        }
    }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
blockStart	Start block address in SST25VF016B memory from where the erase should begin. LSBs (A0-A11) of block start address will be ignored to align it with Erase block size boundary.
nBlock	Total number of blocks to be erased. Each Erase block is of size 4 KByte.

Function

void DRV_SST25VF016B_BlockErase

(

const DRV_HANDLE handle,

DRV_SST25VF016B_BLOCK_COMMAND_HANDLE * commandHandle,

uint32_t blockStart,

uint32_t nBlock

);

DRV_SST25VF016B_BlockEventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_BlockEventHandlerSet(const DRV_HANDLE handle, const DRV_SST25VF016B_EVENT_HANDLER
eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client calls any read, write or erase function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any read/write/erase operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_SST25VF016B_Initialize function must have been called for the specified SPI FLash driver instance.

DRV_SST25VF016B_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle;

// mySST25VF016BHandle is the handle returned
// by the DRV_SST25VF016B_Open function.

// Client registers an event handler with driver. This is done once.

DRV_SST25VF016B_BlockEventHandlerSet( mySST25VF016BHandle,
                APP_SST25VF016BEventHandler, (uintptr_t)&myAppObj );

DRV_SST25VF016B_BlockRead( mySST25VF016BHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_SST25VF016BEventHandler(DRV_SST25VF016B_BLOCK_EVENT event,
        DRV_SST25VF016B_BLOCK_COMMAND_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_SST25VF016B_BlockEventHandlerSet

```
(
const      DRV_HANDLE handle,
const      DRV_SST25VF016B_EVENT_HANDLER eventHandler,
const uintptr_t context
);
```

DRV_SST25VF016B_BlockRead Function

Reads blocks of data starting from the specified address in Flash memory.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_BlockRead(const DRV_HANDLE handle, DRV_SST25VF016B_BLOCK_COMMAND_HANDLE *
commandHandle, uint8_t * targetBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking read operation for reading blocks of data from flash memory. The function returns with a valid handle in the commandHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the target buffer pointer is NULL
- if the client opened the driver for write only
- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully of DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

The maximum read speed is 33 MHz.

Preconditions

The DRV_SST25VF016B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF016B_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF016B_Open call.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = SST25VF016B_BASE_ADDRESS_TO_READ_FROM;
uint32_t    nBlock = 2;
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF016BHandle is the handle returned
// by the DRV_SST25VF016B_Open function.

// Client registers an event handler with driver

DRV_SST25VF016B_BlockEventHandlerSet(mySST25VF016BHandle,
                APP_SST25VF016BEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF016B_BlockRead( mySST25VF016BHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );
```

```
if(DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_SST25VF016BEventHandler(DRV_SST25VF016B_BLOCK_EVENT event,
        DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
*targetBuffer	Buffer into which the data read from the SPI Flash instance will be placed
blockStart	Start block address in SST25VF016B memory from where the read should begin. It can be any address of the flash.
nBlock	Total number of blocks to be read. Each Read block is of 1 byte.

Function

void DRV_SST25VF016B_BlockRead
(
const DRV_HANDLE handle,
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE * commandHandle,
uint8_t *targetBuffer,
uint32_t blockStart,
uint32_t nBlock
);

DRV_SST25VF016B_BlockWrite Function

Write blocks of data starting from a specified address in Flash memory.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
void DRV_SST25VF016B_BlockWrite(DRV_HANDLE handle, DRV_SST25VF016B_BLOCK_COMMAND_HANDLE * commandHandle,
uint8_t * sourceBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data into flash memory. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only
- if the buffer size is 0
- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

In the case of multi bytes write operation, byte by byte writing will happen instead of Address auto Increment writing.

Write Protection will be disabled for the complete flash memory region in the beginning by default.

Preconditions

The DRV_SST25VF016B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF016B_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF016B_Open call.

The flash address location which has to be written, must be erased before using the API DRV_SST25VF016B_BlockErase().

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = SST25VF016B_BASE_ADDRESS_TO_WRITE_TO;
uint32_t    nBlock = 2;
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF016BHandle is the handle returned
// by the DRV_SST25VF016B_Open function.

// Client registers an event handler with driver

DRV_SST25VF016B_BlockEventHandlerSet(mySST25VF016BHandle,
                APP_SST25VF016BEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF016B_BlockWrite( mySST25VF016BHandle, commandHandle,
                                      &myBuffer, blockStart, nBlock );

if(DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_SST25VF016BEventHandler(DRV_SST25VF016B_BLOCK_EVENT event,
        DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE:
```

```
            // This means the data was transferred.
            break;

        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function commandHandle -Pointer to an argument that will contain the return buffer handle
sourceBuffer	The source buffer containing data to be programmed into SPI Flash
blockStart	Start block address of SST25VF016B Flash where the write should begin. It can be any address of the flash.
nBlock	Total number of blocks to be written. Each write block is of 1 byte.

Function

void DRV_SST25VF016B_BlockWrite
(
DRV_HANDLE handle,
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE * commandHandle,
uint8_t *sourceBuffer,
uint32_t blockStart,
uint32_t nBlock
);

d) Media Interface Functions

DRV_SST25VF016B_GeometryGet Function

Returns the geometry of the device.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_SST25VF016B_GeometryGet(DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Structure which holds the media geometry information.

Description

This API gives the following geometrical details of the SST25VF016B Flash:

- Media Property
- Number of Read/Write/Erase regions in the flash device
- Number of Blocks and their size in each region of the device

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
SYS_FS_MEDIA_GEOMETRY * sstFlashGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

sstFlashGeometry = DRV_SST25VF016B_GeometryGet(sstOpenHandle1);

// read block size should be 1 byte
readBlockSize  = sstFlashGeometry->geometryTable->blockSize;
nReadBlocks = sstFlashGeometry->geometryTable->numBlocks;
nReadRegions = sstFlashGeometry->numReadRegions;

// write block size should be 1 byte
writeBlockSize  = (sstFlashGeometry->geometryTable +1)->blockSize;
// erase block size should be 4k byte
eraseBlockSize  = (sstFlashGeometry->geometryTable +2)->blockSize;

// total flash size should be 256k byte
totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY DRV_SST25VF016B_GeometryGet(DRV_HANDLE handle);

DRV_SST25VF016B_MediaIsAttached Function

Returns the status of the media.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
bool DRV_SST25VF016B_MediaIsAttached(DRV_HANDLE handle);
```

Returns

- True - Media is attached
- False - Media is not attached

Description

This API tells if the media is attached or not.

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
if (DRV_SST25VF016B_MediaIsAttached(handle))
{
// Do Something
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_SST25VF016B_MediaIsAttached(DRV_HANDLE handle);

e) Data Types and Constants

DRV_SST25VF016B_BLOCK_COMMAND_HANDLE Type

Handle identifying block commands of the driver.

File

drv_sst25vf016b.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_SST25VF016B_BLOCK_COMMAND_HANDLE;
```

Description

SPI Flash Driver Block Command Handle

A block command handle is returned by a call to the Read, Write, or Erase functions. This handle allows the application to track the completion of the operation. The handle is returned back to the client by the "event handler callback" function registered with the driver.

The handle assigned to a client request expires when the client has been notified of the completion of the operation (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_SST25VF016B_BLOCK_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_sst25vf016b.h

C

```
typedef enum {
  DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR
} DRV_SST25VF016B_BLOCK_EVENT;
```

Members

Members	Description
DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE	Block operation has been completed successfully. Read/Write/Erase Complete
DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR	There was an error during the block operation Read/Write/Erase Error

Description

SST25VF016B SPI Flash Driver Events

This enumeration identifies the possible events that can result from a Read, Write, or Erase request caused by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_SST25VF016B_BlockEventHandlerSet function when a block request is completed.

DRV_SST25VF016B_CLIENT_STATUS Enumeration

Defines the client status.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
typedef enum {
  DRV_SST25VF016B_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0,
  DRV_SST25VF016B_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY,
  DRV_SST25VF016B_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED,
  DRV_SST25VF016B_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR
} DRV_SST25VF016B_CLIENT_STATUS;
```

Members

Members	Description
DRV_SST25VF016B_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0	Up and running, ready to start new operations
DRV_SST25VF016B_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY	Operation in progress, unable to start a new one
DRV_SST25VF016B_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED	Client is closed
DRV_SST25VF016B_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR	Client Error

Description

SPI Flash Client Status

Defines the various client status codes.

Remarks

None.

DRV_SST25VF016B_EVENT_HANDLER Type

Pointer to a SST25VF016B SPI Flash Driver Event handler function.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
typedef void (* DRV_SST25VF016B_EVENT_HANDLER)(DRV_SST25VF016B_BLOCK_EVENT event,
DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t context);
```

Returns

None.

Description

SST25VF016B SPI Flash Driver Event Handler Function Pointer

This data type defines the required function signature for the SST25VF016B SPI Flash driver event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event calls back from the driver.

The parameters and return values and return value are described here and a partial example implementation is provided.

Remarks

If the event is DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE, it means that the data was transferred successfully.

If the event is DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR, it means that the data was not transferred successfully.

The context parameter contains the a handle to the client context, provided at the time the event handling function was registered using the DRV_SST25VF016B_BlockEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write/erase request.

The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

The Read, Write, and Erase functions can be called in the event handler to add a buffer to the driver queue. These functions can only be called to add buffers to the driver whose event handler is running.

Example

```
void APP_MyBufferEventHandler
(
    DRV_SST25VF016B_BLOCK_EVENT event,
    DRV_SST25VF016B_BLOCK_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_SST25VF016B_EVENT_BLOCK_COMMAND_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write/Erase requests
context	Value identifying the context of the application that registered the event handling function

DRV_SST25VF016B_INIT Structure

Contains all the data necessary to initialize the SPI Flash device.

Implementation: Dynamic

File

drv_sst25vf016b.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX spiDriverModuleIndex;
  PORTS_CHANNEL holdPortChannel;
  PORTS_BIT_POS holdBitPosition;
  PORTS_CHANNEL writeProtectPortChannel;
  PORTS_BIT_POS writeProtectBitPosition;
  PORTS_CHANNEL chipSelectPortChannel;
  PORTS_BIT_POS chipSelectBitPosition;
  uint32_t queueSize;
} DRV_SST25VF016B_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX spiDriverModuleIndex;	Identifies the SPI driver to be used
PORTS_CHANNEL holdPortChannel;	HOLD pin port channel
PORTS_BIT_POS holdBitPosition;	HOLD pin port position
PORTS_CHANNEL writeProtectPortChannel;	Write protect pin port channel
PORTS_BIT_POS writeProtectBitPosition;	Write Protect Bit pin position
PORTS_CHANNEL chipSelectPortChannel;	Chip select pin port channel
PORTS_BIT_POS chipSelectBitPosition;	Chip Select Bit pin position

uint32_t queueSize;	This is the buffer queue size. This is the maximum number of requests that this instance of the driver will queue. For a static build of the driver, this is overridden by the DRV_SST25VF016B_QUEUE_SIZE macro in system_config.h

Description

SST SPI Flash Driver Initialization Data

This structure contains all of the data necessary to initialize the SPI Flash device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_SST25VF016B_Initialize function.

DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID Macro

This value defines the SPI Flash Driver Block Command Invalid handle.

File

drv_sst25vf016b.h

C

```
#define DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID
```

Description

SPI Flash Driver Block Event Invalid Handle

This value defines the SPI Flash Driver Block Command Invalid handle. It is returned by read/write/erase routines when the request could not be taken.

Remarks

None.

DRV_SST25VF016B_INDEX_0 Macro

SPI Flash driver index definitions

File

drv_sst25vf016b.h

C

```
#define DRV_SST25VF016B_INDEX_0 0
```

Description

Driver SPI Flash Module Index reference

These constants provide SST25VF016B SPI Flash driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_SST25VF016B_Initialize and DRV_SST25VF016B_Open routines to identify the driver instance in use.

DRV_SST25VF016B_INDEX_1 Macro

File

drv_sst25vf016b.h

C

```
#define DRV_SST25VF016B_INDEX_1 1
```

Description

This is macro DRV_SST25VF016B_INDEX_1.

SST25VF020B API

a) System Functions

	Name	Description
	DRV_SST25VF020B_Initialize	Initializes the SST25VF020B SPI Flash Driver instance for the specified driver index. **Implementation:** Dynamic
	DRV_SST25VF020B_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module. **Implementation:** Dynamic
	DRV_SST25VF020B_Status	Gets the current status of the SPI Flash Driver module. **Implementation:** Dynamic
	DRV_SST25VF020B_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR. **Implementation:** Dynamic

b) Core Client Functions

	Name	Description
	DRV_SST25VF020B_ClientStatus	Gets current client-specific status of the SPI Flash driver. **Implementation:** Dynamic
	DRV_SST25VF020B_CommandStatus	Gets the current status of the command.
	DRV_SST25VF020B_Close	Closes an opened-instance of the SPI Flash driver. **Implementation:** Dynamic
	DRV_SST25VF020B_Open	Opens the specified SPI Flash driver instance and returns a handle to it. **Implementation:** Dynamic

c) Block Operation Functions

	Name	Description
	DRV_SST25VF020B_BlockErase	Erase the specified number of blocks in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockRead	Reads blocks of data starting from the specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockWrite	Write blocks of data starting from a specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockEraseWrite	Erase and Write blocks of data starting from a specified address in SST flash memory.

d) Media Interface Functions

	Name	Description
	DRV_SST25VF020B_GeometryGet	Returns the geometry of the device. **Implementation:** Dynamic
	DRV_SST25VF020B_MediaIsAttached	Returns the status of the media. **Implementation:** Dynamic

e) Data Types and Constants

	Name	Description
	DRV_SST25VF020B_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.
	DRV_SST25VF020B_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_SST25VF020B_CLIENT_STATUS	Defines the client status.
	DRV_SST25VF020B_EVENT_HANDLER	Pointer to a SST25VF020B SPI Flash Driver Event handler function.
	DRV_SST25VF020B_INIT	Contains all the data necessary to initialize the SPI Flash device.
	DRV_SST25VF020B_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.
	DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_SST25VF020B_INDEX_0	SPI Flash driver index definitions.

	DRV_SST25VF020B_INDEX_1	This is macro DRV_SST25VF020B_INDEX_1.

Description

This section contains the SST25V020B Flash device API.

a) System Functions

DRV_SST25VF020B_Initialize Function

Initializes the SST25VF020B SPI Flash Driver instance for the specified driver index.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
SYS_MODULE_OBJ DRV_SST25VF020B_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes the SPI Flash driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This function must be called before any other SPI Flash function is called.

This function should only be called once during system initialization unless DRV_SST25VF020B_Deinitialize is called to deinitialize the driver instance.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the SST25VF020B SPI
// Flash Driver. SPI driver index 0 is used for the purpose. Pin numbers 1, 2
// and 3 of port channel B are configured for hold pin, write protection pin
// and chip select pin respectively. Maximum buffer queue size is set 5.

DRV_SST25VF020B_INIT   SST25VF020BInitData;
SYS_MODULE_OBJ      objectHandle;

SST25VF020BInitData.moduleInit.value      = SYS_MODULE_POWER_RUN_FULL;
SST25VF020BInitData.spiDriverModuleIndex  = DRV_SPI_INDEX_0;
SST25VF020BInitData.holdPortChannel       = PORT_CHANNEL_B;
SST25VF020BInitData.holdBitPosition       = PORTS_BIT_POS_1;
SST25VF020BInitData.writeProtectPortChannel = PORT_CHANNEL_B;
SST25VF020BInitData.writeProtectBitPosition = PORTS_BIT_POS_2;
SST25VF020BInitData.chipSelectPortChannel = PORT_CHANNEL_F;
SST25VF020BInitData.chipSelectBitPosition = PORTS_BIT_POS_2;
SST25VF020BInitData.queueSize = 5;

objectHandle = DRV_SST25VF020B_Initialize(DRV_SST25VF020B_INDEX_0,
                                (SYS_MODULE_INIT*)SST25VF020BInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to a data structure containing data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_SST25VF020B_Initialize
(
const SYS_MODULE_INDEX index,
const SYS_MODULE_INIT * const init
);

DRV_SST25VF020B_Deinitialize Function

Deinitializes the specified instance of the SPI Flash driver module.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SPI Flash Driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This function will NEVER block waiting for hardware.

Preconditions

Function DRV_SST25VF020B_Initialize should have been called before calling this function.

Example

```
// This code snippet shows an example of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_SST25VF020B_Initialize
SYS_STATUS          status;


DRV_SST25VF020B_Deinitialize(object);

status = DRV_SST25VF020B_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SST25VF020B_Initialize

Function

void DRV_SST25VF020B_Deinitialize(SYS_MODULE_OBJ object)

DRV_SST25VF020B_Status Function

Gets the current status of the SPI Flash Driver module.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
SYS_STATUS DRV_SST25VF020B_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and accept requests for new operations

SYS_STATUS_UNINITIALIZED - Indicates that the driver is not initialized

Description

This function provides the current status of the SPI Flash Driver module.

Remarks

A driver can only be opened when its status is SYS_STATUS_READY.

Preconditions

Function DRV_SST25VF020B_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST25VF020B_Initialize
SYS_STATUS          SST25VF020BStatus;

SST25VF020BStatus = DRV_SST25VF020B_Status(object);
else if (SYS_STATUS_ERROR >= SST25VF020BStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SST25VF020B_Initialize

Function

SYS_STATUS DRV_SST25VF020B_Status(SYS_MODULE_OBJ object)

DRV_SST25VF020B_Tasks Function

Maintains the driver's read, erase, and write state machine and implements its ISR.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the driver's internal state machine and should be called from the system's Tasks function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks function (SYS_Tasks).

Preconditions

The DRV_SST25VF020B_Initialize function must have been called for the specified SPI Flash driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST25VF020B_Initialize

while (true)
{
    DRV_SST25VF020B_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_SST25VF020B_Initialize)

Function

void DRV_SST25VF020B_Tasks (SYS_MODULE_OBJ object);

b) Core Client Functions

DRV_SST25VF020B_ClientStatus Function

Gets current client-specific status of the SPI Flash driver.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
DRV_SST25VF020B_CLIENT_STATUS DRV_SST25VF020B_ClientStatus(const DRV_HANDLE handle);
```

Returns

A DRV_SST25VF020B_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the SPI Flash driver associated with the given handle.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_SST25VF020B_Initialize function must have been called.

DRV_SST25VF020B_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      handle;         // Returned from DRV_SST25VF020B_Open
DRV_SST25VF020B_CLIENT_STATUS     clientStatus;

clientStatus = DRV_SST25VF020B_ClientStatus(handle);
if(DRV_SST25VF020B_CLIENT_STATUS_READY == clientStatus)
{
    // do the tasks
}
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's open

Function

DRV_SST25VF020B_CLIENT_STATUS DRV_SST25VF020B_ClientStatus(DRV_HANDLE handle);

DRV_SST25VF020B_CommandStatus Function

Gets the current status of the command.

File

drv_sst25vf020b.h

C

```
DRV_SST25VF020B_COMMAND_STATUS DRV_SST25VF020B_CommandStatus(const DRV_HANDLE handle, const
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle);
```

Returns

A DRV_SST25VF020B_COMMAND_STATUS value describing the current status of the buffer. Returns DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID if the client handle or the command handle is not valid.

Description

This routine gets the current status of the buffer. The application must use this routine where the status of a scheduled buffer needs to polled on. The function may return DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID in a case where the buffer handle has expired. A buffer handle expires when the internal buffer object is re-assigned to another erase, read or write request. It is recommended that this function be called regularly in order to track the buffer status correctly.

The application can alternatively register an event handler to receive write, read or erase operation completion events.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

Block command request must have been made using Erase, Read or Write APIs to get a valid command handle.

Example

```
DRV_HANDLE      sstOpenHandle;   // Returned from DRV_SST25VF020B_Open
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE  commandHandle;
DRV_SST25VF020B_BlockErase
                        (
                            sstOpenHandle,
                            &commandHandle,
                            0,
                            1
                        );

if(DRV_SST25VF020B_CommandStatus(sstOpenHandle, commandHandle) == DRV_SST25VF020B_COMMAND_COMPLETED );
{
    // do something
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
commandHandle	A valid command handle, returned from Read/Write/Erase APIs.

Function

DRV_SST25VF020B_COMMAND_STATUS DRV_SST25VF020B_CommandStatus

(

const DRV_HANDLE handle,
const DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle
);

DRV_SST25VF020B_Close Function

Closes an opened-instance of the SPI Flash driver.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened-instance of the SPI Flash driver, invalidating the handle.

Remarks

After calling this function, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_SST25VF020B_Open before the caller may use the driver again.

Usually, there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_SST25VF020B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF020B_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_SST25VF020B_Open

DRV_SST25VF020B_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

void DRV_SST25VF020B_Close(DRV_Handle handle);

DRV_SST25VF020B_Open Function

Opens the specified SPI Flash driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
DRV_HANDLE DRV_SST25VF020B_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the function returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_SST25VF020B_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver status is not ready.

The driver status becomes ready inside "DRV_SST25VF020B_Tasks" function. To make the SST Driver status ready and hence successfully "Open" the driver, "Task" routine need to be called periodically.

Description

This function opens the specified SPI Flash driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The driver will always work in Non-Blocking mode even if IO-intent is selected as blocking.

The handle returned is valid until the DRV_SST25VF020B_Close function is called.

This function will NEVER block waiting for hardware.

Preconditions

Function DRV_SST25VF020B_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_SST25VF020B_Open(DRV_SST25VF020B_INDEX_0,
                                                DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_SST25VF020B_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

);

c) Block Operation Functions

DRV_SST25VF020B_BlockErase Function

Erase the specified number of blocks in Flash memory.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_BlockErase(const DRV_HANDLE handle, DRV_SST25VF020B_BLOCK_COMMAND_HANDLE *
commandHandle, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It Will be DRV_BUFFER_HANDLE_INVALID if the request was not queued.

Description

This function schedules a non-blocking erase operation in flash memory. The function returns with a valid erase handle in the commandHandle argument if the erase request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the client opened the driver for read only
- if nBlock is 0
- if the queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF020B_EVENT_ERASE_COMPLETE event if the erase operation was successful or DRV_SST25VF020B_EVENT_ERASE_ERROR event if the erase operation was not successful.

Remarks

Write Protection will be disabled for the complete flash memory region in the beginning by default.

Preconditions

The DRV_SST25VF020B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF020B_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF020B_Open call.

Example

```
// Destination address should be block aligned.
uint32_t blockStart;
uint32_t nBlock;
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF020BHandle is the handle returned
// by the DRV_SST25VF020B_Open function.

// Client registers an event handler with driver

DRV_SST25VF020B_BlockEventHandlerSet(mySST25VF020BHandle,
                APP_SST25VF020BEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF020B_BlockErase( mySST25VF020BHandle, commandHandle,
                                                    blockStart, nBlock );

if(DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer queue is processed.

void APP_SST25VF020BEventHandler(DRV_SST25VF020B_BLOCK_EVENT event,
        DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF020B_EVENT_ERASE_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF020B_EVENT_ERASE_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
blockStart	Start block address in SST25VF020B memory from where the erase should begin. LSBs (A0-A11) of block start address will be ignored to align it with Erase block size boundary.
nBlock	Total number of blocks to be erased. Each Erase block is of size 4 KByte.

Function

void DRV_SST25VF020B_BlockErase
(
const DRV_HANDLE handle,
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE * commandHandle,
uint32_t blockStart,
uint32_t nBlock
);

DRV_SST25VF020B_BlockEventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_BlockEventHandlerSet(const DRV_HANDLE handle, const DRV_SST25VF020B_EVENT_HANDLER
eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client calls any read, write or erase function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any read/write/erase operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_SST25VF020B_Initialize function must have been called for the specified SPI FLash driver instance.

DRV_SST25VF020B_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle;

// mySST25VF020BHandle is the handle returned
// by the DRV_SST25VF020B_Open function.

// Client registers an event handler with driver. This is done once.

DRV_SST25VF020B_BlockEventHandlerSet( mySST25VF020BHandle,
```

```
                APP_SST25VF020BEventHandler, (uintptr_t)&myAppObj );

DRV_SST25VF020B_BlockRead( mySST25VF020BHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_SST25VF020BEventHandler(DRV_SST25VF020B_BLOCK_EVENT event,
        DRV_SST25VF020B_BLOCK_COMMAND_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_SST25VF020B_BlockEventHandlerSet

(

const DRV_HANDLE handle,

const DRV_SST25VF020B_EVENT_HANDLER eventHandler,

const uintptr_t context

);

DRV_SST25VF020B_BlockRead Function

Reads blocks of data starting from the specified address in Flash memory.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_BlockRead(const DRV_HANDLE handle, DRV_SST25VF020B_BLOCK_COMMAND_HANDLE *
commandHandle, uint8_t * targetBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking read operation for reading blocks of data from flash memory. The function returns with a valid handle in the commandHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the target buffer pointer is NULL
- if the client opened the driver for write only
- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully of DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

The maximum read speed is 33 MHz.

Preconditions

The DRV_SST25VF020B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF020B_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF020B_Open call.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = SST25VF020B_BASE_ADDRESS_TO_READ_FROM;
uint32_t    nBlock = 2;
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF020BHandle is the handle returned
// by the DRV_SST25VF020B_Open function.

// Client registers an event handler with driver

DRV_SST25VF020B_BlockEventHandlerSet(mySST25VF020BHandle,
                APP_SST25VF020BEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF020B_BlockRead( mySST25VF020BHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_SST25VF020BEventHandler(DRV_SST25VF020B_BLOCK_EVENT event,
        DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;
```

```
            case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR:

                // Error handling here.

                break;

            default:
                break;
        }
    }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
*targetBuffer	Buffer into which the data read from the SPI Flash instance will be placed
blockStart	Start block address in SST25VF020B memory from where the read should begin. It can be any address of the flash.
nBlock	Total number of blocks to be read. Each Read block is of 1 byte.

Function

void DRV_SST25VF020B_BlockRead
(
const DRV_HANDLE handle,
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE * commandHandle,
uint8_t *targetBuffer,
uint32_t blockStart,
uint32_t nBlock
);

DRV_SST25VF020B_BlockWrite Function

Write blocks of data starting from a specified address in Flash memory.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_BlockWrite(DRV_HANDLE handle, DRV_SST25VF020B_BLOCK_COMMAND_HANDLE * commandHandle,
uint8_t * sourceBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data into flash memory. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only
- if the buffer size is 0
- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

In the case of multi bytes write operation, byte by byte writing will happen instead of Address auto Increment writing.

Write Protection will be disabled for the complete flash memory region in the beginning by default.

Preconditions

The DRV_SST25VF020B_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF020B_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF020B_Open call.

The flash address location which has to be written, must be erased before using the API DRV_SST25VF020B_BlockErase().

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = SST25VF020B_BASE_ADDRESS_TO_WRITE_TO;
uint32_t    nBlock = 2;
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF020BHandle is the handle returned
// by the DRV_SST25VF020B_Open function.

// Client registers an event handler with driver

DRV_SST25VF020B_BlockEventHandlerSet(mySST25VF020BHandle,
                APP_SST25VF020BEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF020B_BlockWrite( mySST25VF020BHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_SST25VF020BEventHandler(DRV_SST25VF020B_BLOCK_EVENT event,
        DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function commandHandle -Pointer to an argument that will contain the return buffer handle

sourceBuffer	The source buffer containing data to be programmed into SPI Flash
blockStart	Start block address of SST25VF020B Flash where the write should begin. It can be any address of the flash.
nBlock	Total number of blocks to be written. Each write block is of 1 byte.

Function

void DRV_SST25VF020B_BlockWrite
(
DRV_HANDLE handle,
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE * commandHandle,
uint8_t *sourceBuffer,
uint32_t blockStart,
uint32_t nBlock
);

DRV_SST25VF020B_BlockEraseWrite Function

Erase and Write blocks of data starting from a specified address in SST flash memory.

File

drv_sst25vf020b.h

C

```
void DRV_SST25VF020B_BlockEraseWrite(DRV_HANDLE hClient, DRV_SST25VF020B_BLOCK_COMMAND_HANDLE *
commandHandle, uint8_t * sourceBuffer, uint32_t blockStart, uint32_t nBlock);
```

Description

This function combines the step of erasing blocks of SST Flash and then writing the data. The application can use this function if it wants to avoid having to explicitly delete a block in order to update the bytes contained in the block.

This function schedules a non-blocking operation to erase and write blocks of data into SST flash. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only
- if the buffer size is 0
- if the queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_SST25VF020B_EVENT_ERASE_ERROR event if the buffer was not processed successfully.

Remarks

Refer to drv_sst25vf020b.h for usage information.

Function

void DRV_SST25VF020B_BlockEraseWrite
(
const DRV_HANDLE handle,
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE * commandHandle,
void * sourceBuffer,
uint32_t writeBlockStart,
uint32_t nWriteBlock
)

d) Media Interface Functions

DRV_SST25VF020B_GeometryGet Function

Returns the geometry of the device.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_SST25VF020B_GeometryGet(DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Structure which holds the media geometry information.

Description

This API gives the following geometrical details of the SST25VF020B Flash:

- Media Property
- Number of Read/Write/Erase regions in the flash device
- Number of Blocks and their size in each region of the device

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
SYS_FS_MEDIA_GEOMETRY * sstFlashGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

sstFlashGeometry = DRV_SST25VF020B_GeometryGet(sstOpenHandle1);

// read block size should be 1 byte
readBlockSize  = sstFlashGeometry->geometryTable->blockSize;
nReadBlocks = sstFlashGeometry->geometryTable->numBlocks;
nReadRegions = sstFlashGeometry->numReadRegions;

// write block size should be 1 byte
writeBlockSize  = (sstFlashGeometry->geometryTable +1)->blockSize;
// erase block size should be 4k byte
eraseBlockSize  = (sstFlashGeometry->geometryTable +2)->blockSize;

// total flash size should be 256k byte
totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY DRV_SST25VF020B_GeometryGet(DRV_HANDLE handle);

DRV_SST25VF020B_MediaIsAttached Function

Returns the status of the media.

Implementation: Dynamic

File

drv_sst25vf020b.h

C

```
bool DRV_SST25VF020B_MediaIsAttached(DRV_HANDLE handle);
```

Returns

- True - Media is attached
- False - Media is not attached

Description

This function determines whether or not the media is attached.

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
if (DRV_SST25VF020B_MediaIsAttached(handle))
{
// Do Something
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_SST25VF020B_MediaIsAttached(DRV_HANDLE handle);

e) Data Types and Constants

DRV_SST25VF020B_BLOCK_COMMAND_HANDLE Type

Handle identifying block commands of the driver.

File

drv_sst25vf020b.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_SST25VF020B_BLOCK_COMMAND_HANDLE;
```

Description

SPI Flash Driver Block Command Handle

A block command handle is returned by a call to the Read, Write, or Erase functions. This handle allows the application to track the completion of the operation. The handle is returned back to the client by the "event handler callback" function registered with the driver.

The handle assigned to a client request expires when the client has been notified of the completion of the operation (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_SST25VF020B_BLOCK_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_sst25vf020b.h

C

```
typedef enum {
```

```
  DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR
} DRV_SST25VF020B_BLOCK_EVENT;
```

Members

Members	Description
DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE	Block operation has been completed successfully. Read/Write/Erase Complete
DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR	There was an error during the block operation Read/Write/Erase Error

Description

SST25VF020B SPI Flash Driver Events

This enumeration identifies the possible events that can result from a Read, Write, or Erase request caused by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_SST25VF020B_BlockEventHandlerSet function when a block request is completed.

DRV_SST25VF020B_CLIENT_STATUS Enumeration

Defines the client status.

File

drv_sst25vf020b.h

C

```
typedef enum {
  DRV_SST25VF020B_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0,
  DRV_SST25VF020B_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY,
  DRV_SST25VF020B_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED,
  DRV_SST25VF020B_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR
} DRV_SST25VF020B_CLIENT_STATUS;
```

Members

Members	Description
DRV_SST25VF020B_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0	Up and running, ready to start new operations
DRV_SST25VF020B_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY	Operation in progress, unable to start a new one
DRV_SST25VF020B_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED	Client is closed
DRV_SST25VF020B_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR	Client Error

Description

SPI Flash Client Status

Defines the various client status codes.

Remarks

None.

DRV_SST25VF020B_EVENT_HANDLER Type

Pointer to a SST25VF020B SPI Flash Driver Event handler function.

File

drv_sst25vf020b.h

C

```
typedef void (* DRV_SST25VF020B_EVENT_HANDLER)(DRV_SST25VF020B_BLOCK_EVENT event,
DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t context);
```

Returns

None.

Description

SST25VF020B SPI Flash Driver Event Handler Function Pointer

This data type defines the required function signature for the SST25VF020B SPI Flash driver event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event calls back from the driver.

The parameters and return values and return value are described here and a partial example implementation is provided.

Remarks

If the event is DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE, it means that the data was transferred successfully.

If the event is DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR, it means that the data was not transferred successfully.

The context parameter contains the a handle to the client context, provided at the time the event handling function was registered using the DRV_SST25VF020B_BlockEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write/erase request.

The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

The Read, Write, and Erase functions can be called in the event handler to add a buffer to the driver queue. These functions can only be called to add buffers to the driver whose event handler is running.

Example

```
void APP_MyBufferEventHandler
(
    DRV_SST25VF020B_BLOCK_EVENT event,
    DRV_SST25VF020B_BLOCK_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_SST25VF020B_EVENT_BLOCK_COMMAND_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write/Erase requests
context	Value identifying the context of the application that registered the event handling function

DRV_SST25VF020B_INIT Structure

Contains all the data necessary to initialize the SPI Flash device.

File

drv_sst25vf020b.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX spiDriverModuleIndex;
  PORTS_CHANNEL holdPortChannel;
  PORTS_BIT_POS holdBitPosition;
```

```
  PORTS_CHANNEL writeProtectPortChannel;
  PORTS_BIT_POS writeProtectBitPosition;
  PORTS_CHANNEL chipSelectPortChannel;
  PORTS_BIT_POS chipSelectBitPosition;
  uint32_t queueSize;
} DRV_SST25VF020B_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX spiDriverModuleIndex;	Identifies the SPI driver to be used
PORTS_CHANNEL holdPortChannel;	HOLD pin port channel
PORTS_BIT_POS holdBitPosition;	HOLD pin port position
PORTS_CHANNEL writeProtectPortChannel;	Write protect pin port channel
PORTS_BIT_POS writeProtectBitPosition;	Write Protect Bit pin position
PORTS_CHANNEL chipSelectPortChannel;	Chip select pin port channel
PORTS_BIT_POS chipSelectBitPosition;	Chip Select Bit pin position
uint32_t queueSize;	This is the buffer queue size. This is the maximum number of requests that this instance of the driver will queue. For a static build of the driver, this is overridden by the DRV_SST25VF020B_QUEUE_SIZE macro in system_config.h

Description

SST SPI Flash Driver Initialization Data

This structure contains all of the data necessary to initialize the SPI Flash device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_SST25VF020B_Initialize function.

DRV_SST25VF020B_COMMAND_STATUS Enumeration

Specifies the status of the command for the read, write and erase operations.

File

drv_sst25vf020b.h

C

```
typedef enum {
  DRV_SST25VF020B_COMMAND_COMPLETED,
  DRV_SST25VF020B_COMMAND_QUEUED,
  DRV_SST25VF020B_COMMAND_IN_PROGRESS,
  DRV_SST25VF020B_COMMAND_ERROR_UNKNOWN
} DRV_SST25VF020B_COMMAND_STATUS;
```

Members

Members	Description
DRV_SST25VF020B_COMMAND_COMPLETED	Requested operation is completed
DRV_SST25VF020B_COMMAND_QUEUED	Scheduled but not started
DRV_SST25VF020B_COMMAND_IN_PROGRESS	Currently being in transfer
DRV_SST25VF020B_COMMAND_ERROR_UNKNOWN	Unknown Command

Description

SST Flash Driver Command Status

SST Flash Driver command Status

This type specifies the status of the command for the read, write and erase operations.

Remarks

None.

DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID Macro

This value defines the SPI Flash Driver Block Command Invalid handle.

File

drv_sst25vf020b.h

C

```
#define DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID
```

Description

SPI Flash Driver Block Event Invalid Handle

This value defines the SPI Flash Driver Block Command Invalid handle. It is returned by read/write/erase routines when the request could not be taken.

Remarks

None.

DRV_SST25VF020B_INDEX_0 Macro

SPI Flash driver index definitions.

File

drv_sst25vf020b.h

C

```
#define DRV_SST25VF020B_INDEX_0 0
```

Description

Driver SPI Flash Module Index reference

These constants provide SST25VF020B SPI Flash driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_SST25VF020B_Initialize and DRV_SST25VF020B_Open routines to identify the driver instance in use.

DRV_SST25VF020B_INDEX_1 Macro

File

drv_sst25vf020b.h

C

```
#define DRV_SST25VF020B_INDEX_1 1
```

Description

This is macro DRV_SST25VF020B_INDEX_1.

SST25VF064C API

a) System Functions

	Name	Description
	DRV_SST25VF064C_Initialize	Initializes the SST25VF064C SPI Flash Driver instance for the specified driver index.
	DRV_SST25VF064C_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module.
	DRV_SST25VF064C_Status	Gets the current status of the SPI Flash Driver module.
	DRV_SST25VF064C_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR.

b) Core Client Functions

	Name	Description
	DRV_SST25VF064C_ClientStatus	Gets current client-specific status of the SPI Flash driver.
	DRV_SST25VF064C_Close	Closes an opened-instance of the SPI Flash driver.
	DRV_SST25VF064C_CommandStatus	Gets the current status of the command.

	DRV_SST25VF064C_Open	Opens the specified SPI Flash driver instance and returns a handle to it.

c) Block Operation Functions

	Name	Description
	DRV_SST25VF064C_BlockErase	Erase the specified number of blocks in Flash memory.
	DRV_SST25VF064C_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.
	DRV_SST25VF064C_BlockRead	Reads blocks of data starting from the specified address in Flash memory.
	DRV_SST25VF064C_BlockWrite	Write blocks of data starting from a specified address in Flash memory.

d) Media Interface Functions

	Name	Description
	DRV_SST25VF064C_GeometryGet	Returns the geometry of the device.
	DRV_SST25VF064C_MediaIsAttached	Returns the status of the media.

e) Data Types and Constants

	Name	Description
	DRV_SST25VF064C_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.
	DRV_SST25VF064C_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_SST25VF064C_CLIENT_STATUS	Defines the client status.
	DRV_SST25VF064C_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.
	DRV_SST25VF064C_EVENT_HANDLER	Pointer to a SST25VF064C SPI Flash Driver Event handler function.
	DRV_SST25VF064C_INIT	Contains all the data necessary to initialize the SPI Flash device.
	DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_SST25VF064C_INDEX_0	SPI Flash driver index definitions.
	DRV_SST25VF064C_INDEX_1	This is macro DRV_SST25VF064C_INDEX_1.

Description

a) System Functions

DRV_SST25VF064C_Initialize Function

Initializes the SST25VF064C SPI Flash Driver instance for the specified driver index.

File

drv_sst25vf064c.h

C

```
SYS_MODULE_OBJ DRV_SST25VF064C_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes the SPI Flash driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This function must be called before any other SPI Flash function is called.

This function should only be called once during system initialization unless DRV_SST25VF064C_Deinitialize is called to deinitialize the driver instance.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the SST25VF064C SPI
// Flash Driver. SPI driver index 0 is used for the purpose. Pin numbers 1, 2
// and 3 of port channel B are configured for hold pin, write protection pin
// and chip select pin respectively. Maximum buffer queue size is set 5.

DRV_SST25VF064C_INIT   SST25VF064CInitData;
SYS_MODULE_OBJ      objectHandle;

SST25VF064CInitData.moduleInit.value      = SYS_MODULE_POWER_RUN_FULL;
SST25VF064CInitData.spiDriverModuleIndex  = DRV_SPI_INDEX_0;
SST25VF064CInitData.holdPortChannel       = PORT_CHANNEL_B;
SST25VF064CInitData.holdBitPosition       = PORTS_BIT_POS_1;
SST25VF064CInitData.writeProtectPortChannel = PORT_CHANNEL_B;
SST25VF064CInitData.writeProtectBitPosition = PORTS_BIT_POS_2;
SST25VF064CInitData.chipSelectPortChannel = PORT_CHANNEL_F;
SST25VF064CInitData.chipSelectBitPosition = PORTS_BIT_POS_2;
SST25VF064CInitData.queueSize = 5;

objectHandle = DRV_SST25VF064C_Initialize(DRV_SST25VF064C_INDEX_0,
                                (SYS_MODULE_INIT*)SST25VF064CInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to a data structure containing data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_SST25VF064C_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

);

DRV_SST25VF064C_Deinitialize Function

Deinitializes the specified instance of the SPI Flash driver module.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SPI Flash Driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This function will NEVER block waiting for hardware.

Preconditions

Function DRV_SST25VF064C_Initialize should have been called before calling this function.

Example

```
// This code snippet shows an example of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_SST25VF064C_Initialize
SYS_STATUS          status;


DRV_SST25VF064C_Deinitialize(object);

status = DRV_SST25VF064C_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SST25VF064C_Initialize

Function

void DRV_SST25VF064C_Deinitialize(SYS_MODULE_OBJ object)

DRV_SST25VF064C_Status Function

Gets the current status of the SPI Flash Driver module.

File

drv_sst25vf064c.h

C

```
SYS_STATUS DRV_SST25VF064C_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and accept requests for new operations

SYS_STATUS_UNINITIALIZED - Indicates that the driver is not initialized

Description

This function provides the current status of the SPI Flash Driver module.

Remarks

A driver can only be opened when its status is SYS_STATUS_READY.

Preconditions

Function DRV_SST25VF064C_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST25VF064C_Initialize
SYS_STATUS          SST25VF064CStatus;

SST25VF064CStatus = DRV_SST25VF064C_Status(object);
else if (SYS_STATUS_ERROR >= SST25VF064CStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SST25VF064C_Initialize

Function

SYS_STATUS DRV_SST25VF064C_Status(SYS_MODULE_OBJ object)

DRV_SST25VF064C_Tasks Function

Maintains the driver's read, erase, and write state machine and implements its ISR.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the driver's internal state machine and should be called from the system's Tasks function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks function (SYS_Tasks).

Preconditions

The DRV_SST25VF064C_Initialize function must have been called for the specified SPI Flash driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST25VF064C_Initialize

while (true)
{
    DRV_SST25VF064C_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_SST25VF064C_Initialize)

Function

void DRV_SST25VF064C_Tasks (SYS_MODULE_OBJ object);

b) Core Client Functions

DRV_SST25VF064C_ClientStatus Function

Gets current client-specific status of the SPI Flash driver.

File

drv_sst25vf064c.h

C

```
DRV_SST25VF064C_CLIENT_STATUS DRV_SST25VF064C_ClientStatus(const DRV_HANDLE handle);
```

Returns

A DRV_SST25VF064C_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the SPI Flash driver associated with the given handle.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_SST25VF064C_Initialize function must have been called.

DRV_SST25VF064C_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      handle;         // Returned from DRV_SST25VF064C_Open
DRV_SST25VF064C_CLIENT_STATUS     clientStatus;

clientStatus = DRV_SST25VF064C_ClientStatus(handle);
if(DRV_SST25VF064C_CLIENT_STATUS_READY == clientStatus)
{
    // do the tasks
}
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's open

Function

DRV_SST25VF064C_CLIENT_STATUS DRV_SST25VF064C_ClientStatus(DRV_HANDLE handle);

DRV_SST25VF064C_Close Function

Closes an opened-instance of the SPI Flash driver.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened-instance of the SPI Flash driver, invalidating the handle.

Remarks

After calling this function, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_SST25VF064C_Open before the caller may use the driver again.

Usually, there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_SST25VF064C_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF064C_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_SST25VF064C_Open

DRV_SST25VF064C_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

void DRV_SST25VF064C_Close(DRV_Handle handle);

DRV_SST25VF064C_CommandStatus Function

Gets the current status of the command.

File

drv_sst25vf064c.h

C

```
DRV_SST25VF064C_COMMAND_STATUS DRV_SST25VF064C_CommandStatus(const DRV_HANDLE handle, const
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle);
```

Returns

A DRV_SST25VF064C_COMMAND_STATUS value describing the current status of the buffer. Returns DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID if the client handle or the command handle is not valid.

Description

This routine gets the current status of the buffer. The application must use this routine where the status of a scheduled buffer needs to polled on. The function may return DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID in a case where the buffer handle has expired. A buffer handle expires when the internal buffer object is re-assigned to another erase, read or write request. It is recommended that this function be called regularly in order to track the buffer status correctly.

The application can alternatively register an event handler to receive write, read or erase operation completion events.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

Block command request must have been made using Erase, Read or Write APIs to get a valid command handle.

Example

```
DRV_HANDLE      sstOpenHandle;   // Returned from DRV_SST25VF064C_Open
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE  commandHandle;
DRV_SST25VF064C_BlockErase
                        (
                            sstOpenHandle,
                            &commandHandle,
                            0,
                            1
                        );

if(DRV_SST25VF064C_CommandStatus(sstOpenHandle, commandHandle) == DRV_SST25VF064C_COMMAND_COMPLETED );
{
    // do something
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
commandHandle	A valid command handle, returned from Read/Write/Erase APIs.

Function

DRV_SST25VF064C_COMMAND_STATUS DRV_SST25VF064C_CommandStatus

(

const DRV_HANDLE handle,

const DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle

);

DRV_SST25VF064C_Open Function

Opens the specified SPI Flash driver instance and returns a handle to it.

File

drv_sst25vf064c.h

C

```
DRV_HANDLE DRV_SST25VF064C_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the function returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_SST25VF064C_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.

Description

This function opens the specified SPI Flash driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The driver will always work in Non-Blocking mode even if IO-intent is selected as blocking.

The handle returned is valid until the DRV_SST25VF064C_Close function is called.

This function will NEVER block waiting for hardware.

Preconditions

Function DRV_SST25VF064C_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_SST25VF064C_Open(DRV_SST25VF064C_INDEX_0,
                                                DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_SST25VF064C_Open

(

const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT ioIntent

);

c) Block Operation Functions

DRV_SST25VF064C_BlockErase Function

Erase the specified number of blocks in Flash memory.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_BlockErase(const DRV_HANDLE handle, DRV_SST25VF064C_BLOCK_COMMAND_HANDLE *
commandHandle, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It Will be DRV_BUFFER_HANDLE_INVALID if the request was not queued.

Description

This function schedules a non-blocking erase operation in Flash memory. The function returns with a valid erase handle in the commandHandle argument if the erase request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the client opened the driver for read only
- if nBlock is 0
- if the queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF064C_EVENT_ERASE_COMPLETE event if the erase operation was successful or DRV_SST25VF064C_EVENT_ERASE_ERROR event if the erase operation was not successful.

Remarks

Write Protection will be disabled for the complete Flash memory region in the beginning by default.

Preconditions

The DRV_SST25VF064C_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF064C_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF064C_Open call.

Example

```
// Destination address should be block aligned.
uint32_t blockStart;
uint32_t nBlock;
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF064CHandle is the handle returned
// by the DRV_SST25VF064C_Open function.

// Client registers an event handler with driver

DRV_SST25VF064C_BlockEventHandlerSet(mySST25VF064CHandle,
                APP_SST25VF064CEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF064C_BlockErase( mySST25VF064CHandle, commandHandle,
                                                    blockStart, nBlock );

if(DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer queue is processed.

void APP_SST25VF064CEventHandler(DRV_SST25VF064C_BLOCK_EVENT event,
        DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF064C_EVENT_ERASE_COMPLETE:
```

```
            // This means the data was transferred.
            break;

        case DRV_SST25VF064C_EVENT_ERASE_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
blockStart	Start block address in SST25VF064C memory from where the erase should begin. LSBs (A0-A11) of block start address will be ignored to align it with Erase block size boundary.
nBlock	Total number of blocks to be erased. Each Erase block is of size 4 KByte.

Function

void DRV_SST25VF064C_BlockErase
(
const DRV_HANDLE handle,
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE * commandHandle,
uint32_t blockStart,
uint32_t nBlock
);

DRV_SST25VF064C_BlockEventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_BlockEventHandlerSet(const DRV_HANDLE handle, const DRV_SST25VF064C_EVENT_HANDLER
eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client calls any read, write or erase function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any read/write/erase operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_SST25VF064C_Initialize function must have been called for the specified SPI FLash driver instance.

DRV_SST25VF064C_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
```

```
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle;

// mySST25VF064CHandle is the handle returned
// by the DRV_SST25VF064C_Open function.

// Client registers an event handler with driver. This is done once.

DRV_SST25VF064C_BlockEventHandlerSet( mySST25VF064CHandle,
                APP_SST25VF064CEventHandler, (uintptr_t)&myAppObj );

DRV_SST25VF064C_BlockRead( mySST25VF064CHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_SST25VF064CEventHandler(DRV_SST25VF064C_BLOCK_EVENT event,
        DRV_SST25VF064C_BLOCK_COMMAND_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_SST25VF064C_BlockEventHandlerSet
(
const DRV_HANDLE handle,
const DRV_SST25VF064C_EVENT_HANDLER eventHandler,
const uintptr_t context
);

DRV_SST25VF064C_BlockRead Function

Reads blocks of data starting from the specified address in Flash memory.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_BlockRead(const DRV_HANDLE handle, DRV_SST25VF064C_BLOCK_COMMAND_HANDLE *
commandHandle, uint8_t * targetBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking read operation for reading blocks of data from Flash memory. The function returns with a valid handle in the commandHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the target buffer pointer is NULL
- if the client opened the driver for write only
- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully of DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

The maximum read speed is 33 MHz.

Preconditions

The DRV_SST25VF064C_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF064C_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF064C_Open call.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = SST25VF064C_BASE_ADDRESS_TO_READ_FROM;
uint32_t    nBlock = 2;
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF064CHandle is the handle returned
// by the DRV_SST25VF064C_Open function.

// Client registers an event handler with driver

DRV_SST25VF064C_BlockEventHandlerSet(mySST25VF064CHandle,
                APP_SST25VF064CEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF064C_BlockRead( mySST25VF064CHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.
```

```
void APP_SST25VF064CEventHandler(DRV_SST25VF064C_BLOCK_EVENT event,
        DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
*targetBuffer	Buffer into which the data read from the SPI Flash instance will be placed
blockStart	Start block address in SST25VF064C memory from where the read should begin. It can be any address of the Flash.
nBlock	Total number of blocks to be read. Each Read block is of 1 byte.

Function

void DRV_SST25VF064C_BlockRead

(

const DRV_HANDLE handle,

DRV_SST25VF064C_BLOCK_COMMAND_HANDLE * commandHandle,

uint8_t *targetBuffer,

uint32_t blockStart,

uint32_t nBlock

);

DRV_SST25VF064C_BlockWrite Function

Write blocks of data starting from a specified address in Flash memory.

File

drv_sst25vf064c.h

C

```
void DRV_SST25VF064C_BlockWrite(DRV_HANDLE handle, DRV_SST25VF064C_BLOCK_COMMAND_HANDLE * commandHandle,
uint8_t * sourceBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data into Flash memory. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request

- if the input buffer pointer is NULL
- if the client opened the driver for read only
- if the buffer size is 0
- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

In the case of multi bytes write operation, byte by byte writing will happen instead of Address auto Increment writing.

Write Protection will be disabled for the complete Flash memory region in the beginning by default.

Preconditions

The DRV_SST25VF064C_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_SST25VF064C_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_SST25VF064C_Open call.

The Flash address location which has to be written, must be erased before using the API DRV_SST25VF064C_BlockErase().

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = SST25VF064C_BASE_ADDRESS_TO_WRITE_TO;
uint32_t    nBlock = 2;
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST25VF064CHandle is the handle returned
// by the DRV_SST25VF064C_Open function.

// Client registers an event handler with driver

DRV_SST25VF064C_BlockEventHandlerSet(mySST25VF064CHandle,
                APP_SST25VF064CEventHandler, (uintptr_t)&myAppObj);

DRV_SST25VF064C_BlockWrite( mySST25VF064CHandle, commandHandle,
                                      &myBuffer, blockStart, nBlock );

if(DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_SST25VF064CEventHandler(DRV_SST25VF064C_BLOCK_EVENT event,
        DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
```

```
            break;
        }
    }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function commandHandle -Pointer to an argument that will contain the return buffer handle
sourceBuffer	The source buffer containing data to be programmed into SPI Flash
blockStart	Start block address of SST25VF064C Flash where the write should begin. It can be any address of the Flash.
nBlock	Total number of blocks to be written. Each write block is of 1 byte.

Function

```
void DRV_SST25VF064C_BlockWrite
(
    DRV_HANDLE handle,
    DRV_SST25VF064C_BLOCK_COMMAND_HANDLE * commandHandle,
uint8_t *sourceBuffer,
uint32_t blockStart,
uint32_t nBlock
);
```

d) Media Interface Functions

DRV_SST25VF064C_GeometryGet Function

Returns the geometry of the device.

File

drv_sst25vf064c.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_SST25VF064C_GeometryGet(DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Structure which holds the media geometry information.

Description

This API gives the following geometrical details of the SST25VF064C Flash:

- Media Property
- Number of Read/Write/Erase regions in the Flash device
- Number of Blocks and their size in each region of the device

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
SYS_FS_MEDIA_GEOMETRY * sstFlashGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

sstFlashGeometry = DRV_SST25VF064C_GeometryGet(sstOpenHandle1);

// read block size should be 1 byte
readBlockSize  = sstFlashGeometry->geometryTable->blockSize;
nReadBlocks = sstFlashGeometry->geometryTable->numBlocks;
```

```
nReadRegions = sstFlashGeometry->numReadRegions;

// write block size should be 1 byte
writeBlockSize  = (sstFlashGeometry->geometryTable +1)->blockSize;
// erase block size should be 4k byte
eraseBlockSize  = (sstFlashGeometry->geometryTable +2)->blockSize;

// total Flash size should be 8 MB
totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY DRV_SST25VF064C_GeometryGet(DRV_HANDLE handle);

DRV_SST25VF064C_MediaIsAttached Function

Returns the status of the media.

File

drv_sst25vf064c.h

C

```
bool DRV_SST25VF064C_MediaIsAttached(DRV_HANDLE handle);
```

Returns

- True - Media is attached
- False - Media is not attached

Description

This function determines whether or not the media is attached.

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
if (DRV_SST25VF064C_MediaIsAttached(handle))
{
// Do Something
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_SST25VF064C_MediaIsAttached(DRV_HANDLE handle);

e) Data Types and Constants

DRV_SST25VF064C_BLOCK_COMMAND_HANDLE Type

Handle identifying block commands of the driver.

File

drv_sst25vf064c.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_SST25VF064C_BLOCK_COMMAND_HANDLE;
```

Description

SPI Flash Driver Block Command Handle

A block command handle is returned by a call to the Read, Write, or Erase functions. This handle allows the application to track the completion of the operation. The handle is returned back to the client by the "event handler callback" function registered with the driver.

The handle assigned to a client request expires when the client has been notified of the completion of the operation (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_SST25VF064C_BLOCK_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_sst25vf064c.h

C

```
typedef enum {
  DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR
} DRV_SST25VF064C_BLOCK_EVENT;
```

Members

Members	Description
DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE	Block operation has been completed successfully. Read/Write/Erase Complete
DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR	There was an error during the block operation Read/Write/Erase Error

Description

SST25VF064C SPI Flash Driver Events

This enumeration identifies the possible events that can result from a Read, Write, or Erase request caused by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_SST25VF064C_BlockEventHandlerSet function when a block request is completed.

DRV_SST25VF064C_CLIENT_STATUS Enumeration

Defines the client status.

File

drv_sst25vf064c.h

C

```
typedef enum {
  DRV_SST25VF064C_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0,
  DRV_SST25VF064C_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY,
  DRV_SST25VF064C_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED,
  DRV_SST25VF064C_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR
} DRV_SST25VF064C_CLIENT_STATUS;
```

Members

Members	Description
DRV_SST25VF064C_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0	Up and running, ready to start new operations
DRV_SST25VF064C_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY	Operation in progress, unable to start a new one
DRV_SST25VF064C_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED	Client is closed

DRV_SST25VF064C_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR	Client Error

Description

SPI Flash Client Status

Defines the various client status codes.

Remarks

None.

DRV_SST25VF064C_COMMAND_STATUS Enumeration

Specifies the status of the command for the read, write and erase operations.

File

drv_sst25vf064c.h

C

```
typedef enum {
  DRV_SST25VF064C_COMMAND_COMPLETED,
  DRV_SST25VF064C_COMMAND_QUEUED,
  DRV_SST25VF064C_COMMAND_IN_PROGRESS,
  DRV_SST25VF064C_COMMAND_ERROR_UNKNOWN
} DRV_SST25VF064C_COMMAND_STATUS;
```

Members

Members	Description
DRV_SST25VF064C_COMMAND_COMPLETED	Requested operation is completed
DRV_SST25VF064C_COMMAND_QUEUED	Scheduled but not started
DRV_SST25VF064C_COMMAND_IN_PROGRESS	Currently being in transfer
DRV_SST25VF064C_COMMAND_ERROR_UNKNOWN	Unknown Command

Description

SST Flash Driver Command Status

SST Flash Driver command Status. This type specifies the status of the command for the read, write and erase operations.

Remarks

None.

DRV_SST25VF064C_EVENT_HANDLER Type

Pointer to a SST25VF064C SPI Flash Driver Event handler function.

File

drv_sst25vf064c.h

C

```
typedef void (* DRV_SST25VF064C_EVENT_HANDLER)(DRV_SST25VF064C_BLOCK_EVENT event,
DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t context);
```

Returns

None.

Description

SST25VF064C SPI Flash Driver Event Handler Function Pointer

This data type defines the required function signature for the SST25VF064C SPI Flash driver event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event calls back from the driver.

The parameters and return values and return value are described here and a partial example implementation is provided.

Remarks

If the event is DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE, it means that the data was transferred successfully.

If the event is DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR, it means that the data was not transferred successfully.

The context parameter contains the a handle to the client context, provided at the time the event handling function was registered using the DRV_SST25VF064C_BlockEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write/erase request.

The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

The Read, Write, and Erase functions can be called in the event handler to add a buffer to the driver queue. These functions can only be called to add buffers to the driver whose event handler is running.

Example

```
void APP_MyBufferEventHandler
(
    DRV_SST25VF064C_BLOCK_EVENT event,
    DRV_SST25VF064C_BLOCK_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_SST25VF064C_EVENT_BLOCK_COMMAND_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write/Erase requests
context	Value identifying the context of the application that registered the event handling function

DRV_SST25VF064C_INIT Structure

Contains all the data necessary to initialize the SPI Flash device.

File

drv_sst25vf064c.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX spiDriverModuleIndex;
  PORTS_CHANNEL holdPortChannel;
  PORTS_BIT_POS holdBitPosition;
  PORTS_CHANNEL writeProtectPortChannel;
  PORTS_BIT_POS writeProtectBitPosition;
  PORTS_CHANNEL chipSelectPortChannel;
  PORTS_BIT_POS chipSelectBitPosition;
  uint32_t queueSize;
} DRV_SST25VF064C_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX spiDriverModuleIndex;	Identifies the SPI driver to be used

PORTS_CHANNEL holdPortChannel;	HOLD pin port channel
PORTS_BIT_POS holdBitPosition;	HOLD pin port position
PORTS_CHANNEL writeProtectPortChannel;	Write protect pin port channel
PORTS_BIT_POS writeProtectBitPosition;	Write Protect Bit pin position
PORTS_CHANNEL chipSelectPortChannel;	Chip select pin port channel
PORTS_BIT_POS chipSelectBitPosition;	Chip Select Bit pin position
uint32_t queueSize;	This is the buffer queue size. This is the maximum number of requests that this instance of the driver will queue. For a static build of the driver, this is overridden by the DRV_SST25VF064C_QUEUE_SIZE macro in system_config.h

Description

SST SPI Flash Driver Initialization Data

This structure contains all of the data necessary to initialize the SPI Flash device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_SST25VF064C_Initialize function.

DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID Macro

This value defines the SPI Flash Driver Block Command Invalid handle.

File

drv_sst25vf064c.h

C

```
#define DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID
```

Description

SPI Flash Driver Block Event Invalid Handle

This value defines the SPI Flash Driver Block Command Invalid handle. It is returned by read/write/erase routines when the request could not be taken.

Remarks

None.

DRV_SST25VF064C_INDEX_0 Macro

SPI Flash driver index definitions.

File

drv_sst25vf064c.h

C

```
#define DRV_SST25VF064C_INDEX_0 0
```

Description

Driver SPI Flash Module Index reference

These constants provide SST25VF064C SPI Flash driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_SST25VF064C_Initialize and DRV_SST25VF064C_Open routines to identify the driver instance in use.

DRV_SST25VF064C_INDEX_1 Macro

File

drv_sst25vf064c.h

C

```
#define DRV_SST25VF064C_INDEX_1 1
```

Description

This is macro DRV_SST25VF064C_INDEX_1.

Files

Files

Name	Description
drv_sst25vf016b.h	SPI Flash Driver Interface Definition
drv_sst25vf016b_config_template.h	SST25VF016B Driver Configuration Template.
drv_sst25vf020b.h	SPI Flash Driver Interface Definition
drv_sst25vf020b_config_template.h	SST25VF020B Driver Configuration Template.
drv_sst25vf064c.h	SPI Flash Driver Interface Definition
drv_sst25vf064c_config_template.h	SST25VF064C Driver Configuration Template.

Description

This section lists the source and header files used by the SPI Flash Driver Library.

drv_sst25vf016b.h

SPI Flash Driver Interface Definition

Enumerations

	Name	Description
	DRV_SST25VF016B_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_SST25VF016B_CLIENT_STATUS	Defines the client status. **Implementation:** Dynamic

Functions

	Name	Description
	DRV_SST25VF016B_BlockErase	Erase the specified number of blocks in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF016B_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed. **Implementation:** Dynamic
	DRV_SST25VF016B_BlockRead	Reads blocks of data starting from the specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF016B_BlockWrite	Write blocks of data starting from a specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF016B_ClientStatus	Gets current client-specific status of the SPI Flash driver. **Implementation:** Dynamic
	DRV_SST25VF016B_Close	Closes an opened-instance of the SPI Flash driver. **Implementation:** Dynamic
	DRV_SST25VF016B_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module. **Implementation:** Dynamic
	DRV_SST25VF016B_GeometryGet	Returns the geometry of the device. **Implementation:** Dynamic
	DRV_SST25VF016B_Initialize	Initializes the SST25VF016B SPI Flash Driver instance for the specified driver index. **Implementation:** Dynamic
	DRV_SST25VF016B_MediaIsAttached	Returns the status of the media. **Implementation:** Dynamic
	DRV_SST25VF016B_Open	Opens the specified SPI Flash driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_SST25VF016B_Status	Gets the current status of the SPI Flash Driver module. **Implementation:** Dynamic

	DRV_SST25VF016B_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_SST25VF016B_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_SST25VF016B_INDEX_0	SPI Flash driver index definitions
	DRV_SST25VF016B_INDEX_1	This is macro DRV_SST25VF016B_INDEX_1.

Structures

	Name	Description
	DRV_SST25VF016B_INIT	Contains all the data necessary to initialize the SPI Flash device. **Implementation:** Dynamic

Types

	Name	Description
	DRV_SST25VF016B_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.
	DRV_SST25VF016B_EVENT_HANDLER	Pointer to a SST25VF016B SPI Flash Driver Event handler function. **Implementation:** Dynamic

Description

SPI Flash Driver Interface Definition

The SPI Flash device driver provides a simple interface to manage the SPI Flash modules which are external to Microchip Controllers. This file defines the interface definition for the SPI Flash Driver.

File Name

drv_sst25vf016b.h

Company

Microchip Technology Inc.

drv_sst25vf016b_config_template.h

SST25VF016B Driver Configuration Template.

Macros

	Name	Description
	DRV_SST25VF016B_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_SST25VF016B_HARDWARE_HOLD_ENABLE	Specifies if the hardware hold feature is enabled or not.
	DRV_SST25VF016B_HARDWARE_WRITE_PROTECTION_ENABLE	Specifies if the hardware write protect feature is enabled or not.
	DRV_SST25VF016B_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_SST25VF016B_MODE	Determines whether the driver is implemented as static or dynamic
	DRV_SST25VF016B_QUEUE_DEPTH_COMBINED	Number of entries of queues in all instances of the driver.

Description

SST25VF016B Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_sst25vf016b_config_template.h

Company

Microchip Technology Inc.

drv_sst25vf020b.h

SPI Flash Driver Interface Definition

Enumerations

	Name	Description
	DRV_SST25VF020B_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_SST25VF020B_CLIENT_STATUS	Defines the client status.
	DRV_SST25VF020B_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.

Functions

	Name	Description
	DRV_SST25VF020B_BlockErase	Erase the specified number of blocks in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockEraseWrite	Erase and Write blocks of data starting from a specified address in SST flash memory.
	DRV_SST25VF020B_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockRead	Reads blocks of data starting from the specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF020B_BlockWrite	Write blocks of data starting from a specified address in Flash memory. **Implementation:** Dynamic
	DRV_SST25VF020B_ClientStatus	Gets current client-specific status of the SPI Flash driver. **Implementation:** Dynamic
	DRV_SST25VF020B_Close	Closes an opened-instance of the SPI Flash driver. **Implementation:** Dynamic
	DRV_SST25VF020B_CommandStatus	Gets the current status of the command.
	DRV_SST25VF020B_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module. **Implementation:** Dynamic
	DRV_SST25VF020B_GeometryGet	Returns the geometry of the device. **Implementation:** Dynamic
	DRV_SST25VF020B_Initialize	Initializes the SST25VF020B SPI Flash Driver instance for the specified driver index. **Implementation:** Dynamic
	DRV_SST25VF020B_MediaIsAttached	Returns the status of the media. **Implementation:** Dynamic
	DRV_SST25VF020B_Open	Opens the specified SPI Flash driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_SST25VF020B_Status	Gets the current status of the SPI Flash Driver module. **Implementation:** Dynamic
	DRV_SST25VF020B_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_SST25VF020B_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_SST25VF020B_INDEX_0	SPI Flash driver index definitions.
	DRV_SST25VF020B_INDEX_1	This is macro DRV_SST25VF020B_INDEX_1.

Structures

	Name	Description
	DRV_SST25VF020B_INIT	Contains all the data necessary to initialize the SPI Flash device.

Types

	Name	Description
	DRV_SST25VF020B_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.

	DRV_SST25VF020B_EVENT_HANDLER	Pointer to a SST25VF020B SPI Flash Driver Event handler function.

Description

SPI Flash Driver Interface Definition

The SPI Flash device driver provides a simple interface to manage the SPI Flash modules which are external to Microchip Controllers. This file defines the interface definition for the SPI Flash Driver.

File Name

drv_sst25vf020b.h

Company

Microchip Technology Inc.

drv_sst25vf020b_config_template.h

SST25VF020B Driver Configuration Template.

Macros

	Name	Description
	DRV_SST25VF020B_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_SST25VF020B_HARDWARE_HOLD_ENABLE	Specifies if the hardware hold feature is enabled or not.
	DRV_SST25VF020B_HARDWARE_WRITE_PROTECTION_ENABLE	Specifies if the hardware write protect feature is enabled or not.
	DRV_SST25VF020B_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_SST25VF020B_MODE	Determines whether the driver is implemented as static or dynamic.
	DRV_SST25VF020B_QUEUE_DEPTH_COMBINED	Number of entries of queues in all instances of the driver.

Description

SST25VF020B Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_sst25vf020b_config_template.h

Company

Microchip Technology Inc.

drv_sst25vf064c.h

SPI Flash Driver Interface Definition

Enumerations

	Name	Description
	DRV_SST25VF064C_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_SST25VF064C_CLIENT_STATUS	Defines the client status.
	DRV_SST25VF064C_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.

Functions

	Name	Description
	DRV_SST25VF064C_BlockErase	Erase the specified number of blocks in Flash memory.
	DRV_SST25VF064C_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.
	DRV_SST25VF064C_BlockRead	Reads blocks of data starting from the specified address in Flash memory.
	DRV_SST25VF064C_BlockWrite	Write blocks of data starting from a specified address in Flash memory.
	DRV_SST25VF064C_ClientStatus	Gets current client-specific status of the SPI Flash driver.
	DRV_SST25VF064C_Close	Closes an opened-instance of the SPI Flash driver.

	DRV_SST25VF064C_CommandStatus	Gets the current status of the command.
	DRV_SST25VF064C_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module.
	DRV_SST25VF064C_GeometryGet	Returns the geometry of the device.
	DRV_SST25VF064C_Initialize	Initializes the SST25VF064C SPI Flash Driver instance for the specified driver index.
	DRV_SST25VF064C_MediaIsAttached	Returns the status of the media.
	DRV_SST25VF064C_Open	Opens the specified SPI Flash driver instance and returns a handle to it.
	DRV_SST25VF064C_Status	Gets the current status of the SPI Flash Driver module.
	DRV_SST25VF064C_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR.

Macros

	Name	Description
	DRV_SST25VF064C_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_SST25VF064C_INDEX_0	SPI Flash driver index definitions.
	DRV_SST25VF064C_INDEX_1	This is macro DRV_SST25VF064C_INDEX_1.

Structures

	Name	Description
	DRV_SST25VF064C_INIT	Contains all the data necessary to initialize the SPI Flash device.

Types

	Name	Description
	DRV_SST25VF064C_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.
	DRV_SST25VF064C_EVENT_HANDLER	Pointer to a SST25VF064C SPI Flash Driver Event handler function.

Description

SPI Flash Driver Interface Definition

The SPI Flash device driver provides a simple interface to manage the SPI Flash modules which are external to Microchip Controllers. This file defines the interface definition for the SPI Flash Driver.

File Name

drv_sst25vf064c.h

Company

Microchip Technology Inc.

drv_sst25vf064c_config_template.h

SST25VF064C Driver Configuration Template.

Macros

	Name	Description
	DRV_SST25VF064C_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_SST25VF064C_HARDWARE_HOLD_ENABLE	Specifies whether or not the hardware hold feature is enabled.
	DRV_SST25VF064C_HARDWARE_WRITE_PROTECTION_ENABLE	Specifies whether or not the hardware write protect feature is enabled.
	DRV_SST25VF064C_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_SST25VF064C_MODE	Determines whether the driver is implemented as static or dynamic.
	DRV_SST25VF064C_QUEUE_DEPTH_COMBINED	Number of entries of queues in all instances of the driver.

Description

SST25VF064C Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_sst25vf064c_config_template.h

Company

Microchip Technology Inc.

SPI PIC32WK IPF Flash Driver Library

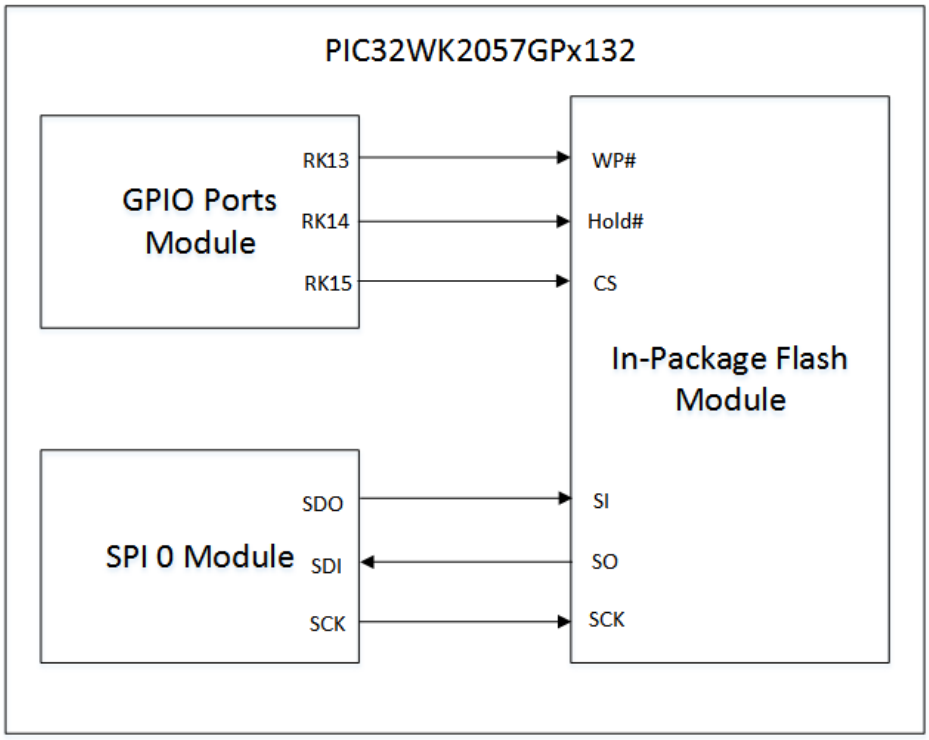
This section describes the Serial Peripheral Interface (SPI) Flash driver library for the PIC32WK IPF (in-package flash) module.

Introduction

This library provides an interface to manage the PIC32WK IPF module in different modes of operation.

Description

PIC32WK consists of an in-package flash (IPF) that is interfaced to the core using SPI, specifically the SPI0 instance. For more information, refer to the PIC32WK Silicon Data Sheet. The SPI module of the controller operates as a master device and the IPF module operates as a slave.



The PIC32WK IPF driver is dynamic in nature, therefore a single instance of it can support multiple clients that want to use the same flash. Multiple instances of the driver can be used when multiple flash devices are required to be part of the system. The SPI driver, which is used by the PIC32WK IPF driver, can be configured for use in either Polled or Interrupt mode.

Using the Library

This topic describes the basic architecture of the SPI PIC32WK IPF Flash Driver Library and provides information and examples about how to use it.

Description

Interface Header File: `drv_ipf.h`

The interface to the SPI PIC32WK IPF Flash Driver Library is defined in the `drv_ipf.h` header file. Any C language source (.c) file that uses the SPI PIC32WK IPF Flash Driver Library should include this header file.

Refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Library Source Files

The SPI PIC32WK IPF Flash Driver Library source files are provided in the `<install-dir>/framework/driver/spi_flash/pic32wk_ipf/src` folder. This folder may contain optional files and alternate

implementations. Refer to Configuring the Library for instructions about how to select optional features, and Building the Library for instructions about how to build the library.

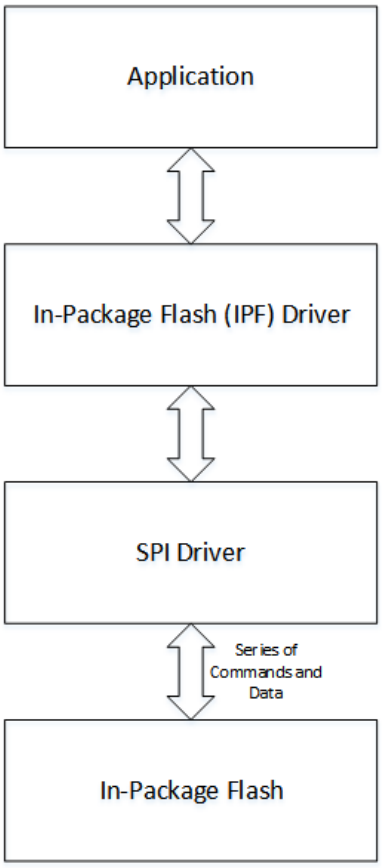
Abstraction Model

This section provides a low-level abstraction of the SPI PIC32WK IPF Flash Driver Library with a convenient C language interface. This topic describes how that abstraction is modeled in software.

Description

To perform a particular operation, the SPI PIC32WK IPF Flash Driver Library needs a specific set of commands to be given on its SPI interface along with the required address and data. The driver abstracts these requirements and provides simple APIs that can be used to perform Erase, Write, Read and memory protect operations. The SPI Driver is used for this purpose. The following layered diagram depicts the communication between different modules.

SPI PIC32WK IPF Flash Driver Library Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

Description

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the SPI PIC32WK IPF Flash Driver Library.

Library Interface Section	Description
System Functions	Accessed by the MPLAB Harmony system module and allow the driver to be initialized, de-initialized, and maintained.
Core Client Functions	Allow the application client to open and close the driver.
Block Operation Functions	Enable the Flash module to be erased, written, and read (to/from).
Media Interface Functions	Provide media status and the Flash geometry.
Memory Protection Functions	Functions protect or unprotect the required block of memory.

Pin Control Functions	Functions provide a means of controlling WP and Hold Pins.

How the Library Works

This topic describes the basic architecture of the SPI PIC32WK IPF Flash Driver Library and provides information and examples about its use.

Description

The library provides interfaces to support:

- System Initialization/Deinitialization
- Opening the Driver
- Block Operations

System Initialization/Deinitialization

This section provides information about initializing the system.

Description

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, the SPI PIC32WK IPF Flash Driver is initialized with the following configuration settings (either passed dynamically at runtime by using DRV_IPF_INIT or by using Initialization Overrides) that are supported or used by the IPF:

- Device-requested power state: one of the System Module Power States. For specific details please refer to "Data Types and Constants" in the Library Interface section.
- The SPI Driver Module Index, which is intended to be used to communicate with the IPF (for example, DRV_SPI_INDEX_0)
- Port Pins of the microcontroller to be used for Chip Select, Write Protection, and Hold operations on the IPF.
- Maximum Buffer Queue Size for that instance of the IPF.

The DRV_IPF_Initialize function returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the Initialize interface is used by the other system interfaces such as DRV_IPF_Deinitialize, DRV_ IPF _Status, and DRV_ IPF _Tasks.

Notes:

1. The system initialization and deinitialization settings affect only the instance of the peripheral that is being initialized or deinitialized.
2. As Hold, WP, and Chip select pins are internally routed, these are not configurable. Refer to the PIC32WK Silicon Data Sheet for more information.

Example:

```
// This code example shows the initialization of the In-Package Flash
// Driver. SPI driver index 0 is used for the purpose. Pin numbers 1, 2,
// and 3 of PORTB are configured for the Hold pin, Write Protection pin, and
// the Chip Select pin, respectively. The maximum buffer queue size is set to 5.
DRV_IPF_INIT IPFInitData;
SYS_MODULE_OBJ objectHandle;
IPFInitData.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
IPFInitData.spiDriverModuleIndex = DRV_SPI_INDEX_0;
IPFInitData.holdPortChannel = PORT_CHANNEL_K;
IPFInitData.holdBitPosition = PORTS_BIT_POS_14;
IPFInitData.writeProtectPortChannel = PORT_CHANNEL_K;
IPFInitData.writeProtectBitPosition = PORTS_BIT_POS_13;
IPFInitData.chipSelectPortChannel = PORT_CHANNEL_K;
IPFInitData.chipSelectBitPosition = PORTS_BIT_POS_15;
IPFInitData.queueSize = 5;

objectHandle = DRV_IPF_Initialize(DRV_IPF_INDEX_0,(SYS_MODULE_INIT*)IPFInitData);

if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
// Handle error
}
```

Tasks Routine

The system will call DRV_IPF_Tasks, from SYS_Tasks. .

Opening the Driver

This section provides information about opening the driver.

Description

To use the SST Flash driver, the application must open the driver. Using the SST25VF020B as an example, this is done by calling the DRV_IPF_Open function. Calling this function with DRV_IO_INTENT_NONBLOCKING will cause the driver to be opened in non blocking mode. Then DRV_IPF_BlockErase, DRV_IPF_BlockWrite and DRV_IPF_BlockRead functions when called by this client will be non-blocking.

The client can also open the driver in Read-only mode (DRV_IO_INTENT_READ), Write-only mode (DRV_IO_INTENT_WRITE), and Exclusive mode (DRV_IO_INTENT_EXCLUSIVE). If the driver has been opened exclusively by a client, it cannot be opened again by another client. If successful, the DRV_IPF_Open function will return a handle to the driver. This handle records the association between the client and the driver instance that was opened. The DRV_IPF_Open function may return DRV_HANDLE_INVALID in the situation where the driver is not ready to be opened. When this occurs, the application can try opening the driver again. Note that the open function may return and invalid handle in other (error) cases as well.

The following code shows an example of the driver being opened in different modes.

```
DRV_HANDLE ipfHandle1, ipfHandle2;
/* Client 1 opens the IPF driver in non blocking mode */
ipfHandle1 = DRV_IPF_Open(DRV_IPF_INDEX_0, DRV_IO_INTENT_NONBLOCKING);
/* Check if the handle is valid */
if(DRV_HANDLE_INVALID == ipfHandle1)
{
/* The driver was not opened successfully. The client
* can try opening it again */
}
/* Client 2 opens the IPF driver in Exclusive Write only mode */
ipfHandle2 = DRV_IPF_Open(DRV_IPF_INDEX_0, DRV_IO_INTENT_WRITE | DRV_IO_INTENT_EXCLUSIVE);
/* Check if the handle is valid */
if(DRV_HANDLE_INVALID == ipfHandle2)
{
/* The driver was not opened successfully. The client
* can try opening it again */
}
```

Block Operations

This section provides information about block operations.

Description

This driver provides simple client interfaces to Erase, Write, and Read the IPF in blocks. A block is the unit to represent minimum amount of data that can be erased, written, or read. Block size may differ for Erase, Write, and Read operations. The DRV_IPF_GeometryGet function can be used to determine the different block sizes for the driver.

The DRV_IPF_BlockErase, DRV_IPF_BlockWrite, and DRV_IPF_BlockRead functions are used to erase, write, and read the data to/from IPF. These functions are always non-blocking. All of these functions follow a standard queue model to read, write, and erase. When any of these functions are called (i.e., a block request is made), the request is queued. The size of the queue is determined by the queueSize member of the DRV_IPF_INIT data structure. All of the requests in the queue are executed by the DRV_IPF_Tasks function one-by-one.

When the driver adds a request to the queue, it returns a buffer handle. This handle allows the client to track the request as it progresses through the queue. The buffer handle expires when the event associated with the buffer completes. The driver provides driver events (DRV_IPF_BLOCK_EVENT) that indicate termination of the buffer requests.

For a simple Block Data Operation, perform the following steps :

1. The system should have completed necessary initialization of the SPI Driver and the IPF Driver, and the DRV_IPF_Tasks function should be running in a polled environment.
2. The DRV_SPI_Tasks function should be running in either a polled environment or an interrupt environment.
3. Open the driver using DRV_IPF_Open with the necessary intent.
4. Set an event handler callback using the function DRV_IPF_BlockEventHandlerSet.
5. Request for block operations using the functions, DRV_IPF_BlockErase, DRV_IPF_BlockWrite, and DRV_IPF_BlockRead, with the appropriate parameters.
6. Wait for event handler callback to occur and check the status of the block operation using the callback function parameter of type DRV_IPF_BLOCK_EVENT.
7. The client will be able to close the driver using the function, DRV_IPF_Close, when required.

Example:

```
/* This code example shows usage of the block operations
* on the PIC32WK IPF */
DRV_HANDLE ipfHandle1;
uint8_t myData1[10], myData2[10];
DRV_IPF_BLOCK_COMMAND_HANDLE blockHandle1, blockHandle2, blockHandle3;

/* The driver is opened for read-write in Exclusive mode */
ipfHandle1 = DRV_IPF_Open(DRV_IPF_INDEX_0,
```

```
DRV_IO_INTENT_READWRITE | DRV_IO_INTENT_EXCLUSIVE);

/* Check if the driver was opened successfully */
if(DRV_HANDLE_INVALID == ipfHandle1)
{
 /* The driver could not be opened successfully */
}

/* Register a Buffer Event Handler with IPF driver.
* This event handler function will be called whenever
* there is a buffer event. An application defined
* context can also be specified. This is returned when
* the event handler is called.
* */
DRV_IPF_BlockEventHandlerSet(sstHandle1,APP_IPFBufferEventHandler, NULL);

/* Request for all the three block operations one by one */
/* first block API to erase 1 block of the flash starting from address 0x0, each block is of 4kbyte */
DRV_IPF_BlockErase(ipfHandle1, &blockHandle1, 0x0, 1);

/* 2nd block API to write myData1 in the first 10 locations of the flash */
DRV_SST25VF020B_BlockWrite(ipfHandle1, &blockHandle2, &myData1[0], 0x0, 10);

/* 3rd block API to read the first 10 locations of the flash into myData2 */
DRV_SST25VF020B_BlockRead(ipfHandle1, &blockHandle3, &myData2[0], 0x0, 10);

/* This is the Driver Event Handler */
void APP_IPFBufferEventHandler(DRV_SST25VF020B_BLOCK_EVENT event, DRV_SST25VF020B_BLOCK_COMMAND_HANDLE
blockHandle, uintptr_t contextHandle)
{
 switch(event)
 {
 case DRV_IPF_EVENT_BLOCK_COMMAND_COMPLETE:
 if ( blockHandle == blockHandle3)
 {
 /* This means the data was read */
 /* Do data verification/processing */
 }
 break;
 case DRV_IPF_EVENT_BLOCK_COMMAND_ERROR:
 /* Error handling here. */
 break;
 default:
 break;
 }
```

Configuring the Library

Use this section for the drivers and system services and middleware. This section will contain any related configuration macros imported into the project from the companion `<library>_config_template.h` file into this topic, if applicable.

Description

The SPI PIC32WK IPF Flash Driver Library requires the specification of compile-time configuration macros. These macros define resource usage, feature availability, and dynamic behavior of the driver. These configuration macros should be defined in the `system_config.h` file.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. For more details, refer to Applications Help.

Building the Library

This section lists the files that are available in the SPI PIC32WK IPF Flash Driver Library.

Description

This section list the files that are available in the `\src` folder of the SPI PIC32WK IPF Flash Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/spi_flash/pic32wk_ipf`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_ipf.h	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_ipf.c	Basic SPI PIC32WK IPF Flash Driver implementation file.
/src/dynamic/drv_ipf_fs.c	File system functions used by the driver API.
/src/drv_ipf_prot.c	Protocol implementation used by the driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The SPI PIC32WK IPF Flash Driver Library depends on the following modules:

- Clock System Service Library

Optional Dependencies

- DMA System Service Library (used when operating in DMA mode)
- Interrupt System Service Library (used when task is running in Interrupt mode)

Library Interface

a) System Initialization Functions

	Name	Description
	DRV_IPF_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module. **Implementation:** Dynamic
	DRV_IPF_Initialize	Initializes the IPF SPI Flash Driver instance for the specified driver index. **Implementation:** Dynamic
	DRV_IPF_Status	Gets the current status of the SPI Flash Driver module. **Implementation:** Dynamic
	DRV_IPF_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR. **Implementation:** Dynamic

b) Client Setup Functions

	Name	Description
	DRV_IPF_ClientStatus	Gets current client-specific status of the SPI Flash driver. **Implementation:** Dynamic
	DRV_IPF_Close	Closes an opened-instance of the SPI Flash driver. **Implementation:** Dynamic
	DRV_IPF_Open	Opens the specified SPI Flash driver instance and returns a handle to it. **Implementation:** Dynamic

c) Other Functions

	Name	Description
	DRV_IPF_BlockErase	Erase the specified number of blocks in Flash memory. **Implementation:** Dynamic

	DRV_IPF_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed. **Implementation:** Dynamic
	DRV_IPF_BlockRead	Reads blocks of data starting from the specified address in Flash memory. **Implementation:** Dynamic
	DRV_IPF_BlockWrite	Write blocks of data starting from a specified address in Flash memory. **Implementation:** Dynamic
	DRV_IPF_GeometryGet	Returns the geometry of the device. **Implementation:** Dynamic
	DRV_IPF_HoldAssert	Asserts the Hold pin for flash. **Implementation:** Dynamic
	DRV_IPF_HoldDeAssert	Deasserts the Hold pin for flash. **Implementation:** Dynamic
	DRV_IPF_MediaIsAttached	Returns the status of the media. **Implementation:** Dynamic
	DRV_IPF_ProtectMemoryVolatile	Protects the memory block to which the given memory address belongs **Implementation:** Dynamic
	DRV_IPF_ReadBlockProtectionStatus	Reads the content of Block Protection Register which belongs to In-Package flash. **Implementation:** Dynamic
	DRV_IPF_UnProtectMemoryVolatile	Un-protects the memory block to which the given memory address belongs **Implementation:** Dynamic
	DRV_IPF_WPAssert	Asserts the WP pin for flash. **Implementation:** Dynamic
	DRV_IPF_WPDeAssert	Deasserts the WP pin for flash. **Implementation:** Dynamic

d) Data Types and Constants

	Name	Description
	DRV_IPF_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.
	DRV_IPF_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_IPF_BLOCK_OPERATION	Lists the different operations that IPF driver can do.
	DRV_IPF_CLIENT_STATUS	Defines the client status. **Implementation:** Dynamic
	DRV_IPF_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.
	DRV_IPF_EVENT_HANDLER	Pointer to a IPF SPI Flash Driver Event handler function. **Implementation:** Dynamic
	DRV_IPF_INIT	Contains all the data necessary to initialize the SPI Flash device. **Implementation:** Dynamic
	DRV_IPF_PROT_MODE	Lists the different memory protection modes.
	_DRV_IPF_H	This is macro _DRV_IPF_H.
	DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_IPF_INDEX_0	SPI Flash driver index definitions
	_DRV_IPF_CONFIG_TEMPLATE_H	This is macro _DRV_IPF_CONFIG_TEMPLATE_H.
	DRV_IPF_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to the hardware instance.
	DRV_IPF_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_IPF_MODE	Determines whether the driver is implemented as static or dynamic

Description

This section describes the API functions of the SPI PIC32WK IPF Flash Driver library.

Refer to each section for a detailed description.

a) System Initialization Functions

DRV_IPF_Deinitialize Function

Deinitializes the specified instance of the SPI Flash driver module.

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SPI Flash Driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This function will NEVER block waiting for hardware.

Preconditions

Function DRV_IPF_Initialize should have been called before calling this function.

Example

```
// This code snippet shows an example of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_IPF_Initialize
SYS_STATUS          status;


DRV_IPF_Deinitialize(object);

status = DRV_IPF_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_IPF_Initialize

Function

void DRV_IPF_Deinitialize(SYS_MODULE_OBJ object)

DRV_IPF_Initialize Function

Initializes the IPF SPI Flash Driver instance for the specified driver index.

Implementation: Dynamic

File

drv_ipf.h

C

```
SYS_MODULE_OBJ DRV_IPF_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes the SPI Flash driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This function must be called before any other SPI Flash function is called.

This function should only be called once during system initialization unless DRV_IPF_Deinitialize is called to deinitialize the driver instance.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the IPF SPI
// Flash Driver. SPI driver index 0 is used for the purpose. Pin numbers 1, 2
// and 3 of port channel B are configured for hold pin, write protection pin
// and chip select pin respectively. Maximum buffer queue size is set 5.

DRV_IPF_INIT   IPFInitData;
SYS_MODULE_OBJ      objectHandle;

IPFInitData.moduleInit.value      = SYS_MODULE_POWER_RUN_FULL;
IPFInitData.spiDriverModuleIndex  = DRV_SPI_INDEX_0;
IPFInitData.holdPortChannel       = PORT_CHANNEL_B;
IPFInitData.holdBitPosition       = PORTS_BIT_POS_1;
IPFInitData.writeProtectPortChannel = PORT_CHANNEL_B;
IPFInitData.writeProtectBitPosition = PORTS_BIT_POS_2;
IPFInitData.chipSelectPortChannel = PORT_CHANNEL_F;
IPFInitData.chipSelectBitPosition = PORTS_BIT_POS_2;
IPFInitData.queueSize = 5;

objectHandle = DRV_IPF_Initialize(DRV_IPF_INDEX_0,
                                  (SYS_MODULE_INIT*)IPFInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to a data structure containing data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_IPF_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

);

DRV_IPF_Status Function

Gets the current status of the SPI Flash Driver module.

Implementation: Dynamic

File

drv_ipf.h

C

```
SYS_STATUS DRV_IPF_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and accept requests for new operations

SYS_STATUS_UNINITIALIZED - Indicates that the driver is not initialized

Description

This function provides the current status of the SPI Flash Driver module.

Remarks

A driver can only be opened when its status is SYS_STATUS_READY.

Preconditions

Function DRV_IPF_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_IPF_Initialize
SYS_STATUS          IPFStatus;

IPFStatus = DRV_IPF_Status(object);
else if (SYS_STATUS_ERROR >= IPFStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_IPF_Initialize

Function

SYS_STATUS DRV_IPF_Status(SYS_MODULE_OBJ object)

DRV_IPF_Tasks Function

Maintains the driver's read, erase, and write state machine and implements its ISR.

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the driver's internal state machine and should be called from the system's Tasks function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks function (SYS_Tasks).

Preconditions

The DRV_IPF_Initialize function must have been called for the specified SPI Flash driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_IPF_Initialize

while (true)
{
    DRV_IPF_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_IPF_Initialize)

Function

void DRV_IPF_Tasks (SYS_MODULE_OBJ object);

b) Client Setup Functions

DRV_IPF_ClientStatus Function

Gets current client-specific status of the SPI Flash driver.

Implementation: Dynamic

File

drv_ipf.h

C

```
DRV_IPF_CLIENT_STATUS DRV_IPF_ClientStatus(const DRV_HANDLE handle);
```

Returns

A DRV_IPF_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the SPI Flash driver associated with the given handle.

Remarks

This function will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_IPF_Initialize function must have been called.

DRV_IPF_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      handle;         // Returned from DRV_IPF_Open
DRV_IPF_CLIENT_STATUS     clientStatus;

clientStatus = DRV_IPF_ClientStatus(handle);
if(DRV_IPF_CLIENT_STATUS_READY == clientStatus)
{
    // do the tasks
}
```

Parameters

Parameters	Description
handle	A valid open instance handle, returned from the driver's open

Function

DRV_IPF_CLIENT_STATUS DRV_IPF_ClientStatus(DRV_HANDLE handle);

DRV_IPF_Close Function

Closes an opened-instance of the SPI Flash driver.

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened-instance of the SPI Flash driver, invalidating the handle.

Remarks

After calling this function, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_IPF_Open before the caller may use the driver again.

Usually, there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_IPF_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_IPF_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_IPF_Open

DRV_IPF_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

void DRV_IPF_Close(DRV_Handle handle);

DRV_IPF_Open Function

Opens the specified SPI Flash driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_ipf.h

C

```
DRV_HANDLE DRV_IPF_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the function returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_IPF_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver status is not ready.

The driver status becomes ready inside "DRV_IPF_Tasks" function. To make the SST Driver status ready and hence successfully "Open" the driver, "Task" routine need to be called periodically.

Description

This function opens the specified SPI Flash driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The driver will always work in Non-Blocking mode even if IO-intent is selected as blocking.

The handle returned is valid until the DRV_IPF_Close function is called.

This function will NEVER block waiting for hardware.

Preconditions

Function DRV_IPF_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_IPF_Open(DRV_IPF_INDEX_0,
                                                DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Identifier for the object instance to be opened
ioIntent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_IPF_Open
(
const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT ioIntent
);

c) Other Functions

DRV_IPF_BlockErase Function

Erase the specified number of blocks in Flash memory.

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_BlockErase(const DRV_HANDLE handle, DRV_IPF_BLOCK_COMMAND_HANDLE * commandHandle, uint32_t
blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It Will be DRV_BUFFER_HANDLE_INVALID if the request was not queued.

Description

This function schedules a non-blocking erase operation in flash memory. The function returns with a valid erase handle in the commandHandle argument if the erase request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the client opened the driver for read only
- if nBlock is 0
- if the queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_IPF_EVENT_ERASE_COMPLETE event if the erase operation was successful or DRV_IPF_EVENT_ERASE_ERROR event if the erase operation was not successful.

Remarks

Write Protection will be disabled for the complete flash memory region in the beginning by default.

Preconditions

The DRV_IPF_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_IPF_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_IPF_Open call.

Example

```
// Destination address should be block aligned.
uint32_t blockStart;
uint32_t nBlock;
DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myIPFHandle is the handle returned
// by the DRV_IPF_Open function.

// Client registers an event handler with driver

DRV_IPF_BlockEventHandlerSet(myIPFHandle,
                APP_IPFEventHandler, (uintptr_t)&myAppObj);

DRV_IPF_BlockErase( myIPFHandle, commandHandle,
                                                    blockStart, nBlock );

if(DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer queue is processed.

void APP_IPFEventHandler(DRV_IPF_BLOCK_EVENT event,
        DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_IPF_EVENT_ERASE_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_IPF_EVENT_ERASE_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
blockStart	Start block address in IPF memory from where the erase should begin. LSBs (A0-A11) of block start address will be ignored to align it with Erase block size boundary.
nBlock	Total number of blocks to be erased. Each Erase block is of size 4 KByte.

Function

void DRV_IPF_BlockErase

(

```
const       DRV_HANDLE handle,
    DRV_IPF_BLOCK_COMMAND_HANDLE * commandHandle,
uint32_t blockStart,
uint32_t nBlock
);
```

DRV_IPF_BlockEventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_BlockEventHandlerSet(const DRV_HANDLE handle, const DRV_IPF_EVENT_HANDLER eventHandler, const
uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client calls any read, write or erase function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any read/write/erase operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_IPF_Initialize function must have been called for the specified SPI FLash driver instance.

DRV_IPF_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle;

// myIPFHandle is the handle returned
// by the DRV_IPF_Open function.

// Client registers an event handler with driver. This is done once.

DRV_IPF_BlockEventHandlerSet( myIPFHandle,
                APP_IPFEventHandler, (uintptr_t)&myAppObj );

DRV_IPF_BlockRead( myIPFHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_IPFEventHandler(DRV_IPF_BLOCK_EVENT event,
        DRV_IPF_BLOCK_COMMAND_HANDLE handle, uintptr_t context)
{
```

```
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_IPF_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_IPF_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_IPF_BlockEventHandlerSet

(

const DRV_HANDLE handle,

const DRV_IPF_EVENT_HANDLER eventHandler,

const uintptr_t context

);

DRV_IPF_BlockRead Function

Reads blocks of data starting from the specified address in Flash memory.

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_BlockRead(const DRV_HANDLE handle, DRV_IPF_BLOCK_COMMAND_HANDLE * commandHandle, uint8_t *
targetBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking read operation for reading blocks of data from flash memory. The function returns with a valid handle in the commandHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the target buffer pointer is NULL
- if the client opened the driver for write only
- if the buffer size is 0

- if the read queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_IPF_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully of DRV_IPF_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

The maximum read speed is 33 MHz.

Preconditions

The DRV_IPF_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_IPF_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_IPF_Open call.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = IPF_BASE_ADDRESS_TO_READ_FROM;
uint32_t    nBlock = 2;
DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myIPFHandle is the handle returned
// by the DRV_IPF_Open function.

// Client registers an event handler with driver

DRV_IPF_BlockEventHandlerSet(myIPFHandle,
                APP_IPFEventHandler, (uintptr_t)&myAppObj);

DRV_IPF_BlockRead( myIPFHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_IPFEventHandler(DRV_IPF_BLOCK_EVENT event,
        DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_IPF_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_IPF_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
*targetBuffer	Buffer into which the data read from the SPI Flash instance will be placed
blockStart	Start block address in IPF memory from where the read should begin. It can be any address of the flash.
nBlock	Total number of blocks to be read. Each Read block is of 1 byte.

Function

```
void DRV_IPF_BlockRead
(
const       DRV_HANDLE handle,
    DRV_IPF_BLOCK_COMMAND_HANDLE * commandHandle,
uint8_t *targetBuffer,
uint32_t blockStart,
uint32_t nBlock
);
```

DRV_IPF_BlockWrite Function

Write blocks of data starting from a specified address in Flash memory.

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_BlockWrite(DRV_HANDLE handle, DRV_IPF_BLOCK_COMMAND_HANDLE * commandHandle, uint8_t *
sourceBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data into flash memory. The function returns with a valid buffer handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only
- if the buffer size is 0
- if the write queue size is full or queue depth is insufficient
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_IPF_EVENT_BLOCK_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_IPF_EVENT_BLOCK_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

In the case of multi bytes write operation, byte by byte writing will happen instead of Address auto Increment writing.

Write Protection will be disabled for the complete flash memory region in the beginning by default.

Preconditions

The DRV_IPF_Initialize function must have been called for the specified SPI Flash driver instance.

DRV_IPF_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_IPF_Open call.

The flash address location which has to be written, must be erased before using the API DRV_IPF_BlockErase().

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// address should be block aligned.
uint32_t blockStart = IPF_BASE_ADDRESS_TO_WRITE_TO;
uint32_t    nBlock = 2;
DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myIPFHandle is the handle returned
// by the DRV_IPF_Open function.

// Client registers an event handler with driver

DRV_IPF_BlockEventHandlerSet(myIPFHandle,
                APP_IPFEventHandler, (uintptr_t)&myAppObj);

DRV_IPF_BlockWrite( myIPFHandle, commandHandle,
                                        &myBuffer, blockStart, nBlock );

if(DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_IPFEventHandler(DRV_IPF_BLOCK_EVENT event,
        DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_IPF_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_IPF_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function commandHandle -Pointer to an argument that will contain the return buffer handle
sourceBuffer	The source buffer containing data to be programmed into SPI Flash
blockStart	Start block address of IPF Flash where the write should begin. It can be any address of the flash.
nBlock	Total number of blocks to be written. Each write block is of 1 byte.

Function

void DRV_IPF_BlockWrite

(

DRV_HANDLE handle,

DRV_IPF_BLOCK_COMMAND_HANDLE * commandHandle,

uint8_t *sourceBuffer,
uint32_t blockStart,
uint32_t nBlock
);

DRV_IPF_GeometryGet Function

Returns the geometry of the device.

Implementation: Dynamic

File

drv_ipf.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_IPF_GeometryGet(DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Structure which holds the media geometry information.

Description

This API gives the following geometrical details of the IPF Flash:

- Media Property
- Number of Read/Write/Erase regions in the flash device
- Number of Blocks and their size in each region of the device

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
SYS_FS_MEDIA_GEOMETRY * sstFlashGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

sstFlashGeometry = DRV_IPF_GeometryGet(sstOpenHandle1);

// read block size should be 1 byte
readBlockSize  = sstFlashGeometry->geometryTable->blockSize;
nReadBlocks = sstFlashGeometry->geometryTable->numBlocks;
nReadRegions = sstFlashGeometry->numReadRegions;

// write block size should be 1 byte
writeBlockSize  = (sstFlashGeometry->geometryTable +1)->blockSize;
// erase block size should be 4k byte
eraseBlockSize  = (sstFlashGeometry->geometryTable +2)->blockSize;

// total flash size should be 256k byte
totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY DRV_IPF_GeometryGet(DRV_HANDLE handle);

DRV_IPF_HoldAssert Function

Asserts the Hold pin for flash.

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_HoldAssert();
```

Returns

None.

Description

This API is used to assert the Hold pin of the in-package flash.

Remarks

The Hold GPIO is fixed in case of PIC32WK devices.

Preconditions

None.

Example

```
DRV_IPF_HoldAssert();
```

Function

void DRV_IPF_HoldAssert();

DRV_IPF_HoldDeAssert Function

Deasserts the Hold pin for flash.

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_HoldDeAssert();
```

Returns

None.

Description

This API is used to deassert the Hold pin of the in-package flash.

Remarks

The Hold GPIO is fixed in case of PIC32WK devices.

Preconditions

None.

Example

```
DRV_IPF_HoldDeAssert();
```

Function

void DRV_IPF_HoldDeAssert();

DRV_IPF_MediaIsAttached Function

Returns the status of the media.

Implementation: Dynamic

File

drv_ipf.h

C

```
bool DRV_IPF_MediaIsAttached(DRV_HANDLE handle);
```

Returns

- True - Media is attached
- False - Media is not attached

Description

This API tells if the media is attached or not.

Remarks

This function is typically used by File System Media Manager.

Preconditions

None.

Example

```
if (DRV_IPF_MediaIsAttached(handle))
{
// Do Something
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_IPF_MediaIsAttached(DRV_HANDLE handle);

DRV_IPF_ProtectMemoryVolatile Function

Protects the memory block to which the given memory address belongs

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_ProtectMemoryVolatile(DRV_HANDLE clientHandle, DRV_IPF_BLOCK_COMMAND_HANDLE * commandHandle,
uintptr_t memAddress, DRV_IPF_PROT_MODE protMode);
```

Returns

None.

Description

This API is used to protect the memory block to which a given memory address belongs. Both read and write protection mode is supported. The memory will be protected until the next power cycle.

Remarks

Only the selected blocks can be read protected, which is as per the in-package flash specification.

Preconditions

In-package flash driver open function must be called and a valid client handle must be available.

Example

```
uintptr_t memAddr = IPF_ADDRESS_PROTECT;
DRV_IPF_PROT_MODE    protMode = DRV_IPF_WRITE_PROTECT;
DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myIPFHandle is the handle returned
// by the DRV_IPF_Open function.
```

```
// Client registers an event handler with driver

DRV_IPF_BlockEventHandlerSet(myIPFHandle,
                APP_IPFEventHandler, (uintptr_t)&myAppObj);

DRV_IPF_ProtectMemoryVolatile( myIPFHandle, commandHandle,
                                        memAddr, protMode );

if(DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_IPFEventHandler(DRV_IPF_BLOCK_EVENT event,
        DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_IPF_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the memory protection is complete.
            break;

        case DRV_IPF_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
clientHandle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
memAddress	Memory address which belongs to the memory block which needs to be protected
protMode	Read or write protect mode. If a block needs to be protected for both read and write, then both enum values can be ORed and passed to the function.

Function

void DRV_IPF_ProtectMemoryVolatile
(
    DRV_HANDLE clientHandle,
    DRV_IPF_BLOCK_COMMAND_HANDLE * commandHandle,
uintptr_t memAddress,
    DRV_IPF_PROT_MODE protMode
);

DRV_IPF_ReadBlockProtectionStatus Function

Reads the content of Block Protection Register which belongs to In-Package flash.

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_ReadBlockProtectionStatus(DRV_HANDLE clientHandle, DRV_IPF_BLOCK_COMMAND_HANDLE *
commandHandle, uint8_t * buffer);
```

Returns

None.

Description

This API is read the current contents of the block protection register in in-package flash and fills the buffer passed by the client.

Remarks

The block protection word is 6-bytes wide.

Preconditions

In-package flash driver open function must be called and a valid client handle must be available.

Example

```
uint8_t buf[6] = {0,};
DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myIPFHandle is the handle returned
// by the DRV_IPF_Open function.

// Client registers an event handler with driver

DRV_IPF_BlockEventHandlerSet(myIPFHandle,
                APP_IPFEventHandler, (uintptr_t)&myAppObj);

DRV_IPF_ReadBlockProtectionStatus( myIPFHandle, commandHandle,
                                        buf );

if(DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_IPFEventHandler(DRV_IPF_BLOCK_EVENT event,
        DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_IPF_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the BPR read is complete.
            break;

        case DRV_IPF_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
clientHandle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
buffer	pointer to a buffer to which the block protection status has to be updated

Function

```
void DRV_IPF_ReadBlockProtectionStatus
(
    DRV_HANDLE clientHandle,
    DRV_IPF_BLOCK_COMMAND_HANDLE * commandHandle,
uint8_t * buffer
);
```

DRV_IPF_UnProtectMemoryVolatile Function

Un-protects the memory block to which the given memory address belongs

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_UnProtectMemoryVolatile(DRV_HANDLE clientHandle, DRV_IPF_BLOCK_COMMAND_HANDLE * commandHandle,
uintptr_t memAddress, DRV_IPF_PROT_MODE protMode);
```

Returns

None.

Description

This API is used to un-protect the memory block to which a given memory address belongs. Both read and write protection mode is supported. The memory will be protected until the next power cycle.

Remarks

If the memory block a client is trying to unprotect, is protected by some other client, then memory unprotection will not executed. The function will return without unprotecting.

Preconditions

In-package flash driver open function must be called and a valid client handle must be available.

Example

```
uintptr_t memAddr = IPF_ADDRESS_UNPROTECT;
DRV_IPF_PROT_MODE    protMode = DRV_IPF_WRITE_PROTECT;
DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// myIPFHandle is the handle returned
// by the DRV_IPF_Open function.

// Client registers an event handler with driver

DRV_IPF_BlockEventHandlerSet(myIPFHandle,
                APP_IPFEventHandler, (uintptr_t)&myAppObj);

DRV_IPF_UnProtectMemoryVolatile( myIPFHandle, commandHandle,
                                        memAddr, protMode );

if(DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}
```

```
// Event is received when
// the buffer is processed.

void APP_IPFEventHandler(DRV_IPF_BLOCK_EVENT event,
        DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_IPF_EVENT_BLOCK_COMMAND_COMPLETE:

            // This means the memory unprotection is complete.
            break;

        case DRV_IPF_EVENT_BLOCK_COMMAND_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
clientHandle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
memAddress	Memory address which belongs to the memory block which needs to be un-protected
protMode	Read or write protect mode. If a block needs to be un-protected for both read and write, then both enum values can be ORed and passed to the function.

Function

void DRV_IPF_UnProtectMemoryVolatile
(
DRV_HANDLE clientHandle,
DRV_IPF_BLOCK_COMMAND_HANDLE * commandHandle,
uintptr_t memAddress,
DRV_IPF_PROT_MODE protMode
);

DRV_IPF_WPAssert Function

Asserts the WP pin for flash.

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_WPAssert();
```

Returns

None.

Description

This API is used to assert the Write Protect (WP) pin of the in-package flash.

Remarks

The Write Protection GPIO is fixed in case of PIC32WK devices.

Preconditions

None.

Example

```
DRV_IPF_WPAssert();
```

Function

void DRV_IPF_WPAssert();

DRV_IPF_WPDeAssert Function

Deasserts the WP pin for flash.

Implementation: Dynamic

File

drv_ipf.h

C

```
void DRV_IPF_WPDeAssert();
```

Returns

None.

Description

This API is used to deassert the Write Protect (WP) pin of the in-package flash.

Remarks

The Write Protection GPIO is fixed in case of PIC32WK devices.

Preconditions

None.

Example

```
DRV_IPF_WPDeAssert();
```

Function

void DRV_IPF_WPAssert();

d) Data Types and Constants

DRV_IPF_BLOCK_COMMAND_HANDLE Type

Handle identifying block commands of the driver.

File

drv_ipf.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_IPF_BLOCK_COMMAND_HANDLE;
```

Description

SPI Flash Driver Block Command Handle

A block command handle is returned by a call to the Read, Write, or Erase functions. This handle allows the application to track the completion of the operation. The handle is returned back to the client by the "event handler callback" function registered with the driver.

The handle assigned to a client request expires when the client has been notified of the completion of the operation (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_IPF_BLOCK_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_ipf.h

C

```
typedef enum {
  DRV_IPF_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_IPF_EVENT_BLOCK_COMMAND_ERROR
} DRV_IPF_BLOCK_EVENT;
```

Members

Members	Description
DRV_IPF_EVENT_BLOCK_COMMAND_COMPLETE	Block operation has been completed successfully. Read/Write/Erase Complete
DRV_IPF_EVENT_BLOCK_COMMAND_ERROR	There was an error during the block operation Read/Write/Erase Error

Description

IPF SPI Flash Driver Events

This enumeration identifies the possible events that can result from a Read, Write, or Erase request caused by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_IPF_BlockEventHandlerSet function when a block request is completed.

DRV_IPF_BLOCK_OPERATION Enumeration

Lists the different operations that IPF driver can do.

File

drv_ipf.h

C

```
typedef enum {
  DRV_IPF_BLOCK_READ,
  DRV_IPF_BLOCK_WRITE,
  DRV_IPF_BLOCK_ERASE,
  DRV_IPF_HW_BLOCK_PROT,
  DRV_IPF_HW_BLOCK_UNPROT,
  DRV_IPF_READ_HW_BLOCK_PROT
} DRV_IPF_BLOCK_OPERATION;
```

Members

Members	Description
DRV_IPF_BLOCK_READ	Block Read
DRV_IPF_BLOCK_WRITE	Block Write
DRV_IPF_BLOCK_ERASE	Block Erase
DRV_IPF_HW_BLOCK_PROT	Hardware Block Protection
DRV_IPF_HW_BLOCK_UNPROT	Hardware Block Un-Protection
DRV_IPF_READ_HW_BLOCK_PROT	Read HW Block Protection Status

Description

IPF Driver Operations

This enumeration lists the different operations that IPF driver can do.

Remarks

None.

DRV_IPF_CLIENT_STATUS Enumeration

Defines the client status.

Implementation: Dynamic

File

drv_ipf.h

C

```
typedef enum {
  DRV_IPF_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0,
  DRV_IPF_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY,
  DRV_IPF_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED,
  DRV_IPF_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR
} DRV_IPF_CLIENT_STATUS;
```

Members

Members	Description
DRV_IPF_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY+0	Up and running, ready to start new operations
DRV_IPF_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY	Operation in progress, unable to start a new one
DRV_IPF_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED	Client is closed
DRV_IPF_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR	Client Error

Description

SPI Flash Client Status

Defines the various client status codes.

Remarks

None.

DRV_IPF_COMMAND_STATUS Enumeration

Specifies the status of the command for the read, write and erase operations.

File

drv_ipf.h

C

```
typedef enum {
  DRV_IPF_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED,
  DRV_IPF_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED,
  DRV_IPF_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS,
  DRV_IPF_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN
} DRV_IPF_COMMAND_STATUS;
```

Members

Members	Description
DRV_IPF_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED	Done OK and ready
DRV_IPF_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED	Scheduled but not started
DRV_IPF_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS	Currently being in transfer
DRV_IPF_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN	Unknown Command

Description

IPF Driver Command Status

IPF Driver command Status

This type specifies the status of the command for the read, write and erase operations.

Remarks

None.

DRV_IPF_EVENT_HANDLER Type

Pointer to a IPF SPI Flash Driver Event handler function.

Implementation: Dynamic

File

drv_ipf.h

C

```
typedef void (* DRV_IPF_EVENT_HANDLER)(DRV_IPF_BLOCK_EVENT event, DRV_IPF_BLOCK_COMMAND_HANDLE
commandHandle, uintptr_t context);
```

Returns

None.

Description

IPF SPI Flash Driver Event Handler Function Pointer

This data type defines the required function signature for the IPF SPI Flash driver event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event calls back from the driver.

The parameters and return values and return value are described here and a partial example implementation is provided.

Remarks

If the event is DRV_IPF_EVENT_BLOCK_COMMAND_COMPLETE, it means that the data was transferred successfully.

If the event is DRV_IPF_EVENT_BLOCK_COMMAND_ERROR, it means that the data was not transferred successfully.

The context parameter contains the a handle to the client context, provided at the time the event handling function was registered using the DRV_IPF_BlockEventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write/erase request.

The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations with in this function.

The Read, Write, and Erase functions can be called in the event handler to add a buffer to the driver queue. These functions can only be called to add buffers to the driver whose event handler is running.

Example

```
void APP_MyBufferEventHandler
(
    DRV_IPF_BLOCK_EVENT event,
    DRV_IPF_BLOCK_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_IPF_EVENT_BLOCK_COMMAND_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_IPF_EVENT_BLOCK_COMMAND_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write/Erase requests
context	Value identifying the context of the application that registered the event handling function

DRV_IPF_INIT Structure

Contains all the data necessary to initialize the SPI Flash device.

Implementation: Dynamic

File

drv_ipf.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  SYS_MODULE_INDEX spiDriverModuleIndex;
  PORTS_CHANNEL holdPortChannel;
  PORTS_BIT_POS holdBitPosition;
  PORTS_CHANNEL writeProtectPortChannel;
  PORTS_BIT_POS writeProtectBitPosition;
  PORTS_CHANNEL chipSelectPortChannel;
  PORTS_BIT_POS chipSelectBitPosition;
  uint32_t queueSize;
} DRV_IPF_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX spiDriverModuleIndex;	Identifies the SPI driver to be used
PORTS_CHANNEL holdPortChannel;	HOLD pin port channel
PORTS_BIT_POS holdBitPosition;	HOLD pin port position
PORTS_CHANNEL writeProtectPortChannel;	Write protect pin port channel
PORTS_BIT_POS writeProtectBitPosition;	Write Protect Bit pin position
PORTS_CHANNEL chipSelectPortChannel;	Chip select pin port channel
PORTS_BIT_POS chipSelectBitPosition;	Chip Select Bit pin position
uint32_t queueSize;	This is the buffer queue size. This is the maximum number of requests that this instance of the driver will queue. For a static build of the driver, this is overridden by the DRV_IPF_QUEUE_SIZE macro in system_config.h

Description

SST SPI Flash Driver Initialization Data

This structure contains all of the data necessary to initialize the SPI Flash device.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the DRV_IPF_Initialize function.

DRV_IPF_PROT_MODE Enumeration

Lists the different memory protection modes.

File

drv_ipf.h

C

```
typedef enum {
  DRV_IPF_WRITE_PROTECT = 1,
  DRV_IPF_READ_PROTECT
} DRV_IPF_PROT_MODE;
```

Members

Members	Description
DRV_IPF_WRITE_PROTECT = 1	Write Protect
DRV_IPF_READ_PROTECT	Read Protect

Description

IPF Driver memory protection modes

This enumeration lists the different memory protection modes.

Remarks

None.

_DRV_IPF_H Macro

File

drv_ipf.h

C

```
#define _DRV_IPF_H
```

Description

This is macro _DRV_IPF_H.

DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID Macro

This value defines the SPI Flash Driver Block Command Invalid handle.

File

drv_ipf.h

C

```
#define DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID
```

Description

SPI Flash Driver Block Event Invalid Handle

This value defines the SPI Flash Driver Block Command Invalid handle. It is returned by read/write/erase routines when the request could not be taken.

Remarks

None.

DRV_IPF_INDEX_0 Macro

SPI Flash driver index definitions

File

drv_ipf.h

C

```
#define DRV_IPF_INDEX_0 0
```

Description

Driver SPI Flash Module Index reference

These constants provide IPF SPI Flash driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_IPF_Initialize and DRV_IPF_Open routines to identify the driver instance in use.

_DRV_IPF_CONFIG_TEMPLATE_H Macro

File

drv_ipf_config_template.h

C

```
#define _DRV_IPF_CONFIG_TEMPLATE_H
```

Description

This is macro _DRV_IPF_CONFIG_TEMPLATE_H.

DRV_IPF_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to the hardware instance.

File

drv_ipf_config_template.h

C

```
#define DRV_IPF_CLIENTS_NUMBER 4
```

Description

IPF Client Count Configuration

Sets up the maximum number of clients that can be connected to the hardware instance. So if IPF will be accessed by 2 clients then this number should be 2. It is recommended that this be set exactly equal to the number of expected clients. Client support consumes RAM memory space.

Remarks

None.

DRV_IPF_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported

File

drv_ipf_config_template.h

C

```
#define DRV_IPF_INSTANCES_NUMBER 1
```

Description

IPF driver objects configuration

Sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of IPF modules that are needed by the application. Hardware Instance support consumes RAM memory space.

Remarks

As PIC32WK has only 1 instance of IPF, this macro is always set to 1.

DRV_IPF_MODE Macro

Determines whether the driver is implemented as static or dynamic

File

drv_ipf_config_template.h

C

```
#define DRV_IPF_MODE DYNAMIC
```

Description

IPF mode

Determines whether the driver is implemented as static or dynamic. Static drivers control the peripheral directly with peripheral library routines.

Remarks

None.

Files

Files

Name	Description
drv_ipf.h	SPI Flash Driver Interface Definition
drv_ipf_config_template.h	IPF Driver Configuration Template.

Description

This section lists the source and header files used by the SPI PIC32WK IPF Flash Driver Library.

drv_ipf.h

SPI Flash Driver Interface Definition

Enumerations

	Name	Description
	DRV_IPF_BLOCK_EVENT	Identifies the possible events that can result from a request.
	DRV_IPF_BLOCK_OPERATION	Lists the different operations that IPF driver can do.
	DRV_IPF_CLIENT_STATUS	Defines the client status. **Implementation:** Dynamic
	DRV_IPF_COMMAND_STATUS	Specifies the status of the command for the read, write and erase operations.
	DRV_IPF_PROT_MODE	Lists the different memory protection modes.

Functions

	Name	Description
	DRV_IPF_BlockErase	Erase the specified number of blocks in Flash memory. **Implementation:** Dynamic
	DRV_IPF_BlockEventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed. **Implementation:** Dynamic
	DRV_IPF_BlockRead	Reads blocks of data starting from the specified address in Flash memory. **Implementation:** Dynamic
	DRV_IPF_BlockWrite	Write blocks of data starting from a specified address in Flash memory. **Implementation:** Dynamic
	DRV_IPF_ClientStatus	Gets current client-specific status of the SPI Flash driver. **Implementation:** Dynamic
	DRV_IPF_Close	Closes an opened-instance of the SPI Flash driver. **Implementation:** Dynamic
	DRV_IPF_Deinitialize	Deinitializes the specified instance of the SPI Flash driver module. **Implementation:** Dynamic
	DRV_IPF_GeometryGet	Returns the geometry of the device. **Implementation:** Dynamic
	DRV_IPF_HoldAssert	Asserts the Hold pin for flash. **Implementation:** Dynamic
	DRV_IPF_HoldDeAssert	Deasserts the Hold pin for flash. **Implementation:** Dynamic
	DRV_IPF_Initialize	Initializes the IPF SPI Flash Driver instance for the specified driver index. **Implementation:** Dynamic
	DRV_IPF_MediaIsAttached	Returns the status of the media. **Implementation:** Dynamic
	DRV_IPF_Open	Opens the specified SPI Flash driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_IPF_ProtectMemoryVolatile	Protects the memory block to which the given memory address belongs **Implementation:** Dynamic

	DRV_IPF_ReadBlockProtectionStatus	Reads the content of Block Protection Register which belongs to In-Package flash. **Implementation:** Dynamic
	DRV_IPF_Status	Gets the current status of the SPI Flash Driver module. **Implementation:** Dynamic
	DRV_IPF_Tasks	Maintains the driver's read, erase, and write state machine and implements its ISR. **Implementation:** Dynamic
	DRV_IPF_UnProtectMemoryVolatile	Un-protects the memory block to which the given memory address belongs **Implementation:** Dynamic
	DRV_IPF_WPAssert	Asserts the WP pin for flash. **Implementation:** Dynamic
	DRV_IPF_WPDeAssert	Deasserts the WP pin for flash. **Implementation:** Dynamic

Macros

	Name	Description
	_DRV_IPF_H	This is macro _DRV_IPF_H.
	DRV_IPF_BLOCK_COMMAND_HANDLE_INVALID	This value defines the SPI Flash Driver Block Command Invalid handle.
	DRV_IPF_INDEX_0	SPI Flash driver index definitions

Structures

	Name	Description
	DRV_IPF_INIT	Contains all the data necessary to initialize the SPI Flash device. **Implementation:** Dynamic

Types

	Name	Description
	DRV_IPF_BLOCK_COMMAND_HANDLE	Handle identifying block commands of the driver.
	DRV_IPF_EVENT_HANDLER	Pointer to a IPF SPI Flash Driver Event handler function. **Implementation:** Dynamic

Description

SPI Flash Driver Interface Definition

The SPI Flash device driver provides a simple interface to manage the SPI Flash modules which are external to Microchip Controllers. This file defines the interface definition for the SPI Flash Driver.

File Name

drv_IPF.h

Company

Microchip Technology Inc.

drv_ipf_config_template.h

IPF Driver Configuration Template.

Macros

	Name	Description
	_DRV_IPF_CONFIG_TEMPLATE_H	This is macro _DRV_IPF_CONFIG_TEMPLATE_H.
	DRV_IPF_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to the hardware instance.
	DRV_IPF_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported
	DRV_IPF_MODE	Determines whether the driver is implemented as static or dynamic

Description

IPF Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_ipf_config_template.h

Company

Microchip Technology Inc.

SQI Driver Library

Introduction

This library provides an interface to manage the Serial Quad Interface (SQI) module on the Microchip family of microcontrollers in different modes of operation.

Description

The MPLAB Harmony Serial Quad Interface (SQI) Driver provides a high-level interface to the SQI peripherals on Microchip's PIC32 microcontrollers. The SQI Driver includes the following features:

- Provides application ready routines to read and write data to an SQI peripheral
- Supports Single, Dual, and Quad Lane modes
- Supports Single Data Rate (SDR)
- Supports Interrupt mode operation only
- Supports multi-client operation
- Provides data transfer events
- Supports non-blocking mode operation only
- Features thread-safe functions for use in RTOS applications
- Uses the SQI module's internal DMA Controller for transfers

Using the Library

This topic describes the basic architecture of the SQI Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_sqi.h`

The interface to the SQI Driver Library is defined in the `drv_sqi.h` header file. Any C language source (`.c`) file that uses the SQI Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the Driver interacts with the framework.

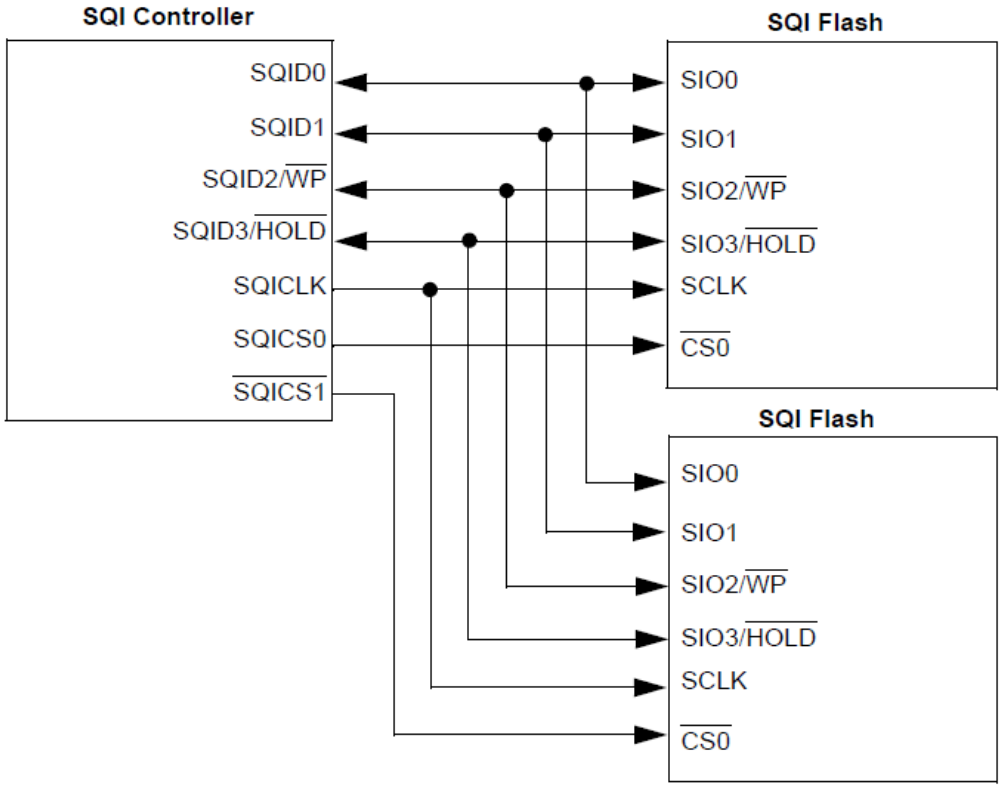
Abstraction Model

This library provides a low-level abstraction of the SQI Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The SQI Driver Library supports up to two Chip Select lines. The following diagram shows the high SQI Driver and the SQI Flash sub-system

SQI Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the SQI module.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	Provides open, close, status and other setup functions.
Data Transfer Functions	Provides data transfer functions available in the configuration.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality

Note: Not all modes are available on all devices, please refer to the specific device data sheet to determine the modes that are supported for your device.

System Functions

Provides information on the system functions provided in the SQI Driver Library.

Description

SQI Driver Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization, each instance of the SQI would be initialized with the following configuration settings (either passed dynamically at run time using DRV_SQI_INIT or by using initialization overrides) that are supported by the specific SQI Controller hardware:

- SQI Peripheral ID - Identifies the SQI Peripheral ID to be used
- Interrupt Source - The interrupt source associated with the SQI Controller
- Enabled Devices - Number of devices to enable
- Device Configuration - This configuration is per enabled device. A maximum of two devices are supported. The following configurations are allowed per device:
  - Clock Divider Value - Clock divider value to be used
  - SPI Mode of Operation - SPI mode of operation to be used for this device
  - LSB First - Send or receive least significant bit of a byte first

The DRV_SQI_Initialize function configures and initializes the SQI controller using the configuration information provided. It returns an object handle of the type SYS_MODULE_OBJ. This object handle would be used by other system interfaces such as DRV_SQI_Status, DRV_SQI_Tasks and DRV_SQI_Deinitialize.

Example:

```
/* SQI Driver Initialization Data */
const DRV_SQI_INIT drvSqiInit =
{
    .sqiId = SQI_ID_0,
    .interruptSource = INT_SOURCE_SQI1,
    .enabledDevices = DRV_SQI_ENABLE_DEVICE_1,
    .clockDivider = DRV_SQI_CLK_DIV_1,
    .devCfg[0].spiMode = DRV_SQI_SPI_MODE_0,
    .devCfg[0].lsbFirst = false,
};

/* Initialize the SQI Driver */
sysObj.drvSqi = DRV_SQI_Initialize(DRV_SQI_INDEX_0, (SYS_MODULE_INIT *)&drvSqiInit);
```

SQI Driver Task Routine

The SQI driver data transfers are interrupt driven. The data transfer request from the client results in the SQI driver kick starting the transfer if there is no transfer in progress otherwise the request is added to the driver queue. The SQI interrupt handler is responsible for invoking the DRV_SQI_Tasks, which maintains the driver state machine. The task routine checks if the current request is complete and if there is another data transfer request queued, then it kick starts the processing of the request.

SQI Driver Status

DRV_SQI_Status returns the current status of the SQI driver module and is called by the Harmony System. The application may not find the need to call this function directly.

Example:

```
SYS_MODULE_OBJ object;
// Returned from DRV_SQI_Initialize
SYS_STATUS sqiStatus;

sqiStatus = DRV_SQI_Status(object);
if (SYS_STATUS_ERROR >= sqiStatus)
{
    // Handle error
}
```

Client Core Functions

Provides information on the client core functions provided in the SQI Driver Library

Description

Opening the Driver

For the application to start using an instance of the module, it must call the DRV_SQI_Open function repeatedly until a valid handle is returned by the driver. This provides the configuration required to open the SQI instance for operation.

For the various options available for I/O INTENT please refer to "Data Types and Constants" in the Library Interface section.

Example:

```
sqiHandle = DRV_SQI_Open (0, DRV_IO_INTENT_READWRITE);
if (sqiHandle != DRV_HANDLE_INVALID)
{
    /* Do further processing. */
}
else
{
```

```
    /* Call until the function returns a valid handle. */
}
```

Closing the Driver

Closes an opened-instance of the SQI driver

Example:

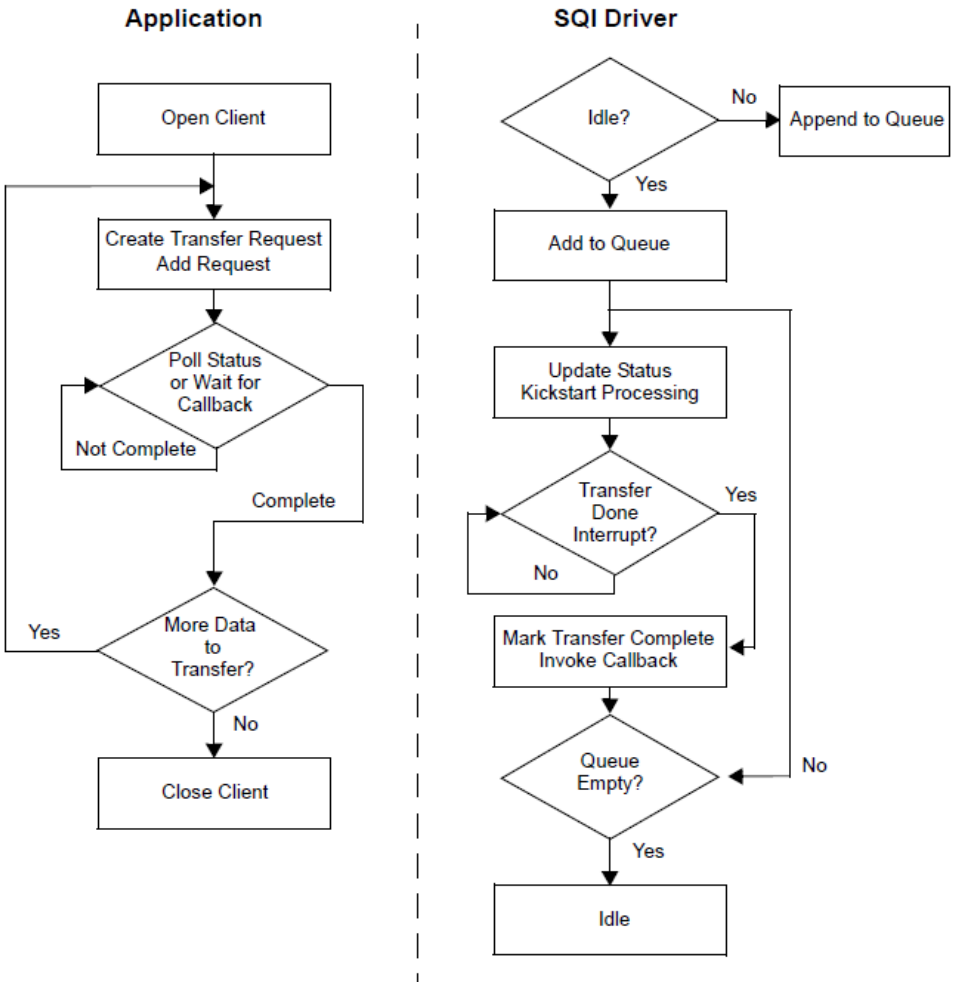
```
DRV_HANDLE handle; // Returned from DRV_SQI_Open
DRV_SQI_Close(handle);
```

Client Data Transfer Functions

Provides information on the client data transfer functions provided in the SQI Driver Library.

Description

Client data transfer functionality provides API interfaces for the data transfer operation. The following diagram illustrates the data transfer model.



Applications need to perform the following steps to transfer data using the SQI Driver:

1. The system should have completed necessary initialization and the DRV_SQI_Tasks should be running in an interrupt environment.
2. Open_the driver using DRV_SQI_Open with the necessary intent. The application should wait call the DRV_SQI_Open until the function returns a valid open handle.
3. Register callback function using the DRV_SQI_EventHandlerSet.
4. Add a transfer request using the buffer using the DRV_SQI_TransferData function. The reads or writes of blocks of data generally involves sending down the read or a write command, the address on the device from/to which data is to be read/written. The client also has to specify the source or destination buffer and the number of bytes to be read or written. The client builds an array of transfer elements containing this information and passes the array and the number of elements of the array as part of this transfer request.
5. Check for the current transfer status using DRV_SQI_CommandStatus until the transfer progress is DRV_SQI_COMMAND_COMPLETED, or wait for the callback to be called.
6. When the client has no more data to be transferred, the client can close the driver using DRV_SQI_Close.

Example:

```
#define READ_BUF_SIZE 512

uint8_t readBuffer[READ_BUF_SIZE] __attribute__((coherent, aligned(16)));
uint8_t command [5] __attribute__((coherent, aligned(16)) = {0x0B, 0x00, 0x00, 0x00, 0x0FF};
uint8_t numElements = 0;
DRV_SQI_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;
DRV_SQI_TRANSFER_ELEMENT xferData[2];

// mySQIHandle is the handle returned by the DRV_SQI_Open function.
// Setup the transfer elements.

xferData[0].data = &command[0];
xferData[0].length = sizeof(command);
xferData[0].flag = (DRV_SQI_FLAG_MODE_SINGLE_LANE);

xferData[1].data = readBuffer;
xferData[1].length = READ_BUF_SIZE;
xferData[1].flag = (DRV_SQI_FLAG_MODE_QUAD_LANE | DRV_SQI_FLAG_DIR_READ | DRV_SQI_FLAG_DEASSERT_CS);

DRV_SQI_TransferData(mySQIHandle, &commandHandle, 0, xferData, 2);

if(DRV_SQI_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}
else
{
    // Transfer operation queued successfully. Wait for the
    // completion event.
}

// Transfer completion can be tracked either by polling on the commandHandle or waiting
// for the event using the event callback function.
status = DRV_SQI_CommandStatus(mySQIHandle, commandHandle);
if(status == DRV_SQI_COMMAND_COMPLETED)
{
    // Operation Done
}

// Event handler.
void APP_SQIEventHandler
(
    DRV_SQI_EVENT event,
    DRV_SQI_COMMAND_HANDLE handle,
    uintptr_t context
)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event
    // handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_SQI_EVENT_COMMAND_COMPLETE:
            // This means the operation was completed
            // successfully.
            break;

        case DRV_SQI_EVENT_COMMAND_ERROR:
            // Operation failed. Handle the error.
            break;

        default:
            break;
    }
}
```

Configuring the Library

Macros

	Name	Description
	DRV_SQI_BUFFER_OBJECT_NUMBER	Selects the maximum number of buffer objects
	DRV_SQI_CLIENTS_NUMBER	Selects the maximum number of clients
	DRV_SQI_DMA_BUFFER_DESCRIPTORS_NUMBER	Selects the maximum number of DMA Buffer descriptors to be used by the driver.
	DRV_SQI_INSTANCES_NUMBER	Selects the maximum number of Driver instances that can be supported by the dynamic driver.
	DRV_SQI_INTERRUPT_MODE	Macro specifies operation of the driver to be in the interrupt mode or polled mode

Description

The configuration of the SQI driver is based on the file `system_config.h`.

This header file contains the configuration selection for the SQI driver. Based on the selections made, the SQI driver may support the selected features. These configuration settings will apply to all instances of the SQI driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_SQI_BUFFER_OBJECT_NUMBER Macro

Selects the maximum number of buffer objects

File

drv_sqi_config_template.h

C

```
#define DRV_SQI_BUFFER_OBJECT_NUMBER 5
```

Description

SQI Driver maximum number of buffer objects

This definition selects the maximum number of buffer objects. This indirectly also specifies the queue depth. The SQI Driver can queue up DRV_SQI_BUFFER_OBJECT_NUMBER of read/write/erase requests before return a DRV_SQI_BUFFER_HANDLE_INVALID due to the queue being full. Buffer objects are shared by all instances of the driver. Increasing this number increases the RAM requirement of the driver.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_SQI_CLIENTS_NUMBER Macro

Selects the maximum number of clients

File

drv_sqi_config_template.h

C

```
#define DRV_SQI_CLIENTS_NUMBER 1
```

Description

SQI maximum number of clients

This definition selects the maximum number of clients that the SQI driver can supported at run time. This constant defines the total number of SQI driver clients that will be available to all instances of the SQI driver.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_SQI_DMA_BUFFER_DESCRIPTORS_NUMBER Macro

Selects the maximum number of DMA Buffer descriptors to be used by the driver.

File

drv_sqi_config_template.h

C

```
#define DRV_SQI_DMA_BUFFER_DESCRIPTORS_NUMBER 4
```

Description

SQI Driver maximum number DMA Buffer Descriptors

This definition selects the maximum number of DMA buffer descriptor objects. The SQI Driver can queue up to DRV_SQI_DMA_BUFFER_DESCRIPTORS_NUMBER of transactions to be processed by the hardware. DMA buffer desired are shared by all instances of the driver. Increasing this number increases the RAM requirement of the driver.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_SQI_INSTANCES_NUMBER Macro

Selects the maximum number of Driver instances that can be supported by the dynamic driver.

File

drv_sqi_config_template.h

C

```
#define DRV_SQI_INSTANCES_NUMBER 1
```

Description

SQI Driver instance configuration

This definition selects the maximum number of Driver instances that can be supported by the dynamic driver. In case of this driver, multiple instances of the driver could use the same hardware instance.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_SQI_INTERRUPT_MODE Macro

Macro specifies operation of the driver to be in the interrupt mode or polled mode

File

drv_sqi_config_template.h

C

```
#define DRV_SQI_INTERRUPT_MODE true
```

Description

SQI interrupt and polled mode operation control

This macro specifies operation of the driver to be in the interrupt mode or polled mode

- true - Select if interrupt mode of SQI operation is desired
- false - Select if polling mode of SQI operation is desired

Not defining this option to true or false will result in build error.

Remarks

This macro is mandatory when building the driver for dynamic operation.

Building the Library

This section lists the files that are available in the SQI Driver Library.

Description

This section list the files that are available in the \src folder of the SQI Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/sqi`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_sqi.h`	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_sqi.c`	This file contains the source code for the dynamic implementation of the SQI driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The SQI Driver Library depends on the following modules:

- Clock System Service Library

Optional Dependencies

- Interrupt System Service Library (used when task is running in Interrupt mode)

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_SQI_Initialize	Initializes the SQI instance for the specified driver index
	DRV_SQI_Deinitialize	Deinitializes the specified instance of the SQI driver module
	DRV_SQI_Status	Gets the current status of the SQI driver module.
	DRV_SQI_Tasks	Maintains the driver's task state machine.

b) Client Setup Functions

	Name	Description
	DRV_SQI_Open	Opens the specified SQI driver instance and returns a handle to it.
	DRV_SQI_Close	Closes an opened-instance of the SQI driver
	DRV_SQI_CommandStatus	Gets the current status of the transfer request.
	DRV_SQI_EventHandlerSet	Allows a client to register an event handling function, which the driver can invoke when the queued transfer request has completed.

c) Data Transfer Functions

	Name	Description
	DRV_SQI_TransferData	Queue a data transfer operation on the specified SQI device.

	DRV_SQI_TransferFrames	Queue a transfer request operation on the SQI device.

d) Data Types and Constants

	Name	Description
	DRV_SQI_COMMAND_HANDLE	Handle to identify the transfer request queued at the SQI driver.
	DRV_SQI_COMMAND_STATUS	Specifies the status of the transfer request.
	DRV_SQI_EVENT	Identifies the possible events that can result from a transfer request.
	DRV_SQI_EVENT_HANDLER	Pointer to a SQI Driver Event handler function
	DRV_SQI_SPI_OPERATION_MODE	Enumeration of the SPI mode of operation supported by the SQI Controller.
	DRV_SQI_TRANSFER_FLAGS	Enumeration of the configuration options associated with a single transfer element.
	DRV_SQI_TransferElement	Defines the data transfer element of the SQI driver.
	DRV_SQI_COMMAND_HANDLE_INVALID	Identifies an invalid command handle.
	DRV_SQI_INDEX_0	SQI driver index definitions.
	DRV_SQI_FLAG_32_BIT_ADDR_ENABLE	This is macro DRV_SQI_FLAG_32_BIT_ADDR_ENABLE.
	DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_MASK	This is macro DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_MASK.
	DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_POS	Enables 32-bit addressing instead of 24-bit addressing.
	DRV_SQI_FLAG_ADDR_ENABLE	This is macro DRV_SQI_FLAG_ADDR_ENABLE.
	DRV_SQI_FLAG_ADDR_ENABLE_MASK	This is macro DRV_SQI_FLAG_ADDR_ENABLE_MASK.
	DRV_SQI_FLAG_ADDR_ENABLE_POS	Address Enable Macro.
	DRV_SQI_FLAG_CRM_ENABLE	This is macro DRV_SQI_FLAG_CRM_ENABLE.
	DRV_SQI_FLAG_CRM_ENABLE_MASK	This is macro DRV_SQI_FLAG_CRM_ENABLE_MASK.
	DRV_SQI_FLAG_CRM_ENABLE_POS	Continuous Read Mode Enable Macro.
	DRV_SQI_FLAG_DATA_DIRECTION_MASK	This is macro DRV_SQI_FLAG_DATA_DIRECTION_MASK.
	DRV_SQI_FLAG_DATA_DIRECTION_POS	Macros to select the direction of the transfers.
	DRV_SQI_FLAG_DATA_DIRECTION_READ	This is macro DRV_SQI_FLAG_DATA_DIRECTION_READ.
	DRV_SQI_FLAG_DATA_DIRECTION_WRITE	This is macro DRV_SQI_FLAG_DATA_DIRECTION_WRITE.
	DRV_SQI_FLAG_DATA_ENABLE	This is macro DRV_SQI_FLAG_DATA_ENABLE.
	DRV_SQI_FLAG_DATA_ENABLE_MASK	This is macro DRV_SQI_FLAG_DATA_ENABLE_MASK.
	DRV_SQI_FLAG_DATA_ENABLE_POS	Data Enable Macro.
	DRV_SQI_FLAG_DATA_TARGET_MASK	This is macro DRV_SQI_FLAG_DATA_TARGET_MASK.
	DRV_SQI_FLAG_DATA_TARGET_MEMORY	This is macro DRV_SQI_FLAG_DATA_TARGET_MEMORY.
	DRV_SQI_FLAG_DATA_TARGET_POS	Macros to select the source and destination of a transfer.
	DRV_SQI_FLAG_DATA_TARGET_REGISTER	This is macro DRV_SQI_FLAG_DATA_TARGET_REGISTER.
	DRV_SQI_FLAG_DDR_ENABLE	This is macro DRV_SQI_FLAG_DDR_ENABLE.
	DRV_SQI_FLAG_DDR_ENABLE_MASK	This is macro DRV_SQI_FLAG_DDR_ENABLE_MASK.
	DRV_SQI_FLAG_DDR_ENABLE_POS	DDR Enable Macro.
	DRV_SQI_FLAG_INSTR_ENABLE	This is macro DRV_SQI_FLAG_INSTR_ENABLE.
	DRV_SQI_FLAG_INSTR_ENABLE_MASK	This is macro DRV_SQI_FLAG_INSTR_ENABLE_MASK.
	DRV_SQI_FLAG_INSTR_ENABLE_POS	Macros listing the bitmap values for the flags member of the DRV_SQI_TransferFrame structure. Instruction Enable Macro.
	DRV_SQI_FLAG_OPT_ENABLE	This is macro DRV_SQI_FLAG_OPT_ENABLE.
	DRV_SQI_FLAG_OPT_ENABLE_MASK	This is macro DRV_SQI_FLAG_OPT_ENABLE_MASK.
	DRV_SQI_FLAG_OPT_ENABLE_POS	Option Enable Macro.
	DRV_SQI_FLAG_OPT_LENGTH	This is macro DRV_SQI_FLAG_OPT_LENGTH.
	DRV_SQI_FLAG_OPT_LENGTH_1BIT	This is macro DRV_SQI_FLAG_OPT_LENGTH_1BIT.
	DRV_SQI_FLAG_OPT_LENGTH_2BIT	This is macro DRV_SQI_FLAG_OPT_LENGTH_2BIT.
	DRV_SQI_FLAG_OPT_LENGTH_4BIT	This is macro DRV_SQI_FLAG_OPT_LENGTH_4BIT.
	DRV_SQI_FLAG_OPT_LENGTH_8BIT	This is macro DRV_SQI_FLAG_OPT_LENGTH_8BIT.
	DRV_SQI_FLAG_OPT_LENGTH_MASK	This is macro DRV_SQI_FLAG_OPT_LENGTH_MASK.
	DRV_SQI_FLAG_OPT_LENGTH_POS	Macros to enable and specify the option length.
	DRV_SQI_FLAG_SQI_CS_NUMBER	This is macro DRV_SQI_FLAG_SQI_CS_NUMBER.
	DRV_SQI_FLAG_SQI_CS_NUMBER_0	This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_0.
	DRV_SQI_FLAG_SQI_CS_NUMBER_1	This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_1.
	DRV_SQI_FLAG_SQI_CS_NUMBER_2	This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_2.

	DRV_SQI_FLAG_SQI_CS_NUMBER_3	This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_3.
	DRV_SQI_FLAG_SQI_CS_NUMBER_MASK	This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_MASK.
	DRV_SQI_FLAG_SQI_CS_NUMBER_POS	Macros to select the SQI CS Line Number to be used for the current transfer • frame.
	DRV_SQI_LANE_CONFIG	Defines the SQI lane configuration options.
	DRV_SQI_TransferFrame	Defines the transfer frame of the SQI driver.

Description

This section describes the API functions of the SQI Driver Library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_SQI_Initialize Function

Initializes the SQI instance for the specified driver index

File

drv_sqi.h

C

```
SYS_MODULE_OBJ DRV_SQI_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT *const init);
```

Returns

Returns a valid handle to a driver instance object on success. Otherwise returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the SQI driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This routine must be called before any other SQI routines are called.

This routine should only be called once during system initialization unless DRV_SQI_Deinitialize is called to deinitialize the driver instance.

This routine will NEVER block for hardware access. If the operation requires time to allow the hardware to initialize, it will be reported by the DRV_SQI_Status operation. The system must use DRV_SQI_Status to find out when the driver is in the ready state.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the SQI Driver.

SYS_MODULE_OBJ  objectHandle;
TODO:Replace with appropriate init snippet.
    // SQI Driver Initialization Data
    const DRV_SQI_INIT drvSqiInit =
    {
        .sqiId = SQI_ID_0,
        .interruptSource = INT_SOURCE_SQI1,
        .enabledDevices = DRV_SQI_ENABLE_BOTH_DEVICES,
        .clockDivider = DRV_SQI_CLK_DIV_1,
        .devCfg[0].spiMode = DRV_SQI_SPI_MODE_0,
        .devCfg[0].lsbFirst = true,
        .devCfg[1].spiMode = DRV_SQI_SPI_MODE_3,
        .devCfg[1].lsbFirst = false,
    };

    objectHandle = DRV_SQI_Initialize(DRV_SQI_INDEX_0, (SYS_MODULE_INIT*)&drvSqiInit);
    if (SYS_MODULE_OBJ_INVALID == objectHandle)
    {
        // Handle error
    }
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized.
init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_SQI_Initialize
(
const SYS_MODULE_INDEX index,
const SYS_MODULE_INIT * const init
);

DRV_SQI_Deinitialize Function

Deinitializes the specified instance of the SQI driver module

File

drv_sqi.h

C

```
void DRV_SQI_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SQI driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

Preconditions

Function DRV_SQI_Initialize should have been called before calling this function.

Parameter: object - Driver object handle, returned from the DRV_SQI_Initialize routine

Example

```
// This code snippet shows an example of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_SQI_Initialize
SYS_STATUS          status;

DRV_SQI_Deinitialize(object);

status = DRV_SQI_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later to know if the driver is deinitialized.
}
```

Function

void DRV_SQI_Deinitialize
(
SYS_MODULE_OBJ object
);

DRV_SQI_Status Function

Gets the current status of the SQI driver module.

File

drv_sqi.h

C

```
SYS_STATUS DRV_SQI_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and can accept transfer requests.

SYS_STATUS_UNINITIALIZED - Indicates the driver is not initialized.

Description

This routine provides the current status of the SQI driver module.

Remarks

This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_SQI_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SQI_Initialize
SYS_STATUS          sqiStatus;

sqiStatus = DRV_SQI_Status(object);
else if (SYS_STATUS_ERROR >= sqiStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SQI_Initialize routine

Function

SYS_STATUS DRV_SQI_Status

(

SYS_MODULE_OBJ object

);

DRV_SQI_Tasks Function

Maintains the driver's task state machine.

File

drv_sqi.h

C

```
void DRV_SQI_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal task state machine.

Remarks

This routine may either be called by the system's task routine(SYS_Tasks) or the from the interrupt service routine of the peripheral.

Preconditions

The DRV_SQI_Initialize routine must have been called for the specified SQI driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SQI_Initialize

while (true)
{
    DRV_SQI_Tasks (object);
    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_SQI_Initialize)

Function

void DRV_SQI_Tasks

(

SYS_MODULE_OBJ object

);

b) Client Setup Functions

DRV_SQI_Open Function

Opens the specified SQI driver instance and returns a handle to it.

File

drv_sqi.h

C

```
DRV_HANDLE DRV_SQI_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, DRV_HANDLE_INVALID is returned. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_SQI_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver hardware instance being opened is not initialized or is invalid

Description

This routine opens the specified SQI driver instance and provides a handle identifying the SQI driver instance. This handle must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_SQI_Close routine is called. This routine will NEVER block waiting for hardware. If the driver has has already been opened, it cannot be opened exclusively.

Preconditions

Function DRV_SQI_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_SQI_Open(DRV_SQI_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened.
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver.

Function

```
  DRV_HANDLE DRV_SQI_Open
(
const SYS_MODULE_INDEX index,
const    DRV_IO_INTENT ioIntent
);
```

DRV_SQI_Close Function

Closes an opened-instance of the SQI driver

File

drv_sqi.h

C

```
void DRV_SQI_Close(const DRV_HANDLE handle);
```

Returns

None

Description

This routine closes an opened-instance of the SQI driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_SQI_Open before the caller may use the driver again. Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_SQI_Initialize routine must have been called for the specified SQI driver instance.

DRV_SQI_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_SQI_Open

DRV_SQI_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

```
void DRV_SQI_Close
(
const    DRV_HANDLE handle
);
```

DRV_SQI_CommandStatus Function

Gets the current status of the transfer request.

File

drv_sqi.h

C

```
DRV_SQI_COMMAND_STATUS DRV_SQI_CommandStatus(const DRV_HANDLE handle, const DRV_SQI_COMMAND_HANDLE
commandHandle);
```

Returns

A DRV_SQI_COMMAND_STATUS value describing the current status of the transfer request. Returns DRV_SQI_COMMAND_ERROR_UNKNOWN if the client handle or the handle is not valid.

Description

This routine gets the current status of the tranfer request. The application must use this routine where the status of a scheduled transfer request needs to polled on. The function may return DRV_SQI_COMMAND_COMPLETED in a case where the handle has expired. A handle expires when the internal buffer object is re-assigned to another transfer request. It is recommended that this function be called regularly in order to track the status of the transfer request correctly.

The application can alternatively register an event handler to receive the transfer completion events.

Remarks

This routine will not block for hardware access and will immediately return the current status of the transfer request.

Preconditions

The DRV_SQI_Initialize() routine must have been called.

The DRV_SQI_Open() must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE                  handle;         // Returned from DRV_SQI_Open
DRV_SQI_COMMAND_HANDLE      commandHandle;
DRV_SQI_COMMAND_STATUS      status;

status = DRV_SQI_CommandStatus(handle, commandHandle);
if(status == DRV_SQI_COMMAND_COMPLETED)
{
    // Operation Done
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_SQI_COMMAND_STATUS DRV_SQI_CommandStatus

(

const DRV_HANDLE handle,

const DRV_SQI_COMMAND_HANDLE commandHandle

);

DRV_SQI_EventHandlerSet Function

Allows a client to register an event handling function, which the driver can invoke when the queued transfer request has completed.

File

drv_sqi.h

C

```
void DRV_SQI_EventHandlerSet(const DRV_HANDLE handle, const void * eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client queues a transfer request with the driver, it is provided with a handle identifying the transfer request that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any transfer operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_SQI_Initialize() routine must have been called for the specified SQI driver instance.

The DRV_SQI_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

DRV_SQI_TransferFrame xferFrame;
DRV_SQI_COMMAND_HANDLE commandHandle;

// drvSQIHandle is the handle returned by the DRV_SQI_Open function.
// Client registers an event handler with driver. This is done once.
DRV_SQI_EventHandlerSet(drvSQIHandle, APP_SQIEventHandler, (uintptr_t)&myAppObj);

DRV_SQI_Read(drvSQIHandle, &commandHandle, &xferFrame, 1);

if(DRV_SQI_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event handler.
void APP_SQIEventHandler
(
    DRV_SQI_EVENT event,
    DRV_SQI_COMMAND_HANDLE handle,
    uintptr_t context
)
{
    // The context handle was set to an application specific object. It is
    // now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_SQI_EVENT_COMMAND_COMPLETE:
            // This means the operation was completed successfully.
            break;

        case DRV_SQI_EVENT_COMMAND_ERROR:
            // Operation failed. Handle the error.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user

context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_SQI_EventHandlerSet
(
const DRV_HANDLE handle,
const void *eventHandler,
const uintptr_t context
);

c) Data Transfer Functions

DRV_SQI_TransferData Function

Queue a data transfer operation on the specified SQI device.

File

drv_sqi.h

C

```
void DRV_SQI_TransferData(DRV_HANDLE handle, DRV_SQI_COMMAND_HANDLE * commandHandle, uint8_t sqiDevice,
DRV_SQI_TransferElement * xferData, uint8_t numElements);
```

Returns

The handle to the command request is returned in the commandHandle argument. It will be DRV_SQI_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This routine queues a data transfer operation on the specified SQI device. The reads or writes of blocks of data generally involves sending down the read or a write command, the address on the device from/to which data is to be read/written. The client also has to specify the source or destination buffer and the number of bytes to be read or written. The client builds an array of transfer elements containing these information and passes the array and the number of elements of the array as part of this transfer operation. If an event handler is registered with the driver the event handler would be invoked with the status of the operation once the operation has been completed. The function returns DRV_SQI_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the driver handle is invalid
- if the transfer element is NULL or number of transfer elements is zero
- if a buffer object could not be allocated to the request

Remarks

None.

Preconditions

The DRV_SQI_Initialize routine must have been called for the specified SQI driver instance.

DRV_SQI_Open must have been called with DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE as the ioIntent to obtain a valid opened device handle.

Example

```
#define READ_BUF_SIZE 512

uint8_t readBuffer[READ_BUF_SIZE] __attribute__((coherent, aligned(16)));
uint8_t command [5] __attribute__((coherent, aligned(16)) = {0x0B, 0x00, 0x00, 0x00, 0x0FF};
uint8_t numElements = 0;
DRV_SQI_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;
DRV_SQI_TransferElement xferData[2];

// mySQIHandle is the handle returned by the DRV_SQI_Open function.
// Setup the transfer elements.
```

```
xferData[0].data = &command[0];
xferData[0].length = sizeof(command);
xferData[0].flag = (DRV_SQI_FLAG_MODE_SINGLE_LANE);

xferData[1].data = readBuffer;
xferData[1].length = READ_BUF_SIZE;
xferData[1].flag = (DRV_SQI_FLAG_MODE_QUAD_LANE | DRV_SQI_FLAG_DIR_READ | DRV_SQI_FLAG_DEASSERT_CS);

DRV_SQI_TransferData(mySQIHandle, &commandHandle, 0, xferData, 2);

if(DRV_SQI_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}
else
{
    // Transfer operation queued successfully. Wait for the completion event.
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
sqiDevice	The SQI device index on which the operation is to be performed.
xferData	Pointer to the transfer elements array.
numElements	Number of elements in the transfer elements array.

Function

void DRV_SQI_TransferData
(
DRV_HANDLE handle,
DRV_SQI_COMMAND_HANDLE *commandHandle,
uint8_t sqiDevice,
DRV_SQI_TransferElement *xferData,
uint8_t numElements
);

DRV_SQI_TransferFrames Function

Queue a transfer request operation on the SQI device.

File

drv_sqi.h

C

```
void DRV_SQI_TransferFrames(DRV_HANDLE handle, DRV_SQI_COMMAND_HANDLE * commandHandle,
DRV_SQI_TransferFrame * frame, uint8_t numFrames);
```

Returns

The handle to the transfer request is returned in the commandHandle argument. It will be DRV_SQI_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This routine queues a transfer request operation on the SQI device. In order to perform any operation on the sqi flash device, a one byte instruction specifying the operation to be performed needs to be sent out. This is followed by optional address from/to which data is to be read/written, option, dummy and data bytes.

If an event handler is registered with the driver the event handler would be invoked with the status of the operation once the operation has been completed. The function returns DRV_SQI_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the driver handle is invalid
- if the transfer element is NULL or number of transfer elements is zero
- if a buffer object could not be allocated to the request

Remarks

None.

Preconditions

The DRV_SQI_Initialize routine must have been called for the specified SQI driver instance.

DRV_SQI_Open must have been called with DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE as the ioIntent to obtain a valid opened device handle.

Example

```
#define READ_BUF_SIZE 512

uint8_t readBuffer[READ_BUF_SIZE];
uint8_t numElements = 0;
DRV_SQI_COMMAND_HANDLE cmdHandle;
DRV_SQI_TransferFrame xferFrame;

DRV_SQI_TransferFrame *frame = &xferFrame;
frame->instruction = 0x6B;
frame->address = 0x00;
frame->data = readBuffer;
frame->length = READ_BUF_SIZE;
frame->laneCfg = DRV_SQI_LANE_QUAD_DATA;
frame->numDummyBytes = 8;
frame->flags = (DRV_SQI_FLAG_INSTR_ENABLE_MASK | DRV_SQI_FLAG_DATA_ENABLE_MASK |
    DRV_SQI_FLAG_ADDR_ENABLE_MASK | DRV_SQI_FLAG_DATA_TARGET_MEMORY |
    DRV_SQI_FLAG_DATA_DIRECTION_READ);
DRV_SQI_TransferFrames (sqiHandle, &cmdHandle, frame, 1);
if (cmdHandle == DRV_SQI_COMMAND_HANDLE_INVALID)
{
    // handle the failure.
}
else
{
    // continue with the rest of the operation.
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the handle to the track the status of the transfer request.
frame	Pointer to the transfer frame array.
numFrames	Number of elements in the transfer frame array.

Function

void DRV_SQI_TransferFrames

(

DRV_HANDLE handle,

DRV_SQI_COMMAND_HANDLE *commandHandle,

DRV_SQI_TransferFrame *frame,

uint8_t numFrames

);

d) Data Types and Constants

DRV_SQI_COMMAND_HANDLE Type

Handle to identify the transfer request queued at the SQI driver.

File

drv_sqi.h

C

```
typedef uintptr_t DRV_SQI_COMMAND_HANDLE;
```

Description

SQI Driver Command Handle

A command handle is returned by a call to the DRV_SQI_TransferFrames () function. This handle allows the application to track the completion of the request. This command handle is also returned to the client along with the event that has occurred with respect to the request. This allows the application to connect the event to a specific transfer request in case where multiple requests are queued.

The command handle associated with the transfer request expires when the client has been notified of the completion of the request (after event handler function that notifies the client returns) or after the request has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_SQI_COMMAND_STATUS Enumeration

Specifies the status of the transfer request.

File

drv_sqi.h

C

```
typedef enum {
  DRV_SQI_COMMAND_COMPLETED,
  DRV_SQI_COMMAND_QUEUED,
  DRV_SQI_COMMAND_IN_PROGRESS,
  DRV_SQI_COMMAND_ERROR_UNKNOWN
} DRV_SQI_COMMAND_STATUS;
```

Members

Members	Description
DRV_SQI_COMMAND_COMPLETED	Command completed.
DRV_SQI_COMMAND_QUEUED	Command is pending.
DRV_SQI_COMMAND_IN_PROGRESS	Command is being processed
DRV_SQI_COMMAND_ERROR_UNKNOWN	There was an error while processing the command.

Description

SQI Driver Command Status

This enumeration identifies the possible status values associated with a transfer request. The client can retrieve the status by calling the DRV_SQI_CommandStatus () function and passing the command handle associated with the request.

Remarks

None.

DRV_SQI_EVENT Enumeration

Identifies the possible events that can result from a transfer request.

File

drv_sqi.h

C

```
typedef enum {
  DRV_SQI_EVENT_COMMAND_COMPLETE = 0,
  DRV_SQI_EVENT_COMMAND_ERROR
} DRV_SQI_EVENT;
```

Members

Members	Description
DRV_SQI_EVENT_COMMAND_COMPLETE = 0	Operation has been completed successfully.
DRV_SQI_EVENT_COMMAND_ERROR	There was an error during the operation

Description

SQI Driver Events

This enumeration identifies the possible events that can result from a transfer request issued by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_SQI_EventHandlerSet function when a request is completed.

DRV_SQI_EVENT_HANDLER Type

Pointer to a SQI Driver Event handler function

File

drv_sqi.h

C

```
typedef void (* DRV_SQI_EVENT_HANDLER)(DRV_SQI_EVENT event, DRV_SQI_COMMAND_HANDLE commandHandle, void
*context);
```

Returns

None.

Description

SQI Driver Event Handler Function Pointer data type.

This data type defines the required function signature for the SQI driver event handling callback function. A client must register a pointer to a event handling function the signature(parameter and return value types) of which should match the types specified by this function pointer in order to receive transfer request related event call backs from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_SQI_EVENT_COMMAND_COMPLETE, that the operation associated with the transfer request was completed successfully. If the event is DRV_SQI_EVENT_COMMAND_ERROR, there was an error while executing the transfer request.

The context parameter contains context details provided by the client as part of registering the event handler function. This context value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) of the client that made the request.

Example

```
void MyAppCommandEventHandler
(
    DRV_SQI_EVENT event,
    DRV_SQI_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT)context;

    switch(event)
    {
        case DRV_SQI_EVENT_COMMAND_COMPLETE:
            // Handle the completed transfer request.
            break;

        case DRV_SQI_EVENT_COMMAND_ERROR:
        default:
            // Handle the failed transfer request.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle identifying the transfer request to which this event relates

context	Value identifying the context of the application that registered the event handling function.

DRV_SQI_SPI_OPERATION_MODE Enumeration

Enumeration of the SPI mode of operation supported by the SQI Controller.

File

drv_sqi.h

C

```
typedef enum {
  DRV_SQI_SPI_MODE_0 = 0,
  DRV_SQI_SPI_MODE_3 = 3
} DRV_SQI_SPI_OPERATION_MODE;
```

Members

Members	Description
DRV_SQI_SPI_MODE_0 = 0	CPOL = 0 and CPHA = 0. SCK Idle state = LOW
DRV_SQI_SPI_MODE_3 = 3	CPOL = 1 and CPHA = 1. SCK Idle state = HIGH

Description

SQI SPI Mode of operation

This enumeration lists the SPI mode of operation supported by the SQI controller. In MODE 0 of operation: CPOL = 0 and CPHA = 0. SCK Idle state = LOW

In MODE 3 of operation: CPOL = 1 and CPHA = 1. SCK Idle state = HIGH

In both MODE 0 and MODE 3 of operation the: SQI Data Input is sampled on the rising edge of the SQI Clock SQI Data is Output on the falling edge of the SQI Clock

Remarks

None

DRV_SQI_TRANSFER_FLAGS Enumeration

Enumeration of the configuration options associated with a single transfer element.

File

drv_sqi.h

C

```
typedef enum {
  DRV_SQI_FLAG_MODE_SINGLE_LANE = 0x00,
  DRV_SQI_FLAG_MODE_DUAL_LANE = 0x01,
  DRV_SQI_FLAG_MODE_QUAD_LANE = 0x02,
  DRV_SQI_FLAG_DDR_MODE = 0x04,
  DRV_SQI_FLAG_DEASSERT_CS = 0x08,
  DRV_SQI_FLAG_DIR_READ = 0x80
} DRV_SQI_TRANSFER_FLAGS;
```

Members

Members	Description
DRV_SQI_FLAG_MODE_SINGLE_LANE = 0x00	Bits 0-1: Indicates the Lane configuration to be used.
DRV_SQI_FLAG_DDR_MODE = 0x04	Bit 2: This bit indicates if DDR or SDR mode of operation is to be used.
DRV_SQI_FLAG_DEASSERT_CS = 0x08	Bit 3: This bit indicates if CS is to be de-asserted at the end of this • transaction.
DRV_SQI_FLAG_DIR_READ = 0x80	Bit 7: This bit indicates if the operation is a read or a write.

Description

Flags associated with the SQI Driver Transfer element.

This enumeration lists the various configuration options associated with a single transfer element(Refer to the data structure DRV_SQI_TransferElement). The client can specify one or more of these as configuration parameters of a single transfer element.

Remarks

None

DRV_SQI_TransferElement Structure

Defines the data transfer element of the SQI driver.

File

drv_sqi.h

C

```
typedef struct {
  uint8_t * data;
  uint32_t length;
  uint8_t flag;
} DRV_SQI_TransferElement;
```

Members

Members	Description
uint8_t * data;	Pointer to the source or destination buffer
uint32_t length;	Length of the buffer in bytes.
uint8_t flag;	This is a bitmap used to indicate the configuration options to be used • for this transfer element. One or more values of the enumeration • DRV_SQI_TRANSFER_FLAGS can be passed as part of this flag.

Description

SQI Driver data transfer element.

This data type defines the composition of a single transfer element. A single element will consist of the pointer to the source of destination buffer, length of the data to be transferred or received and the various configuration options to be used for the element. The configuration options also indicate if data is transferred to/from the device. A client builds an array of such transfer elements and passes the array and the number of elements of the array as part of the read or write operation.

Remarks

None.

DRV_SQI_COMMAND_HANDLE_INVALID Macro

Identifies an invalid command handle.

File

drv_sqi.h

C

```
#define DRV_SQI_COMMAND_HANDLE_INVALID ((DRV_SQI_COMMAND_HANDLE)(-1))
```

Description

SQI Driver Invalid Command Handle

This is the definition of an invalid command handle. An invalid command handle is returned by DRV_SQI_TransferFrames() function if the transfer request was not queued.

Remarks

None.

DRV_SQI_INDEX_0 Macro

SQI driver index definitions.

File

drv_sqi.h

C

```
#define DRV_SQI_INDEX_0 0
```

Description

SQI Driver Module Index Numbers

This constant provides the SQI driver index definition.

Remarks

This constant should be used in place of hard-coded numeric literal.

This value should be passed into the DRV_SQI_Initialize and DRV_SQI_Open functions to identify the driver instance in use.

DRV_SQI_FLAG_32_BIT_ADDR_ENABLE Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_32_BIT_ADDR_ENABLE(value) (DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_MASK & ((value) <<
DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_POS))
```

Description

This is macro DRV_SQI_FLAG_32_BIT_ADDR_ENABLE.

DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_MASK Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_MASK (0x1U << DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_POS)
```

Description

This is macro DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_MASK.

DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_POS Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_POS (6)
```

Description

Enables 32-bit addressing instead of 24-bit addressing.

DRV_SQI_FLAG_ADDR_ENABLE Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_ADDR_ENABLE(value) (DRV_SQI_FLAG_ADDR_ENABLE_MASK & ((value) <<
DRV_SQI_FLAG_ADDR_ENABLE_POS))
```

Description

This is macro DRV_SQI_FLAG_ADDR_ENABLE.

DRV_SQI_FLAG_ADDR_ENABLE_MASK Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_ADDR_ENABLE_MASK (0x1U << DRV_SQI_FLAG_ADDR_ENABLE_POS)
```

Description

This is macro DRV_SQI_FLAG_ADDR_ENABLE_MASK.

DRV_SQI_FLAG_ADDR_ENABLE_POS Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_ADDR_ENABLE_POS (1)
```

Description

Address Enable Macro.

DRV_SQI_FLAG_CRM_ENABLE Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_CRM_ENABLE(value) (DRV_SQI_FLAG_CRM_ENABLE_MASK & ((value) <<
DRV_SQI_FLAG_CRM_ENABLE_POS))
```

Description

This is macro DRV_SQI_FLAG_CRM_ENABLE.

DRV_SQI_FLAG_CRM_ENABLE_MASK Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_CRM_ENABLE_MASK (0x1U << DRV_SQI_FLAG_CRM_ENABLE_POS)
```

Description

This is macro DRV_SQI_FLAG_CRM_ENABLE_MASK.

DRV_SQI_FLAG_CRM_ENABLE_POS Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_CRM_ENABLE_POS (5)
```

Description

Continuous Read Mode Enable Macro.

DRV_SQI_FLAG_DATA_DIRECTION_MASK Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DATA_DIRECTION_MASK (0x1U << DRV_SQI_FLAG_DATA_DIRECTION_POS)
```

Description

This is macro DRV_SQI_FLAG_DATA_DIRECTION_MASK.

DRV_SQI_FLAG_DATA_DIRECTION_POS Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DATA_DIRECTION_POS (11)
```

Description

Macros to select the direction of the transfers.

DRV_SQI_FLAG_DATA_DIRECTION_READ Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DATA_DIRECTION_READ (DRV_SQI_FLAG_DATA_DIRECTION_MASK & ((0x1U) <<
DRV_SQI_FLAG_DATA_DIRECTION_POS))
```

Description

This is macro DRV_SQI_FLAG_DATA_DIRECTION_READ.

DRV_SQI_FLAG_DATA_DIRECTION_WRITE Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DATA_DIRECTION_WRITE (DRV_SQI_FLAG_DATA_DIRECTION_MASK & ((0x0U) <<
DRV_SQI_FLAG_DATA_DIRECTION_POS))
```

Description

This is macro DRV_SQI_FLAG_DATA_DIRECTION_WRITE.

DRV_SQI_FLAG_DATA_ENABLE Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DATA_ENABLE(value) (DRV_SQI_FLAG_DATA_ENABLE_MASK & ((value) <<
DRV_SQI_FLAG_DATA_ENABLE_POS))
```

Description

This is macro DRV_SQI_FLAG_DATA_ENABLE.

DRV_SQI_FLAG_DATA_ENABLE_MASK Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DATA_ENABLE_MASK (0x1U << DRV_SQI_FLAG_DATA_ENABLE_POS)
```

Description

This is macro DRV_SQI_FLAG_DATA_ENABLE_MASK.

DRV_SQI_FLAG_DATA_ENABLE_POS Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DATA_ENABLE_POS (3)
```

Description

Data Enable Macro.

DRV_SQI_FLAG_DATA_TARGET_MASK Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DATA_TARGET_MASK (0x1U << DRV_SQI_FLAG_DATA_TARGET_POS)
```

Description

This is macro DRV_SQI_FLAG_DATA_TARGET_MASK.

DRV_SQI_FLAG_DATA_TARGET_MEMORY Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DATA_TARGET_MEMORY (DRV_SQI_FLAG_DATA_TARGET_MASK & ((0x1U) <<
DRV_SQI_FLAG_DATA_TARGET_POS))
```

Description

This is macro DRV_SQI_FLAG_DATA_TARGET_MEMORY.

DRV_SQI_FLAG_DATA_TARGET_POS Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DATA_TARGET_POS (10)
```

Description

Macros to select the source and destination of a transfer.

DRV_SQI_FLAG_DATA_TARGET_REGISTER Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DATA_TARGET_REGISTER (DRV_SQI_FLAG_DATA_TARGET_MASK & ((0x0U) <<
DRV_SQI_FLAG_DATA_TARGET_POS))
```

Description

This is macro DRV_SQI_FLAG_DATA_TARGET_REGISTER.

DRV_SQI_FLAG_DDR_ENABLE Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DDR_ENABLE(value) (DRV_SQI_FLAG_DDR_ENABLE_MASK & ((value) <<
DRV_SQI_FLAG_DDR_ENABLE_POS))
```

Description

This is macro DRV_SQI_FLAG_DDR_ENABLE.

DRV_SQI_FLAG_DDR_ENABLE_MASK Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DDR_ENABLE_MASK (0x1U << DRV_SQI_FLAG_DDR_ENABLE_POS)
```

Description

This is macro DRV_SQI_FLAG_DDR_ENABLE_MASK.

DRV_SQI_FLAG_DDR_ENABLE_POS Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_DDR_ENABLE_POS (4)
```

Description

DDR Enable Macro.

DRV_SQI_FLAG_INSTR_ENABLE Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_INSTR_ENABLE(value) (DRV_SQI_FLAG_INSTR_ENABLE_MASK & ((value) <<
DRV_SQI_FLAG_INSTR_ENABLE_POS))
```

Description

This is macro DRV_SQI_FLAG_INSTR_ENABLE.

DRV_SQI_FLAG_INSTR_ENABLE_MASK Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_INSTR_ENABLE_MASK (0x1U << DRV_SQI_FLAG_INSTR_ENABLE_POS)
```

Description

This is macro DRV_SQI_FLAG_INSTR_ENABLE_MASK.

DRV_SQI_FLAG_INSTR_ENABLE_POS Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_INSTR_ENABLE_POS (0)
```

Description

Macros listing the bitmap values for the flags member of the DRV_SQI_TransferFrame structure. Instruction Enable Macro.

DRV_SQI_FLAG_OPT_ENABLE Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_OPT_ENABLE(value) (DRV_SQI_FLAG_OPT_ENABLE_MASK & ((value) <<
DRV_SQI_FLAG_OPT_ENABLE_POS))
```

Description

This is macro DRV_SQI_FLAG_OPT_ENABLE.

DRV_SQI_FLAG_OPT_ENABLE_MASK Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_OPT_ENABLE_MASK (0x1U << DRV_SQI_FLAG_OPT_ENABLE_POS)
```

Description

This is macro DRV_SQI_FLAG_OPT_ENABLE_MASK.

DRV_SQI_FLAG_OPT_ENABLE_POS Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_OPT_ENABLE_POS (2)
```

Description

Option Enable Macro.

DRV_SQI_FLAG_OPT_LENGTH Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_OPT_LENGTH(value) (DRV_SQI_FLAG_OPT_LENGTH_MASK & ((value) <<
DRV_SQI_FLAG_OPT_LENGTH_POS))
```

Description

This is macro DRV_SQI_FLAG_OPT_LENGTH.

DRV_SQI_FLAG_OPT_LENGTH_1BIT Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_OPT_LENGTH_1BIT (0x0U)
```

Description

This is macro DRV_SQI_FLAG_OPT_LENGTH_1BIT.

DRV_SQI_FLAG_OPT_LENGTH_2BIT Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_OPT_LENGTH_2BIT (0x1U)
```

Description

This is macro DRV_SQI_FLAG_OPT_LENGTH_2BIT.

DRV_SQI_FLAG_OPT_LENGTH_4BIT Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_OPT_LENGTH_4BIT (0x2U)
```

Description

This is macro DRV_SQI_FLAG_OPT_LENGTH_4BIT.

DRV_SQI_FLAG_OPT_LENGTH_8BIT Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_OPT_LENGTH_8BIT (0x3U)
```

Description

This is macro DRV_SQI_FLAG_OPT_LENGTH_8BIT.

DRV_SQI_FLAG_OPT_LENGTH_MASK Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_OPT_LENGTH_MASK (0x3U << DRV_SQI_FLAG_OPT_LENGTH_POS)
```

Description

This is macro DRV_SQI_FLAG_OPT_LENGTH_MASK.

DRV_SQI_FLAG_OPT_LENGTH_POS Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_OPT_LENGTH_POS (8)
```

Description

Macros to enable and specify the option length.

DRV_SQI_FLAG_SQI_CS_NUMBER Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_SQI_CS_NUMBER(value) (DRV_SQI_FLAG_SQI_CS_NUMBER_MASK & ((value) <<
DRV_SQI_FLAG_SQI_CS_NUMBER_POS))
```

Description

This is macro DRV_SQI_FLAG_SQI_CS_NUMBER.

DRV_SQI_FLAG_SQI_CS_NUMBER_0 Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_SQI_CS_NUMBER_0 (0x0U)
```

Description

This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_0.

DRV_SQI_FLAG_SQI_CS_NUMBER_1 Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_SQI_CS_NUMBER_1 (0x1U)
```

Description

This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_1.

DRV_SQI_FLAG_SQI_CS_NUMBER_2 Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_SQI_CS_NUMBER_2 (0x2U)
```

Description

This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_2.

DRV_SQI_FLAG_SQI_CS_NUMBER_3 Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_SQI_CS_NUMBER_3 (0x3U)
```

Description

This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_3.

DRV_SQI_FLAG_SQI_CS_NUMBER_MASK Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_SQI_CS_NUMBER_MASK (0x3U << DRV_SQI_FLAG_SQI_CS_NUMBER_POS)
```

Description

This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_MASK.

DRV_SQI_FLAG_SQI_CS_NUMBER_POS Macro

File

drv_sqi.h

C

```
#define DRV_SQI_FLAG_SQI_CS_NUMBER_POS (16)
```

Description

Macros to select the SQI CS Line Number to be used for the current transfer

- frame.

DRV_SQI_LANE_CONFIG Enumeration

Defines the SQI lane configuration options.

File

drv_sqi.h

C

```
typedef enum {
  DRV_SQI_LANE_SINGLE = 0,
  DRV_SQI_LANE_DUAL_DATA,
  DRV_SQI_LANE_QUAD_DATA,
  DRV_SQI_LANE_DUAL_ADDR_DATA,
  DRV_SQI_LANE_QUAD_ADDR_DATA,
```

```
  DRV_SQI_LANE_DUAL_ALL,
  DRV_SQI_LANE_QUAD_ALL
} DRV_SQI_LANE_CONFIG;
```

Members

Members	Description
DRV_SQI_LANE_SINGLE = 0	Instruction opcode, Address and Data are all sent in single lane
DRV_SQI_LANE_DUAL_DATA	Instruction opcode and Address are sent in single lane, while data is • sent using dual lane.
DRV_SQI_LANE_QUAD_DATA	Instruction opcode and Address are sent in single lane, while data is • sent using quad lane.
DRV_SQI_LANE_DUAL_ADDR_DATA	Instruction opcode is sent in single lane, Address and Data are sent • using dual lane.
DRV_SQI_LANE_QUAD_ADDR_DATA	Instruction opcode is sent in single lane, Address and Data are sent • using quad lane.
DRV_SQI_LANE_DUAL_ALL	Instruction opcode, Address and Data are sent using dual lanes.
DRV_SQI_LANE_QUAD_ALL	Instruction opcode, Address and Data are sent using quad lanes.

Description

SQI lane configuration options.

This enumeration lists the various lane configuration options provided by the driver.

Remarks

None.

DRV_SQI_TransferFrame Structure

Defines the transfer frame of the SQI driver.

File

drv_sqi.h

C

```
typedef struct {
  uint8_t instruction;
  uint32_t address;
  uint8_t * data;
  uint32_t length;
  DRV_SQI_LANE_CONFIG laneCfg;
  uint8_t option;
  uint8_t numDummyBytes;
  uint32_t flags;
} DRV_SQI_TransferFrame;
```

Members

Members	Description
uint8_t instruction;	8-bit instruction opcode.
uint32_t address;	24/32-bit address.
uint8_t * data;	Pointer to the source or destination buffer
uint32_t length;	Length of the buffer in bytes.
DRV_SQI_LANE_CONFIG laneCfg;	Lane Configuration.
uint8_t option;	Option code associated with the current command.
uint8_t numDummyBytes;	Optional number of dummy bytes associated with the current command.
uint32_t flags;	This is bit-map field providing various configuration options for the • current frame.

Description

SQI Driver transfer frame.

This data type defines the composition of a single transfer frame. In order to perform any operation on the SQI flash device, a one byte instruction specifying the operation to be performed needs to be sent out. This is followed by optional address from/to which data is to be read/written, option, dummy and data bytes.

The configuration options also indicate if data is transferred to/from the device.

Remarks

None.

Files

Files

Name	Description
drv_sqi.h	SQI Driver Interface Definition
drv_sqi_config_template.h	SQI driver configuration definitions.

Description

This section lists the source and header files used by the SQI Driver Library.

drv_sqi.h

SQI Driver Interface Definition

Enumerations

	Name	Description
	DRV_SQI_COMMAND_STATUS	Specifies the status of the transfer request.
	DRV_SQI_EVENT	Identifies the possible events that can result from a transfer request.
	DRV_SQI_LANE_CONFIG	Defines the SQI lane configuration options.
	DRV_SQI_SPI_OPERATION_MODE	Enumeration of the SPI mode of operation supported by the SQI Controller.
	DRV_SQI_TRANSFER_FLAGS	Enumeration of the configuration options associated with a single transfer element.

Functions

	Name	Description
	DRV_SQI_Close	Closes an opened-instance of the SQI driver
	DRV_SQI_CommandStatus	Gets the current status of the transfer request.
	DRV_SQI_Deinitialize	Deinitializes the specified instance of the SQI driver module
	DRV_SQI_EventHandlerSet	Allows a client to register an event handling function, which the driver can invoke when the queued transfer request has completed.
	DRV_SQI_Initialize	Initializes the SQI instance for the specified driver index
	DRV_SQI_Open	Opens the specified SQI driver instance and returns a handle to it.
	DRV_SQI_Status	Gets the current status of the SQI driver module.
	DRV_SQI_Tasks	Maintains the driver's task state machine.
	DRV_SQI_TransferData	Queue a data transfer operation on the specified SQI device.
	DRV_SQI_TransferFrames	Queue a transfer request operation on the SQI device.

Macros

	Name	Description
	DRV_SQI_COMMAND_HANDLE_INVALID	Identifies an invalid command handle.
	DRV_SQI_FLAG_32_BIT_ADDR_ENABLE	This is macro DRV_SQI_FLAG_32_BIT_ADDR_ENABLE.
	DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_MASK	This is macro DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_MASK.
	DRV_SQI_FLAG_32_BIT_ADDR_ENABLE_POS	Enables 32-bit addressing instead of 24-bit addressing.
	DRV_SQI_FLAG_ADDR_ENABLE	This is macro DRV_SQI_FLAG_ADDR_ENABLE.
	DRV_SQI_FLAG_ADDR_ENABLE_MASK	This is macro DRV_SQI_FLAG_ADDR_ENABLE_MASK.
	DRV_SQI_FLAG_ADDR_ENABLE_POS	Address Enable Macro.
	DRV_SQI_FLAG_CRM_ENABLE	This is macro DRV_SQI_FLAG_CRM_ENABLE.
	DRV_SQI_FLAG_CRM_ENABLE_MASK	This is macro DRV_SQI_FLAG_CRM_ENABLE_MASK.
	DRV_SQI_FLAG_CRM_ENABLE_POS	Continuous Read Mode Enable Macro.
	DRV_SQI_FLAG_DATA_DIRECTION_MASK	This is macro DRV_SQI_FLAG_DATA_DIRECTION_MASK.
	DRV_SQI_FLAG_DATA_DIRECTION_POS	Macros to select the direction of the transfers.
	DRV_SQI_FLAG_DATA_DIRECTION_READ	This is macro DRV_SQI_FLAG_DATA_DIRECTION_READ.
	DRV_SQI_FLAG_DATA_DIRECTION_WRITE	This is macro DRV_SQI_FLAG_DATA_DIRECTION_WRITE.

	DRV_SQI_FLAG_DATA_ENABLE	This is macro DRV_SQI_FLAG_DATA_ENABLE.
	DRV_SQI_FLAG_DATA_ENABLE_MASK	This is macro DRV_SQI_FLAG_DATA_ENABLE_MASK.
	DRV_SQI_FLAG_DATA_ENABLE_POS	Data Enable Macro.
	DRV_SQI_FLAG_DATA_TARGET_MASK	This is macro DRV_SQI_FLAG_DATA_TARGET_MASK.
	DRV_SQI_FLAG_DATA_TARGET_MEMORY	This is macro DRV_SQI_FLAG_DATA_TARGET_MEMORY.
	DRV_SQI_FLAG_DATA_TARGET_POS	Macros to select the source and destination of a transfer.
	DRV_SQI_FLAG_DATA_TARGET_REGISTER	This is macro DRV_SQI_FLAG_DATA_TARGET_REGISTER.
	DRV_SQI_FLAG_DDR_ENABLE	This is macro DRV_SQI_FLAG_DDR_ENABLE.
	DRV_SQI_FLAG_DDR_ENABLE_MASK	This is macro DRV_SQI_FLAG_DDR_ENABLE_MASK.
	DRV_SQI_FLAG_DDR_ENABLE_POS	DDR Enable Macro.
	DRV_SQI_FLAG_INSTR_ENABLE	This is macro DRV_SQI_FLAG_INSTR_ENABLE.
	DRV_SQI_FLAG_INSTR_ENABLE_MASK	This is macro DRV_SQI_FLAG_INSTR_ENABLE_MASK.
	DRV_SQI_FLAG_INSTR_ENABLE_POS	Macros listing the bitmap values for the flags member of the DRV_SQI_TransferFrame structure. Instruction Enable Macro.
	DRV_SQI_FLAG_OPT_ENABLE	This is macro DRV_SQI_FLAG_OPT_ENABLE.
	DRV_SQI_FLAG_OPT_ENABLE_MASK	This is macro DRV_SQI_FLAG_OPT_ENABLE_MASK.
	DRV_SQI_FLAG_OPT_ENABLE_POS	Option Enable Macro.
	DRV_SQI_FLAG_OPT_LENGTH	This is macro DRV_SQI_FLAG_OPT_LENGTH.
	DRV_SQI_FLAG_OPT_LENGTH_1BIT	This is macro DRV_SQI_FLAG_OPT_LENGTH_1BIT.
	DRV_SQI_FLAG_OPT_LENGTH_2BIT	This is macro DRV_SQI_FLAG_OPT_LENGTH_2BIT.
	DRV_SQI_FLAG_OPT_LENGTH_4BIT	This is macro DRV_SQI_FLAG_OPT_LENGTH_4BIT.
	DRV_SQI_FLAG_OPT_LENGTH_8BIT	This is macro DRV_SQI_FLAG_OPT_LENGTH_8BIT.
	DRV_SQI_FLAG_OPT_LENGTH_MASK	This is macro DRV_SQI_FLAG_OPT_LENGTH_MASK.
	DRV_SQI_FLAG_OPT_LENGTH_POS	Macros to enable and specify the option length.
	DRV_SQI_FLAG_SQI_CS_NUMBER	This is macro DRV_SQI_FLAG_SQI_CS_NUMBER.
	DRV_SQI_FLAG_SQI_CS_NUMBER_0	This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_0.
	DRV_SQI_FLAG_SQI_CS_NUMBER_1	This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_1.
	DRV_SQI_FLAG_SQI_CS_NUMBER_2	This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_2.
	DRV_SQI_FLAG_SQI_CS_NUMBER_3	This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_3.
	DRV_SQI_FLAG_SQI_CS_NUMBER_MASK	This is macro DRV_SQI_FLAG_SQI_CS_NUMBER_MASK.
	DRV_SQI_FLAG_SQI_CS_NUMBER_POS	Macros to select the SQI CS Line Number to be used for the current transfer • frame.
	DRV_SQI_INDEX_0	SQI driver index definitions.

Structures

	Name	Description
	DRV_SQI_TransferElement	Defines the data transfer element of the SQI driver.
	DRV_SQI_TransferFrame	Defines the transfer frame of the SQI driver.

Types

	Name	Description
	DRV_SQI_COMMAND_HANDLE	Handle to identify the transfer request queued at the SQI driver.
	DRV_SQI_EVENT_HANDLER	Pointer to a SQI Driver Event handler function

Description

SQI Driver Interface Definition

The SQI driver provides data structures and interfaces to manage the SQI controller. This file contains the data structures and interface definitions of the SQI driver.

File Name

drv_sqi.h

Company

Microchip Technology Inc.

drv_sqi_config_template.h

SQI driver configuration definitions.

Macros

	Name	Description
	DRV_SQI_BUFFER_OBJECT_NUMBER	Selects the maximum number of buffer objects
	DRV_SQI_CLIENTS_NUMBER	Selects the maximum number of clients
	DRV_SQI_DMA_BUFFER_DESCRIPTORS_NUMBER	Selects the maximum number of DMA Buffer descriptors to be used by the driver.
	DRV_SQI_INSTANCES_NUMBER	Selects the maximum number of Driver instances that can be supported by the dynamic driver.
	DRV_SQI_INTERRUPT_MODE	Macro specifies operation of the driver to be in the interrupt mode or polled mode

Description

SQI Driver Configuration Template Header file.

This template file describes all the mandatory and optional configuration macros that are needed for building the SQI driver. Do not include this file in source code.

File Name

drv_sqi_config_template.h

Company

Microchip Technology Inc.

SQI Flash Driver Library

This section describes the Serial Quad Interface (SQI) Flash Driver Library.

Introduction

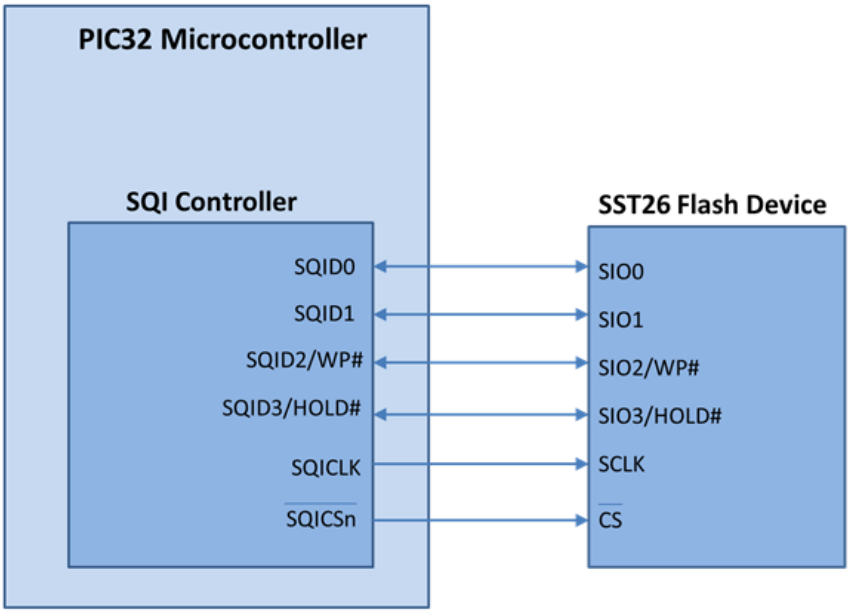
This library provides an interface to manage the SST26VF family of SQI Flash devices in different modes of operation.

Description

The MPLAB Harmony SST26 SQI Flash Driver provides a high-level interface to manage the SST26VF family of Flash devices over the SQI interface. The driver includes the following features:

- Provides application ready routines to perform block operations on the SQI Flash devices
- Supports Single, Dual, and Quad Lane modes
- Supports multi-client operation
- Provides data transfer events
- Supports non-blocking mode of operation only
- Thread-safe functions for use in RTOS applications

The SST26 Flash Driver uses the SQI module to establish the communication between SST26 Flash devices and Microchip microcontrollers. The following diagram shows the pin connections that are required to make the driver operational:



The SST26 Flash driver supports multi-client operation. This feature allows multiple application clients to access the same Flash device. Multiple instances of the driver can be used when multiple Flash devices are required to be part of the system.

Using the Library

This topic describes the basic architecture of the SQI Flash Driver Library and provides information and examples on its use.

Description

Interface Header Files: `drv_sst26.h`

The interface to the SQI Flash Driver Library is defined in the header file. Any C language source (`.c`) file that uses the SQI Flash Driver library should include this header.

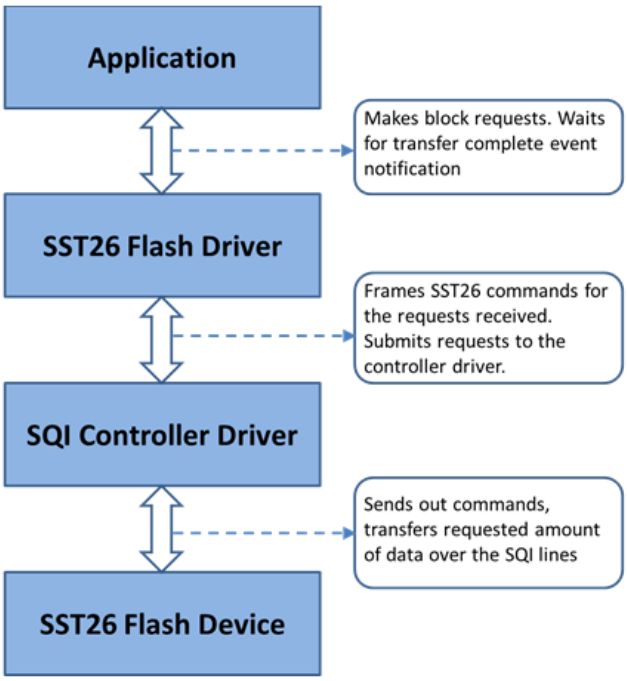
Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the SQI Flash Driver Library with a convenient C language interface. This topic describes how that abstraction is modeled in software.

Description

The SST26 SQI Flash needs a specific set of commands to be given on its SQI interface along with the required address and data to do different operations. This driver abstracts these requirements and provide simple APIs that can be used to perform Erase, Write, and Read operations. The SQI Driver is used for this purpose. The following layered diagram depicts the communication between different modules.



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the SQI Flash Driver module.

Library Interface Section	Description
System Functions	These functions are accessed by the MPLAB Harmony System module and allow the driver to be initialized, deinitialized, and maintained.
Core Client Functions	These functions allow the application client to open and close the driver.
Block Operation Functions	These functions enable the Flash module to be erased, written, and read (to/from).
Media Interface Functions	These functions provide media status and the Flash geometry.

How the Library Works

This topic provides information on how the SQI Flash Driver Library works.

Description

System Functions

SST26 Driver Initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization each instance of the SST26 Flash driver would be initialized with the following configuration settings passed dynamically at run time using DRV_SST26_INIT, that are supported by the specific SST26 Flash driver:

- sqiDevice: The SQI controller supports a maximum of two slave devices. This identifies the SQI device index on which the flash device is located.

The DRV_SST26_Initialize function configures and initializes the SST26 Flash driver using the configuration information provided. It returns an object handle of the type SYS_MODULE_OBJ. This object handle would be used by other system interfaces such as DRV_SST26_Status, DRV_SST26_Tasks and DRV_SST26_Deinitialize.

Example:

```
/*** SST26 FLASH Driver Initialization Data ***/
const DRV_SST26_INIT drvSst26InitData0 =
{
    .sqiDevice = 1,
};
```

```
/* Initialize the SST26 Driver */
sysObj.drvSst26Obj0 = DRV_SST26_Initialize(DRV_SST26_INDEX_0,
                                           (SYS_MODULE_INIT *)&drvSst26InitData0);
```

SST26 Flash Driver Task Routine

The SST26 Driver task routine DRV_SST26_Tasks, will be called from the system task routine, SYS_Tasks. The driver task routine is responsible maintaining the driver state machine. The block operation requests from the application or from other modules are added to the driver queue. The task routine processes these queued requests by invoking the SQI driver routines for handling the transfer to the flash media.

SST26 Flash Driver Status

DRV_SST26_Status returns the current status of the SST26 Flash driver and is called by MPLAB Harmony. The application may not find the need to call this function directly.

Example:

```
SYS_MODULE_OBJ object;
// Returned from DRV_SST26_Initialize
SYS_STATUS sst26Status;

sst26Status = DRV_SST26_Status(object);
if (SYS_STATUS_ERROR >= sst26Status)
{
    // Handle error
}
```

Client Core Functions

Opening the Driver

For the application to start using an instance of the module, it must call the DRV_SST26_Open function repeatedly until a valid handle is returned by the driver. The application client uses this driver handle to access the driver functions.

For the various options available for I/O INTENT please refer to Data Types and Constants in the Library Interface section.

Example:

```
handle = DRV_SST26_Open(DRV_SST26_INDEX_0,
         DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    /* Call until the function returns a valid handle. */
}
else
{
    /* Do further processing. */
}
```

Closing the Driver

DRV_SST26_Close closes an opened-instance of the SST26 Flash driver, which also invalidates the driver handle. The application must open the driver again to obtain a valid handle.

Example:

```
DRV_HANDLE handle; // Returned from DRV_SST26_Open
DRV_SST26_Close(handle);
```

Client Block Operation Functions

The driver provides client interfaces to perform operations in terms of blocks. A block is a unit that represents the minimum amount of data that can be erased, written, or read. The block sizes may differ for Erase, Write, and Read operations. The DRV_SST26_GeometryGet function can be used to read out the geometry of the flash device. The geometry indicates the number of read, write and erase regions, blocks per region and the size of each block.

The DRV_SST26_Erase, DRV_SST26_Write, and DRV_SST26_Read functions are used to erase, write, and read the data to/from SST26 Flash devices. In addition to these functions, the driver also provides the DRV_SST26_EraseWrite function that combines the step of erasing a sector and then writing a page. The application can use this function if it wants to avoid having to explicitly delete a sector in order to update the pages contained in the sector.

These functions are non-blocking in nature and queue the operation request into the driver queue. All of the requests in the queue are executed by the DRV_SST26_Tasks function one-by-one. A command handle associated with the operation request is returned to the application client when the operation request is queued at the driver. This handle allows the application client to track the request as it progresses through the queue. The handle expires when the request processing is complete. The driver provides events (DRV_SST26_EVENT) that indicate the completion of the requests.

The following steps can be used for a simple Block Data Operation:

1. The system should have completed necessary initialization of the SQI Driver and the SST26 Flash Driver, and the DRV_SST26_Tasks function should be running in a polled environment.
2. Open the driver using DRV_SST26_Open with the necessary intent.
3. Set an event handler callback using the function DRV_SST26_EventHandlerSet.
4. Request for block operations using the functions, DRV_SST26_Erase, DRV_SST26_Write, DRV_SST26_Read and DRV_SST26_EraseWrite with the appropriate parameters.
5. Wait for event handler callback to occur and check the status of the block operation using the callback function parameter of type DRV_SST26_ EVENT.
6. After performing the required block operations, the client can close the driver using the function , DRV_SST25VF020B_Close .

Example:

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_SST26_COMMAND_HANDLE commandHandle;

// drvSST26Handle is the handle returned by the DRV_SST26_Open
// function. Client registers an event handler with driver. This is done once.

DRV_SST26_EventHandlerSet(drvSST26Handle, APP_SST26EventHandler, (uintptr_t)&myAppObj);

DRV_SST26_Read(drvSST26Handle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SST26_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_SST26EventHandler
(
    DRV_SST26_EVENT event,
    DRV_SST26_COMMAND_HANDLE commandHandle,
    uintptr_t contextHandle
)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event
    // handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_SST26_EVENT_COMMAND_COMPLETE:
            // Operation completed successfully.
            break;

        case DRV_SST26_EVENT_COMMAND_ERROR:
            // Error handling here.
            break;

        default:
            break;
    }
}
```

Media Interface Functions

Reading the Device Geometry

The application can call the DRV_SST26_GeometryGet function to obtain the geometry of the flash device. The geometry indicates the number of read, write and erase regions, number of blocks per region and the size of each block.

Example:

```
SYS_FS_MEDIA_GEOMETRY * sst26FlashGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;
```

```
sst26FlashGeometry = DRV_SST26_GeometryGet(sst26OpenHandle1);

readBlockSize  = sst26FlashGeometry->geometryTable->blockSize;
nReadBlocks = sst26FlashGeometry->geometryTable->numBlocks;
nReadRegions = sst26FlashGeometry->numReadRegions;

writeBlockSize  = (sst26FlashGeometry->geometryTable +1)->blockSize;
eraseBlockSize  = (sst26FlashGeometry->geometryTable +2)->blockSize;

//The below expression provides the flash memory size.
totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Configuring the Library

Macros

	Name	Description
	DRV_SST26_BUFFER_OBJECT_NUMBER	Selects the maximum number of buffer objects
	DRV_SST26_CLIENTS_NUMBER	Selects the maximum number of clients
	DRV_SST26_INSTANCES_NUMBER	Selects the maximum number of Driver instances that can be supported by the dynamic driver.
	DRV_SST26_SYS_FS_REGISTER	Register to use with the File system

Description

The SQI Flash Driver requires the specification of compile-time configuration macros. These macros define resource usage, feature availability, and dynamic behavior of the driver. These configuration macros should be defined in the `system_config.h` file.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_SST26_BUFFER_OBJECT_NUMBER Macro

Selects the maximum number of buffer objects

File

drv_sst26_config_template.h

C

```
#define DRV_SST26_BUFFER_OBJECT_NUMBER 5
```

Description

SST26 Driver maximum number of buffer objects

This definition selects the maximum number of buffer objects. This indirectly also specifies the queue depth. The SST26 Driver can queue up DRV_SST26_BUFFER_OBJECT_NUMBER of read/write/erase requests before return a DRV_SST26_BUFFER_HANDLE_INVALID due to the queue being full. Buffer objects are shared by all instances of the driver. Increasing this number increases the RAM requirement of the driver.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_SST26_CLIENTS_NUMBER Macro

Selects the maximum number of clients

File

drv_sst26_config_template.h

C

```
#define DRV_SST26_CLIENTS_NUMBER 1
```

Description

SST26 maximum number of clients

This definition selects the maximum number of clients that the SST26 driver can supported at run time. This constant defines the total number of SST26 driver clients that will be available to all instances of the SST26 driver.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_SST26_INSTANCES_NUMBER Macro

Selects the maximum number of Driver instances that can be supported by the dynamic driver.

File

drv_sst26_config_template.h

C

```
#define DRV_SST26_INSTANCES_NUMBER 1
```

Description

SST26 Driver instance configuration

This definition selects the maximum number of Driver instances that can be supported by the dynamic driver. In case of this driver, multiple instances of the driver could use the same hardware instance.

Remarks

This macro is mandatory when building the driver for dynamic operation.

DRV_SST26_SYS_FS_REGISTER Macro

Register to use with the File system

File

drv_sst26_config_template.h

C

```
#define DRV_SST26_SYS_FS_REGISTER
```

Description

SST26 Driver Register with File System

Specifying this macro enables the SST26 driver to register its services with the SYS FS.

Remarks

This macro is optional and should be specified only if the SST26 driver is to be used with the File System.

Building the Library

This section lists the files that are available in the SQI Flash Driver Library.

Description

This section list the files that are available in the `/src` folder of the SQI Flash Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/sqi_flash`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`sst26/drv_sst26.h`	Header file that exports the SST26VF driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
sst26/src/dynamic/drv_sst26.c	Basic SQI Flash Driver SST26VF implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The SQI Flash Driver Library depends on the following modules:

- SQI Driver Library
- Ports System Service Library

Library Interface

a) System Functions

	Name	Description
	DRV_SST26_Initialize	Initializes the SST26 instance for the specified driver index
	DRV_SST26_Deinitialize	Deinitializes the specified instance of the SST26 driver module
	DRV_SST26_Status	Gets the current status of the SST26 driver module.
	DRV_SST26_Tasks	Maintains the SST26 driver's internal state machine.

b) Core Client Functions

	Name	Description
	DRV_SST26_Open	Opens the specified SST26 driver instance and returns a handle to it
	DRV_SST26_Close	Closes an opened-instance of the SST26 driver

c) Block Operation Functions

	Name	Description
	DRV_SST26_Erase	Erase the specified number of flash blocks from the specified block start address.
	DRV_SST26_EraseWrite	Erase and Write blocks of data starting from a specified block start address.
	DRV_SST26_Read	Reads blocks of data from the specified block start address.
	DRV_SST26_Write	Writes blocks of data starting at the specified block start address.
	DRV_SST26_CommandStatus	Gets the current status of the command.
	DRV_SST26_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

d) Media Interface Functions

	Name	Description
	DRV_SST26_AddressGet	Returns the SST26 media start address
	DRV_SST26_GeometryGet	Returns the geometry of the device.
	DRV_SST26_IsAttached	Returns the physical attach status of the SST26.
	DRV_SST26_IsWriteProtected	Returns the write protect status of the SST26.

e) Data Types and Constants

	Name	Description
	DRV_SST26_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_SST26_COMMAND_STATUS	SST26 Driver command Status
	DRV_SST26_EVENT	Identifies the possible events that can result from a request.
	DRV_SST26_EVENT_HANDLER	Pointer to a SST26 Driver Event handler function
	DRV_SST26_INIT	Defines the data required to initialize or reinitialize the SST26 driver
	DRV_SST26_COMMAND_HANDLE_INVALID	This value defines the SST26 Driver's Invalid Command Handle.
	DRV_SST26_INDEX_0	SST26 driver index definitions
	DRV_SST26_INDEX_1	This is macro DRV_SST26_INDEX_1.

Description

This section describes the API functions of the SQI Flash Driver Library.

Refer to each section for a detailed description.

a) System Functions

DRV_SST26_Initialize Function

Initializes the SST26 instance for the specified driver index

File

drv_sst26.h

C

```
SYS_MODULE_OBJ DRV_SST26_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the SST26 driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This routine must be called before any other SST26 routine is called.

This routine should only be called once during system initialization unless DRV_SST26_Deinitialize is called to deinitialize the driver instance.

This routine will NEVER block for hardware access. If the operation requires time to allow the hardware to initialize, it will be reported by the DRV_SST26_Status operation. The system must use DRV_SST26_Status to find out when the driver is in the ready state.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the SST26 Driver.

SYS_MODULE_OBJ  objectHandle;

const DRV_SST26_INIT drvSst26InitData0 =
{
    .sqiDevice = 1,
};

//usage of DRV_SST26_INDEX_0 indicates usage of Flash-related APIs
objectHandle = DRV_SST26_Initialize(DRV_SST26_INDEX_0, (SYS_MODULE_INIT*)&drvSst26InitData0);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_SST26_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

);

DRV_SST26_Deinitialize Function

Deinitializes the specified instance of the SST26 driver module

File

drv_sst26.h

C

```
void DRV_SST26_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SST26 driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

Preconditions

Function DRV_SST26_Initialize should have been called before calling this function.

Parameter: object - Driver object handle, returned from the DRV_SST26_Initialize routine

Example

```
// This code snippet shows an example of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_SST26_Initialize
SYS_STATUS          status;

DRV_SST26_Deinitialize(object);

status = DRV_SST26_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know when the driver is
    // deinitialized.
}
```

Function

void DRV_SST26_Deinitialize

(

SYS_MODULE_OBJ object

);

DRV_SST26_Status Function

Gets the current status of the SST26 driver module.

File

drv_sst26.h

C

```
SYS_STATUS DRV_SST26_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and accept requests for new operations.

SYS_STATUS_UNINITIALIZED - Indicates the driver is not initialized.

Description

This routine provides the current status of the SST26 driver module.

Remarks

This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_SST26_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST26_Initialize
SYS_STATUS          SST26Status;

SST26Status = DRV_SST26_Status(object);
else if (SYS_STATUS_ERROR >= SST26Status)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SST26_Initialize routine

Function

SYS_STATUS DRV_SST26_Status

(

SYS_MODULE_OBJ object

);

DRV_SST26_Tasks Function

Maintains the SST26 driver's internal state machine.

File

drv_sst26.h

C

```
void DRV_SST26_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine maintains the driver's internal state machine. Part of the driver initialization is done in this routine. This routine is responsible for processing the read, write, erase or erasewrite requests queued for the SST26 driver.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks).

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_SST26_Initialize routine must have been called for the specified SST26 driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SST26_Initialize

while (true)
{
    DRV_SST26_Tasks (object);
    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_SST26_Initialize)

Function

void DRV_SST26_Tasks
(
SYS_MODULE_OBJ object
);

b) Core Client Functions

DRV_SST26_Open Function

Opens the specified SST26 driver instance and returns a handle to it

File

drv_sst26.h

C

```
DRV_HANDLE DRV_SST26_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, DRV_HANDLE_INVALID is returned. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_SST26_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver hardware instance being opened is not initialized or is invalid

Description

This routine opens the specified SST26 driver instance and provides a handle. This handle must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_SST26_Close routine is called. This routine will NEVER block waiting for hardware. If the driver has has already been opened, it cannot be opened exclusively.

Preconditions

Function DRV_SST26_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_SST26_Open(DRV_SST26_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_SST26_Open

(
const SYS_MODULE_INDEX index,
const DRV_IO_INTENT ioIntent
);

DRV_SST26_Close Function

Closes an opened-instance of the SST26 driver

File

drv_sst26.h

C

```
void DRV_SST26_Close(const DRV_HANDLE handle);
```

Returns

None

Description

This routine closes an opened-instance of the SST26 driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_SST26_Open before the caller may use the driver again. Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_SST26_Initialize routine must have been called for the specified SST26 driver instance.

DRV_SST26_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_SST26_Open

DRV_SST26_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_SST26_Close
(
const DRV_HANDLE handle
);

c) Block Operation Functions

DRV_SST26_Erase Function

Erase the specified number of flash blocks from the specified block start address.

File

drv_sst26.h

C

```
void DRV_SST26_Erase(const DRV_HANDLE handle, DRV_SST26_COMMAND_HANDLE * commandHandle, uint32_t
blockStart, uint32_t nBlock);
```

Returns

The command handle is returned in the commandHandle argument. It Will be DRV_SST26_COMMAND_HANDLE_INVALID if the request was not queued.

Description

This function schedules a non-blocking erase operation of flash memory. The function returns with a valid erase handle in the commandHandle argument if the erase request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_SST26_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer object could not be allocated to the request
- if the client opened the driver for read only
- if the number of blocks to be erased is either zero or more than the number of blocks actually available
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST26_EVENT_COMMAND_COMPLETE event if the erase operation was successful or DRV_SST26_EVENT_COMMAND_ERROR event if the erase operation was not successful.

Remarks

None.

Preconditions

The DRV_SST26_Initialize() routine must have been called for the specified SST26 driver instance.

The DRV_SST26_Open() routine must have been called with DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE to obtain a valid opened device handle.

Example

```
// Use DRV_SST26_GeometryGet () to find the read region geometry.
// Find the erase block start address from where the number of blocks
// should be erased.
uint32_t blockStart = 0;
uint32_t nBlock = 4;
DRV_SST26_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST26Handle is the handle returned by the DRV_SST26_Open function.

// Client registers an event handler with driver
DRV_SST26_EventHandlerSet(mySST26Handle, APP_SST26EventHandler, (uintptr_t)&myAppObj);

DRV_SST26_Erase( mySST26Handle, &commandHandle, blockStart, nBlock );

if(DRV_SST26_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when the buffer queue is processed.

void APP_SST26EventHandler
(
    DRV_SST26_EVENT event,
    DRV_SST26_COMMAND_HANDLE commandHandle,
    uintptr_t contextHandle
)
{
    // contextHandle points to myAppObj.
    switch(event)
    {
        case DRV_SST26_EVENT_COMMAND_COMPLETE:
            // Erase operation completled successfully.
            break;

        case DRV_SST26_EVENT_COMMAND_ERROR:
            // Error handling here.
            break;
```

```
            default:
                break;
        }
    }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
blockStart	Erase block start address from where the blocks should be erased.
nBlock	Total number of blocks to be erased.

Function

void DRV_SST26_Erase
(
const DRV_HANDLE handle,
DRV_SST26_COMMAND_HANDLE * commandHandle,
uint32_t blockStart,
uint32_t nBlock
);

DRV_SST26_EraseWrite Function

Erase and Write blocks of data starting from a specified block start address.

File

drv_sst26.h

C

```
void DRV_SST26_EraseWrite(const DRV_HANDLE handle, DRV_SST26_COMMAND_HANDLE * commandHandle, void *
sourceBuffer, uint32_t writeBlockStart, uint32_t nWriteBlock);
```

Returns

The command handle is returned in the commandHandle argument. It Will be DRV_SST26_COMMAND_HANDLE_INVALID if the request was not queued.

Description

This function combines the step of erasing a sector and then writing the page. The application can use this function if it wants to avoid having to explicitly delete a sector in order to update the pages contained in the sector.

This function schedules a non-blocking operation to erase and write blocks of data into flash memory. The function returns with a valid command handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST26_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read only
- if the number of blocks to be written is either zero or more than the number of blocks actually available
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST26_EVENT_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_SST26_EVENT_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

None.

Preconditions

The DRV_SST26_Initialize() routine must have been called for the specified SST26 driver instance.

The DRV_SST26_Open() must have been called with DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE as a parameter to obtain a valid opened device handle.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// Use DRV_SST26_GeometryGet () to find the write region geometry.
// Find the block address to which data is to be written.
uint32_t blockStart = SST26_BLOCK_ADDRESS_TO_WRITE_TO;
uint32_t nBlock = 2;
DRV_SST26_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST26Handle is the handle returned by the DRV_SST26_Open function.
// Client registers an event handler with driver

DRV_SST26_EventHandlerSet(mySST26Handle, APP_SST26EventHandler, (uintptr_t)&myAppObj);

DRV_SST26_EraseWrite(mySST26Handle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SST26_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_SST26EventHandler
(
    DRV_SST26_EVENT event,
    DRV_SST26_COMMAND_HANDLE commandHandle,
    uintptr_t contextHandle
)
{
    // contextHandle points to myAppObj.
    switch(event)
    {
        case DRV_SST26_EVENT_COMMAND_COMPLETE:
            // Operation completled successfully.
            break;

        case DRV_SST26_EVENT_COMMAND_ERROR:
            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return command handle. If NULL, then command handle is not returned.
sourceBuffer	The source buffer containing data to be programmed into SST26 Flash
writeBlockStart	Write block start address where the write should begin.
nWriteBlock	Total number of blocks to be written.

Function

void DRV_SST26_EraseWrite

(

const DRV_HANDLE handle,

DRV_SST26_COMMAND_HANDLE * commandHandle,

void * sourceBuffer,

uint32_t writeBlockStart,

uint32_t nWriteBlock

```
);
```

DRV_SST26_Read Function

Reads blocks of data from the specified block start address.

File

drv_sst26.h

C

```
void DRV_SST26_Read(const DRV_HANDLE handle, DRV_SST26_COMMAND_HANDLE * commandHandle, void * targetBuffer,
uint32_t blockStart, uint32_t nBlock);
```

Returns

The command handle is returned in the commandHandle argument. It will be DRV_SST26_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking read operation for reading blocks of data from the flash memory. The function returns with a valid command handle in the commandHandle argument if the request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST26_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer object could not be allocated to the request
- if the target buffer pointer is NULL
- if the client opened the driver for write only
- if the number of blocks to be read is either zero or more than the number of blocks actually available
- if the driver handle is invalid

Remarks

None.

Preconditions

The DRV_SST26_Initialize routine must have been called for the specified SST26 driver instance.

DRV_SST26_Open must have been called with DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE as the ioIntent to obtain a valid opened device handle.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// Use DRV_SST26_GeometryGet () to find the read region geometry.
// Find the block address from which to read data.
uint32_t blockStart = SST26_BLOCK_ADDRESS_TO_READ_FROM;
uint32_t nBlock = 2;
DRV_SST26_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST26Handle is the handle returned by the DRV_SST26_Open function.
// Client registers an event handler with driver

DRV_SST26_EventHandlerSet(mySST26Handle, APP_SST26EventHandler, (uintptr_t)&myAppObj);

DRV_SST26_Read(mySST26Handle, &commandHandle, &myBuffer, blockStart, nBlock);
if(DRV_SST26_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}
else
{
    // Read queued successfully.
}

// Event is received when the command request is processed.

void APP_SST26EventHandler
(
```

```
    DRV_SST26_EVENT event,
    DRV_SST26_COMMAND_HANDLE commandHandle,
    uintptr_t contextHandle
)
{
    // contextHandle points to myAppObj.
    switch(event)
    {
        case DRV_SST26_EVENT_COMMAND_COMPLETE:
            // This means the data was transferred.
            break;

        case DRV_SST26_EVENT_COMMAND_ERROR:
            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the command handle
targetBuffer	Buffer into which the data read from the SST26 Flash memory will be placed
blockStart	Read block start address from where the data should be read.
nBlock	Total number of blocks to be read.

Function

void DRV_SST26_Read
(
const DRV_HANDLE handle,
DRV_SST26_COMMAND_HANDLE * commandHandle,
void * targetBuffer,
uint32_t blockStart,
uint32_t nBlock
);

DRV_SST26_Write Function

Writes blocks of data starting at the specified block start address.

File

drv_sst26.h

C

```
void DRV_SST26_Write(const DRV_HANDLE handle, DRV_SST26_COMMAND_HANDLE * commandHandle, void *
sourceBuffer, uint32_t blockStart, uint32_t nBlock);
```

Returns

The command handle is returned in the commandHandle argument. It will be DRV_SST26_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking write operation for writing blocks of data into flash memory. The function returns with a valid command handle in the commandHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_SST26_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if a buffer object could not be allocated to the request
- if the source buffer pointer is NULL
- if the client opened the driver for read only

- if the number of blocks to be written is either zero or more than the number of blocks actually available
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_SST26_EVENT_COMMAND_COMPLETE event if the buffer was processed successfully or DRV_SST26_EVENT_COMMAND_ERROR event if the buffer was not processed successfully.

Remarks

None.

Preconditions

The DRV_SST26_Initialize() routine must have been called for the specified SST26 driver instance.

DRV_SST26_Open() routine must have been called to obtain a valid opened device handle. DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified as a parameter to this routine.

The flash address location which has to be written, must have be erased before using the DRV_SST26_Erase() routine.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];

// Use DRV_SST26_GeometryGet () to find the write region geometry.
// Find the block address to which data is to be written.
uint32_t blockStart = SST26_BLOCK_ADDRESS_TO_WRITE_TO;
uint32_t nBlock = 2;
DRV_SST26_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySST26Handle is the handle returned by the DRV_SST26_Open function.
// Client registers an event handler with driver

DRV_SST26_EventHandlerSet(mySST26Handle, APP_SST26EventHandler, (uintptr_t)&myAppObj);

DRV_SST26_Write(mySST26Handle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SST26_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event is received when the buffer is processed.

void APP_SST26EventHandler
(
    DRV_SST26_EVENT event,
    DRV_SST26_COMMAND_HANDLE commandHandle,
    uintptr_t contextHandle
)
{
    // contextHandle points to myAppObj.
    switch(event)
    {
        case DRV_SST26_EVENT_COMMAND_COMPLETE:
            // This means the data was transferred.
            break;

        case DRV_SST26_EVENT_COMMAND_ERROR:
            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
sourceBuffer	The source buffer containing data to be programmed into SST26 Flash

blockStart	Write block start address from where the data should be written to.
nBlock	Total number of blocks to be written.

Function

void DRV_SST26_Write
(
const DRV_HANDLE handle,
DRV_SST26_COMMAND_HANDLE * commandHandle,
void * sourceBuffer,
uint32_t blockStart,
uint32_t nBlock
);

DRV_SST26_CommandStatus Function

Gets the current status of the command.

File

drv_sst26.h

C

```
DRV_SST26_COMMAND_STATUS DRV_SST26_CommandStatus(const DRV_HANDLE handle, const DRV_SST26_COMMAND_HANDLE commandHandle);
```

Returns

A DRV_SST26_COMMAND_STATUS value describing the current status of the command. Returns DRV_SST26_COMMAND_HANDLE_INVALID if the client handle or the command handle is not valid.

Description

This routine gets the current status of the command. The application must use this routine where the status of a scheduled command needs to be polled on. The function may return DRV_SST26_COMMAND_COMPLETED in a case where the command handle has expired. A command handle expires when the internal buffer object is re-assigned to another request. It is recommended that this function be called regularly in order to track the command status correctly.

The application can alternatively register an event handler to receive the command completion events.

Remarks

This routine will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_SST26_Initialize() routine must have been called.

The DRV_SST26_Open() must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE                  handle;         // Returned from DRV_SST26_Open
DRV_SST26_COMMAND_HANDLE      commandHandle;
DRV_SST26_COMMAND_STATUS      status;

status = DRV_SST26_CommandStatus(handle, commandHandle);
if(status == DRV_SST26_COMMAND_COMPLETED)
{
    // Operation Done
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_SST26_COMMAND_STATUS DRV_SST26_CommandStatus
(
const DRV_HANDLE handle,

```
    const       DRV_SST26_COMMAND_HANDLE commandHandle
    );
```

DRV_SST26_EventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when queued operation has completed.

File

drv_sst26.h

C

```
void DRV_SST26_EventHandlerSet(const DRV_HANDLE handle, const void * eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when queued operation has completed. When a client calls a read, write, erase or a erasewrite function, it is provided with a handle identifying the command that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the queued operation has completed.

The event handler should be set before the client performs any operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_SST26_Initialize() routine must have been called for the specified SST26 driver instance.

The DRV_SST26_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_SST26_COMMAND_HANDLE commandHandle;

// drvSST26Handle is the handle returned by the DRV_SST26_Open function.
// Client registers an event handler with driver. This is done once.

DRV_SST26_EventHandlerSet(drvSST26Handle, APP_SST26EventHandler, (uintptr_t)&myAppObj);

DRV_SST26_Read(drvSST26Handle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SST26_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.

void APP_SST26EventHandler
(
    DRV_SST26_EVENT event,
    DRV_SST26_COMMAND_HANDLE commandHandle,
    uintptr_t contextHandle
)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
```

```
            case DRV_SST26_EVENT_COMMAND_COMPLETE:
                // Operation completled successfully.
                break;

            case DRV_SST26_EVENT_COMMAND_ERROR:
                // Error handling here.
                break;

            default:
                break;
        }
    }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_SST26_EventHandlerSet
(
const DRV_HANDLE handle,
const void * eventHandler,
const uintptr_t context
);

d) Media Interface Functions

DRV_SST26_AddressGet Function

Returns the SST26 media start address

File

drv_sst26.h

C

```
uintptr_t DRV_SST26_AddressGet(const DRV_HANDLE handle);
```

Returns

Start address of the SST26 Media if the handle is valid otherwise NULL.

Description

This function returns the SST26 Media start address.

Remarks

None.

Preconditions

The DRV_SST26_Initialize() routine must have been called for the specified SST26 driver instance.

The DRV_SST26_Open() routine must have been called to obtain a valid opened device handle.

Example

```
uintptr_t startAddress;
startAddress = DRV_SST26_AddressGet(drvSST26Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

uintptr_t DRV_SST26_AddressGet
(
const DRV_HANDLE handle
);

DRV_SST26_GeometryGet Function

Returns the geometry of the device.

File

drv_sst26.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_SST26_GeometryGet(const DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Pointer to structure which holds the media geometry information.

Description

This API gives the following geometrical details of the SST26 Flash:

- Media Property
- Number of Read/Write/Erase regions in the flash device
- Number of Blocks and their size in each region of the device

Remarks

None.

Preconditions

The DRV_SST26_Initialize() routine must have been called for the specified SST26 driver instance.

The DRV_SST26_Open() routine must have been called to obtain a valid opened device handle.

Example

```
SYS_FS_MEDIA_GEOMETRY * sst26FlashGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalFlashSize;

sst26FlashGeometry = DRV_SST26_GeometryGet(sst26OpenHandle1);

readBlockSize  = sst26FlashGeometry->geometryTable->blockSize;
nReadBlocks = sst26FlashGeometry->geometryTable->numBlocks;
nReadRegions = sst26FlashGeometry->numReadRegions;

writeBlockSize  = (sst26FlashGeometry->geometryTable +1)->blockSize;
eraseBlockSize  = (sst26FlashGeometry->geometryTable +2)->blockSize;

totalFlashSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY * DRV_SST26_GeometryGet
(
const DRV_HANDLE handle

);

DRV_SST26_IsAttached Function

Returns the physical attach status of the SST26.

File

drv_sst26.h

C

```
bool DRV_SST26_IsAttached(const DRV_HANDLE handle);
```

Returns

Returns false if the handle is invalid otherwise returns true.

Description

This function returns the physical attach status of the SST26.

Remarks

None.

Preconditions

The DRV_SST26_Initialize() routine must have been called for the specified SST26 driver instance.

The DRV_SST26_Open() routine must have been called to obtain a valid opened device handle.

Example

```
bool isSST26Attached;
isSST26Attached = DRV_SST26_isAttached(drvSST26Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_SST26_IsAttached

(

const DRV_HANDLE handle

);

DRV_SST26_IsWriteProtected Function

Returns the write protect status of the SST26.

File

drv_sst26.h

C

```
bool DRV_SST26_IsWriteProtected(const DRV_HANDLE handle);
```

Returns

True - If the flash is write protected. False - If the flash is not write protected.

Description

This function returns the write protect status of the SST26.

Remarks

None.

Preconditions

The DRV_SST26_Initialize() routine must have been called for the specified SST26 driver instance.

The DRV_SST26_Open() routine must have been called to obtain a valid opened device handle.

Example

```
bool isWriteProtected;
isWriteProtected = DRV_SST26_IsWriteProtected(drvSST26Handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_SST26_IsWriteProtected

(

const DRV_HANDLE handle

);

e) Data Types and Constants

DRV_SST26_COMMAND_HANDLE Type

Handle identifying commands queued in the driver.

File

drv_sst26.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_SST26_COMMAND_HANDLE;
```

Description

SST26 Driver command handle.

A command handle is returned by a call to the Read, Write, Erase or EraseWrite functions. This handle allows the application to track the completion of the operation. This command handle is also returned to the client along with the event that has occurred with respect to the command. This allows the application to connect the event to a specific command in case where multiple commands are queued.

The command handle associated with the command request expires when the client has been notified of the completion of the command (after event handler function that notifies the client returns) or after the command has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_SST26_COMMAND_STATUS Enumeration

SST26 Driver command Status

File

drv_sst26.h

C

```
typedef enum {
  DRV_SST26_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED,
  DRV_SST26_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED,
  DRV_SST26_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS,
  DRV_SST26_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN
} DRV_SST26_COMMAND_STATUS;
```

Members

Members	Description
DRV_SST26_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED	Done OK and ready
DRV_SST26_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED	Scheduled but not started
DRV_SST26_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS	Currently being in transfer

DRV_SST26_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN	Unknown Command

Description

SST26 Driver Command Status

Specifies the status of the command for the read, write, erase and erasewrite operations.

Remarks

None.

DRV_SST26_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_sst26.h

C

```
typedef enum {
  DRV_SST26_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_SST26_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR
} DRV_SST26_EVENT;
```

Members

Members	Description
DRV_SST26_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE	Operation has been completed successfully.
DRV_SST26_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR	There was an error during the operation

Description

SST26 Driver Events

This enumeration identifies the possible events that can result from a read, write, erase or erasewrite request caused by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_SST26_EventHandlerSet function when a request is completed.

DRV_SST26_EVENT_HANDLER Type

Pointer to a SST26 Driver Event handler function

File

drv_sst26.h

C

```
typedef SYS_FS_MEDIA_EVENT_HANDLER DRV_SST26_EVENT_HANDLER;
```

Returns

None.

Description

SST26 Driver Event Handler Function Pointer

This data type defines the required function signature for the SST26 event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event calls back from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_SST26_EVENT_COMMAND_COMPLETE, it means that the requested operation was completed successfully.

If the event is DRV_SST26_EVENT_COMMAND_ERROR, it means that the scheduled operation was not completed successfully.

The context parameter contains the handle to the client context, provided at the time the event handling function was registered using the DRV_SST26_EventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write/erase request.

The event handler function executes in the driver peripheral's interrupt context when the driver is configured for interrupt mode operation. It is recommended of the application to not perform process intensive or blocking operations within this function.

Example

```
void APP_MySst26EventHandler
(
    DRV_SST26_EVENT event,
    DRV_SST26_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_SST26_EVENT_COMMAND_COMPLETE:
            // Handle the completed buffer.
            break;

        case DRV_SST26_EVENT_COMMAND_ERROR:
        default:
            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write/Erase/EraseWrite requests
context	Value identifying the context of the application that registered the event handling function

DRV_SST26_INIT Structure

Defines the data required to initialize or reinitialize the SST26 driver

File

drv_sst26.h

C

```
typedef struct {
  uint8_t sqiDevice;
} DRV_SST26_INIT;
```

Members

Members	Description
uint8_t sqiDevice;	SQI Device Index.

Description

SST26 Driver Initialization Data

This data type defines the data required to initialize or reinitialize the SST26 driver.

Remarks

Not all initialization features are available for all devices. Please refer to the specific device data sheet to determine availability.

DRV_SST26_COMMAND_HANDLE_INVALID Macro

This value defines the SST26 Driver's Invalid Command Handle.

File

drv_sst26.h

C

```
#define DRV_SST26_COMMAND_HANDLE_INVALID SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID
```

Description

SST26 Driver Invalid Command Handle.

This value defines the SST26 Driver's Invalid Command Handle. This value is returned by read/write/erase/erasewrite routines when the command request was not accepted.

Remarks

None.

DRV_SST26_INDEX_0 Macro

SST26 driver index definitions

File

drv_sst26.h

C

```
#define DRV_SST26_INDEX_0 0
```

Description

Driver SST26 Module Index reference

These constants provide SST26 driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_SST26_Initialize and DRV_SST26_Open routines to identify the driver instance in use.

DRV_SST26_INDEX_1 Macro

File

drv_sst26.h

C

```
#define DRV_SST26_INDEX_1 1
```

Description

This is macro DRV_SST26_INDEX_1.

Files

Files

Name	Description
drv_sst26.h	SST26 Driver Interface Definition
drv_sst26_config_template.h	SST26 driver configuration definitions.

Description

This section lists the source and header files used by the SQI Flash Driver Library.

drv_sst26.h

SST26 Driver Interface Definition

Enumerations

	Name	Description
	DRV_SST26_COMMAND_STATUS	SST26 Driver command Status
	DRV_SST26_EVENT	Identifies the possible events that can result from a request.

Functions

	Name	Description
	DRV_SST26_AddressGet	Returns the SST26 media start address
	DRV_SST26_Close	Closes an opened-instance of the SST26 driver
	DRV_SST26_CommandStatus	Gets the current status of the command.
	DRV_SST26_Deinitialize	Deinitializes the specified instance of the SST26 driver module
	DRV_SST26_Erase	Erase the specified number of flash blocks from the specified block start address.
	DRV_SST26_EraseWrite	Erase and Write blocks of data starting from a specified block start address.
	DRV_SST26_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when queued operation has completed.
	DRV_SST26_GeometryGet	Returns the geometry of the device.
	DRV_SST26_Initialize	Initializes the SST26 instance for the specified driver index
	DRV_SST26_IsAttached	Returns the physical attach status of the SST26.
	DRV_SST26_IsWriteProtected	Returns the write protect status of the SST26.
	DRV_SST26_Open	Opens the specified SST26 driver instance and returns a handle to it
	DRV_SST26_Read	Reads blocks of data from the specified block start address.
	DRV_SST26_Status	Gets the current status of the SST26 driver module.
	DRV_SST26_Tasks	Maintains the SST26 driver's internal state machine.
	DRV_SST26_Write	Writes blocks of data starting at the specified block start address.

Macros

	Name	Description
	DRV_SST26_COMMAND_HANDLE_INVALID	This value defines the SST26 Driver's Invalid Command Handle.
	DRV_SST26_INDEX_0	SST26 driver index definitions
	DRV_SST26_INDEX_1	This is macro DRV_SST26_INDEX_1.

Structures

	Name	Description
	DRV_SST26_INIT	Defines the data required to initialize or reinitialize the SST26 driver

Types

	Name	Description
	DRV_SST26_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_SST26_EVENT_HANDLER	Pointer to a SST26 Driver Event handler function

Description

SST26 Driver Interface Definition

The SST26 driver provides a simple interface to manage the SST26VF series of SQI Flash Memory connected to Microchip microcontrollers. This file defines the interface definition for the SST26 driver.

File Name

drv_sst26.h

Company

Microchip Technology Inc.

drv_sst26_config_template.h

SST26 driver configuration definitions.

Macros

	Name	Description
	DRV_SST26_BUFFER_OBJECT_NUMBER	Selects the maximum number of buffer objects
	DRV_SST26_CLIENTS_NUMBER	Selects the maximum number of clients
	DRV_SST26_INSTANCES_NUMBER	Selects the maximum number of Driver instances that can be supported by the dynamic driver.
	DRV_SST26_SYS_FS_REGISTER	Register to use with the File system

Description

SST26 Driver Configuration Template Header file.

This template file describes all the mandatory and optional configuration macros that are needed for building the SST26 driver. Do not include this file in source code.

File Name

drv_sst26_config_template.h

Company

Microchip Technology Inc.

SRAM Driver Library

This section describes the Static Random Access (SRAM) driver library.

Introduction

The SRAM Media Driver library provides a high-level interface to manage the onboard SRAM as a media

Description

The SRAM Media driver features the following:

- Provides application ready routines to perform block operations on the SRAM media
- Supports multi-client operation
- Provides data transfer events
- Supports blocking mode of operation only

Using the Library

This topic describes the basic architecture of the SRAM Media Driver Library and provides information and examples about how to use it.

Description

Interface Header Files: `drv_sram.h`

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Topics

Name	Description
Abstraction Model	This library provides a low-level abstraction of the SRAM media with a convenient C language interface. This topic describes how that abstraction is modeled in software.
Library Overview	This library provides information about how the driver operates in a system. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the SRAM Media driver module.
How the Library Works	This section describes how the SRAM Media Driver Library operates

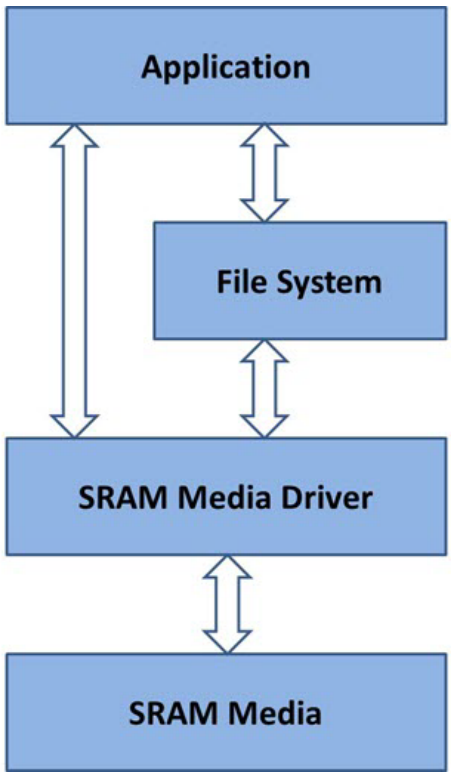
Abstraction Model

This library provides a low-level abstraction of the SRAM media with a convenient C language interface. This section describes how that abstraction is modeled in software.

Description

The SRAM Media driver facilitates the block access to the SRAM media by providing APIs that can be used to perform read/write operations. The end applications can either access the media directly through the driver or through MPLAB Harmony. The following diagram shows the communication between different modules.

SRAM Media Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information about how the SRAM driver operates within a system.

Description

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the SRAM module.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, de-initialization and status functions.
Data Types and Constants	Provides data types and macros.

How the Library Works

The library provides interfaces to support:

- System Functionality
- Client Functionality
- Media Functionality

System Initialization/Status Functions

The SRAM driver provides the following system functions:

- SRAM driver initialization
- SRAM driver status.

Description

SRAM driver initialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized. During system initialization each instance of the SRAM Media driver would be initialized with the following configuration settings passed dynamically at run time using DRV_SRAM_INIT:

- registerWithFs: This controls the registration of the driver with the Harmony File System.
- mediaStartAddress: This indicates the start address of the SRAM media on which driver is to operate.
- sramMediaGeometry: This provides the SRAM media geometry information that the driver will use while performing the read/write operations on the media.

The DRV_SRAM_Initialize function configures and initializes the SRAM Media driver by using the configuration information provided. The function returns an object handle of the type SYS_MODULE_OBJ. This object handle is used by other system interfaces such as DRV_SRAM_Status and DRV_SRAM_Deinitialize.

Example:

```
/*** SRAM Driver Initialization Data ***/
SYS_FS_MEDIA_REGION_GEOMETRY sramMedia0GeometryTable[3] =
{
{
.blockSize = 1,
.numBlocks = (32 * (1024/1)),
},
{
.blockSize = 1,
.numBlocks = (32 * (1024/1)),
},
{
.blockSize = 1,
.numBlocks = (32 * (1024/1)),
}
};

const SYS_FS_MEDIA_GEOMETRY sramMedia0Geometry =
{
.mediaProperty = SYS_FS_MEDIA_WRITE_IS_BLOCKING | SYS_FS_MEDIA_READ_IS_BLOCKING,
.numReadRegions = 1,
.numWriteRegions = 1,
.numEraseRegions = 1,
.geometryTable = (SYS_FS_MEDIA_REGION_GEOMETRY *)&sramMedia0GeometryTable
};

extern uint8_t SRAM_MEDIA_0_DATA[];
const DRV_SRAM_INIT drvSram0Init =
{
.registerWithFs = true,
.mediaStartAddress = (uint8_t *)SRAM_MEDIA_0_DATA,
.sramMediaGeometry = (SYS_FS_MEDIA_GEOMETRY
*)&sramMedia0Geometry
};

/* Initialize the SRAM Driver Instance 0 */
sysObj.drvSramObj0 = DRV_SRAM_Initialize(DRV_SRAM_INDEX_0, (SYS_MODULE_INIT *)&drvSram0Init);
```

SRAM driver status

DRV_SRAM_Status() returns the current status of the SRAM Media driver and is called by the Harmony System. The application may not find the need to call this function directly.

Example:

```
SYS_MODULE_OBJ object;
// Returned from DRV_SRAM_Initialize
```

```
SYS_STATUS sramStatus;

sramStatus = DRV_SRAM_Status(object);
if (SYS_STATUS_ERROR >= sramStatus)
{
// Handle error
}
```

Client Core Functions

The SRAM driver provides the following client core functions:

- Opening the driver
- Closing the driver

Description

Opening the driver

For the application to start using an instance of the module, it must call the function DRV_SRAM_Open and obtain a valid driver handle. The application client uses this driver handle to access the driver functions.

For the various options available for I/O INTENT, please refer to Data Types and Constants in the Library Interface section.

Example:

```
handle = DRV_SRAM_Open(DRV_SRAM_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
/* Call until the function returns a valid handle. */
}
else
{
/* Do further processing. */
}
```

Closing the driver

This function closes an opened-instance of the SRAM Media driver. This invalidates the driver handle. The application must open the driver again to obtain a valid handle.

Example:

```
DRV_HANDLE handle; // Returned from DRV_SRAM_Open
DRV_SRAM_Close(handle);
```

Client Block Operation Functions

The SRAM driver provides client interfaces to perform operations in terms of blocks.

Description

A block is a unit that represents the minimum amount of data that can be written or read. The block sizes may differ for write and read operations. The DRV_SRAM_GeometryGet function can be used to read out the geometry of the SRAM Media. The geometry indicates the number of read, write, and erase regions; blocks per region; and the size of each block.

The DRV_SRAM_Write and DRV_SRAM_Read functions are used to write and read the data to/from SRAM media. These functions block operations until the requested amount of data is transferred. If an event handler has been registered to receive the driver events, then the event handler will be invoked from within these functions. The driver provides events (DRV_SRAM_EVENT) that indicate the completion of the requests.

The following steps can be performed for a simple block data transfer operation:

1. Make sure the system has completed the necessary initialization of the SRAM driver.
2. Open the driver using DRV_SRAM_Open with the necessary intent.
3. Register an event handler callback by using the function DRV_SRAM_EventHandlerSet.
4. Request for block operation by using the functions DRV_SRAM_Write and DRV_SRAM_Read with the appropriate parameters.
5. Wait for the event handler callback to occur and check the status of the block operation by using the callback function parameter of type DRV_SRAM_ EVENT.
6. After performing the required block operations, the client can close the driver by using the function DRV_SRAM_Close.

Example:

```
uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart = 0;
uint32_t nBlock = 2;
```

```
DRV_SRAM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySRAMHandle is the handle returned by the DRV_SRAM_Open
// function.

DRV_SRAM_EventHandlerSet(
mySRAMHandle,
APP_SRAMEventHandler,
(uintptr_t)&myAppObj);

DRV_SRAM_Read(
mySRAMHandle,
&commandHandle,
&myBuffer,
blockStart,
nBlock);
if(DRV_SRAM_COMMAND_HANDLE_INVALID == commandHandle)
{
// Error handling here
}
else
{
// Read operation completed successfully.
}

// Event is invoked from within the DRV_SRAM_Read function when
// the read operation processing is complete.

void APP_SRAMEventHandler
(
 DRV_SRAM_EVENT event,
 DRV_SRAM_COMMAND_HANDLE commandHandle,
uintptr_t contextHandle
)
{
// contextHandle points to myAppObj.
switch(event)
{
case DRV_SRAM_EVENT_COMMAND_COMPLETE:
// This means the data was transferred.
break;

case DRV_SRAM_EVENT_COMMAND_ERROR:
// Error handling here.
break;

default:
break;
}
}
```

Media Interface Functions

The SRAM driver provides the following media interface function:

- Reading the device geometry

Description

The application can call the DRV_SRAM_GeometryGet function to obtain the geometry of the media device.

Reading the device geometry

The geometry indicates the number of read, write, and erase regions; the number of blocks per region; and the size of each block.

Example:

```
SYS_FS_MEDIA_GEOMETRY * sramGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalSize;

sramGeometry = DRV_SRAM_GeometryGet(sramOpenHandle1);

readBlockSize = sramGeometry->geometryTable->blockSize;
nReadBlocks = sramGeometry->geometryTable->numBlocks;
nReadRegions = sramGeometry->numReadRegions;

writeBlockSize = (sramGeometry->geometryTable +1)->blockSize;
eraseBlockSize = (sramGeometry->geometryTable +2)->blockSize;
totalSize = readBlockSize * nReadBlocks * nReadRegions;
```

Configuring the Library

This section contains related configuration macros.

Description

The configuration of the SRAM driver is based on the file `system_config.h`.

This header file contains the configuration selection for the SRAM driver. Based on the selections made, the SRAM driver may support the selected features. These configuration settings apply to all instances of the SRAM driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Building the Library

This section lists the files that are available in the SRAM library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/sram/`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_sram.h`	This is the SRAM library's interface header file.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_sram.c`	This file contains the source code for the dynamic implementation of the SRAM driver.
`/config/drv_sram_config_template.h`	This file contains configuration macros for the SRAM driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Library Interface

This section describes the Application Programming Interface (API) functions of the SRAM driver library.

Refer to each section for a detailed description.

a) System Functions

	Name	Description
	DRV_SRAM_AddressGet	Returns the SRAM media start address
	DRV_SRAM_Close	Closes an opened-instance of the SRAM driver
	DRV_SRAM_CommandStatus	Gets the current status of the command.
	DRV_SRAM_Deinitialize	Deinitializes the specified instance of the SRAM driver module
	DRV_SRAM_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when an operation has completed.
	DRV_SRAM_GeometryGet	Returns the geometry of the device.
	DRV_SRAM_Initialize	Initializes the SRAM instance for the specified driver index.
	DRV_SRAM_IsAttached	Returns the physical attach status of the SRAM.
	DRV_SRAM_IsWriteProtected	Returns the write protect status of the SRAM.
	DRV_SRAM_Open	Opens the specified SRAM driver instance and returns a handle to it
	DRV_SRAM_Read	Reads blocks of data from the specified block start address.
	DRV_SRAM_Status	Gets the current status of the SRAM driver module.
	DRV_SRAM_Write	Writes blocks of data starting from the specified block start address of the SRAM media.

c) Data Types and Constants

	Name	Description
	DRV_SRAM_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_SRAM_COMMAND_STATUS	Specifies the status of the command for the read and write operations.
	DRV_SRAM_EVENT	Identifies the possible events that can result from a request.
	DRV_SRAM_EVENT_HANDLER	Pointer to a SRAM Driver Event handler function
	DRV_SRAM_INIT	Defines the data required to initialize the SRAM driver
	_DRV_SRAM_H	This is macro _DRV_SRAM_H.
	DRV_SRAM_COMMAND_HANDLE_INVALID	This value defines the SRAM Driver's Invalid Command Handle.
	DRV_SRAM_INDEX_0	SRAM driver index definitions
	DRV_SRAM_INDEX_1	This is macro DRV_SRAM_INDEX_1.

a) System Functions

DRV_SRAM_AddressGet Function

Returns the SRAM media start address

File

drv_sram.h

C

```
uintptr_t DRV_SRAM_AddressGet(const DRV_HANDLE handle);
```

Returns

Start address of the SRAM Media if the handle is valid otherwise NULL.

Description

This function returns the SRAM Media start address.

Remarks

None.

Preconditions

The DRV_SRAM_Initialize() routine must have been called for the specified SRAM driver instance.

The DRV_SRAM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
uintptr_t startAddress;
startAddress = DRV_SRAM_AddressGet(drvSRAMHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open

Function

uintptr_t DRV_SRAM_AddressGet

(

const DRV_HANDLE handle

);

DRV_SRAM_Close Function

Closes an opened-instance of the SRAM driver

File

drv_sram.h

C

```
void DRV_SRAM_Close(const DRV_HANDLE handle);
```

Returns

None

Description

This routine closes an opened-instance of the SRAM driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_SRAM_Open before the caller may use the driver again. Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_SRAM_Initialize routine must have been called for the specified SRAM driver instance.

DRV_SRAM_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_SRAM_Open

DRV_SRAM_Close(handle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_SRAM_Close

(

const DRV_HANDLE handle

);

DRV_SRAM_CommandStatus Function

Gets the current status of the command.

File

drv_sram.h

C

```
DRV_SRAM_COMMAND_STATUS DRV_SRAM_CommandStatus(const DRV_HANDLE handle, const DRV_SRAM_COMMAND_HANDLE
commandHandle);
```

Returns

A DRV_SRAM_COMMAND_STATUS value describing the current status of the command. Returns DRV_SRAM_COMMAND_COMPLETED if the client handle or the command handle is not valid.

Description

This routine gets the current status of the command. The application must use this routine where the status of a scheduled command needs to be polled on. The function may return DRV_SRAM_COMMAND_COMPLETED in a case where the command handle has expired. A command handle expires when the internal buffer object is re-assigned to another read or write request. It is recommended that this function be called regularly in order to track the command status correctly.

The application can alternatively register an event handler to receive read or write operation completion events.

Remarks

This routine will not block for hardware access and will immediately return the current status.

Preconditions

The DRV_SRAM_Initialize() routine must have been called.

The DRV_SRAM_Open() must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE                  handle;         // Returned from DRV_SRAM_Open
DRV_SRAM_COMMAND_HANDLE     commandHandle;
DRV_SRAM_COMMAND_STATUS     status;

status = DRV_SRAM_CommandStatus(handle, commandHandle);
if(status == DRV_SRAM_COMMAND_COMPLETED)
{
    // Operation Done
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_SRAM_COMMAND_STATUS DRV_SRAM_CommandStatus

(

const DRV_HANDLE handle,

const DRV_SRAM_COMMAND_HANDLE commandHandle

);

DRV_SRAM_Deinitialize Function

Deinitializes the specified instance of the SRAM driver module

File

drv_sram.h

C

```
void DRV_SRAM_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the SRAM driver module, disabling its operation. Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

Preconditions

Function DRV_SRAM_Initialize should have been called before calling this function.

Parameter: object - Driver object handle, returned from the DRV_SRAM_Initialize routine

Example

```
// This code snippet shows an example of deinitializing the driver.

SYS_MODULE_OBJ      object;     //  Returned from DRV_SRAM_Initialize
SYS_STATUS          status;

DRV_SRAM_Deinitialize(object);

status = DRV_SRAM_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know when the driver is
    // deinitialized.
}
```

Function

void DRV_SRAM_Deinitialize

(

SYS_MODULE_OBJ object

);

DRV_SRAM_EventHandlerSet Function

Allows a client to identify an event handling function for the driver to call back when an operation has completed.

File

drv_sram.h

C

```
void DRV_SRAM_EventHandlerSet(const DRV_HANDLE handle, const void * eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the driver to call back when an operation has completed. When a client calls a read or a write function, it is provided with a handle identifying the read/write request. The driver will pass this handle back to the client by calling "eventHandler" function when the operation has completed.

The event handler should be set before the client performs any read or write operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued operation has completed, it does not need to register a callback.

Preconditions

The DRV_SRAM_Initialize() routine must have been called for the specified SRAM driver instance.

The DRV_SRAM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart, nBlock;
DRV_SRAM_COMMAND_HANDLE commandHandle;

// drvSRAMHandle is the handle returned by the DRV_SRAM_Open function.
// Client registers an event handler with driver. This is done once.

DRV_SRAM_EventHandlerSet(drvSRAMHandle, APP_SRAMEventHandler, (uintptr_t)&myAppObj);

DRV_SRAM_Read(drvSRAMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SRAM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when operation is done.
void APP_SRAMEventHandler
(
    DRV_SRAM_EVENT event,
    DRV_SRAM_COMMAND_HANDLE handle,
    uintptr_t context
)
{
    // The context handle was set to an application specific object. It is
    // now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_SRAM_EVENT_COMMAND_COMPLETE:
            // This means the data was transferred.
            break;

        case DRV_SRAM_EVENT_COMMAND_ERROR:
            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_SRAM_EventHandlerSet

(

const DRV_HANDLE handle,

const void * eventHandler,

const uintptr_t context

);

DRV_SRAM_GeometryGet Function

Returns the geometry of the device.

File

drv_sram.h

C

```
SYS_FS_MEDIA_GEOMETRY * DRV_SRAM_GeometryGet(const DRV_HANDLE handle);
```

Returns

SYS_FS_MEDIA_GEOMETRY - Pointer to structure which holds the media geometry information.

Description

This API gives the following geometrical details of the SRAM memory:

- Media Property
- Number of Read/Write/Erase regions
- Number of Blocks and their size in each region of the device

Remarks

None.

Preconditions

The DRV_SRAM_Initialize() routine must have been called for the specified SRAM driver instance.

The DRV_SRAM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
SYS_FS_MEDIA_GEOMETRY * sramGeometry;
uint32_t readBlockSize, writeBlockSize, eraseBlockSize;
uint32_t nReadBlocks, nReadRegions, totalSize;

sramGeometry = DRV_SRAM_GeometryGet(sramOpenHandle1);

readBlockSize  = sramGeometry->geometryTable->blockSize;
nReadBlocks = sramGeometry->geometryTable->numBlocks;
nReadRegions = sramGeometry->numReadRegions;

writeBlockSize  = (sramGeometry->geometryTable +1)->blockSize;
eraseBlockSize  = (sramGeometry->geometryTable +2)->blockSize;

totalSize = readBlockSize * nReadBlocks * nReadRegions;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

SYS_FS_MEDIA_GEOMETRY * DRV_SRAM_GeometryGet

(

const DRV_HANDLE handle

);

DRV_SRAM_Initialize Function

Initializes the SRAM instance for the specified driver index.

File

drv_sram.h

C

```
SYS_MODULE_OBJ DRV_SRAM_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise it returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the SRAM driver instance for the specified driver index, making it ready for clients to open and use it.

Remarks

This routine must be called before any other SRAM routine is called.

This routine should only be called once during system initialization unless DRV_SRAM_Deinitialize is called to deinitialize the driver instance.

This routine will NEVER block for hardware access. The system must use DRV_SRAM_Status to find out when the driver is in the ready state.

Preconditions

None.

Example

```
// This code snippet shows an example of initializing the SRAM Driver.

SYS_MODULE_OBJ  objectHandle;

SYS_FS_MEDIA_REGION_GEOMETRY gSramGeometryTable[3] =
{
    {
        // Read Region Geometry
        .blockSize = 512,
        .numBlocks = (DRV_SRAM_MEDIA_SIZE * (1024/512)),
    },
    {
        // Write Region Geometry
        .blockSize = 512,
        .numBlocks = ((DRV_SRAM_MEDIA_SIZE * (1024/512))
    },
    {
        // Erase Region Geometry
        .blockSize = 512,
        .numBlocks = ((DRV_SRAM_MEDIA_SIZE * (1024/512))
    }
};

const SYS_FS_MEDIA_GEOMETRY gSramGeometry =
{
    .mediaProperty = SYS_FS_MEDIA_WRITE_IS_BLOCKING,

    // Number of read, write and erase entries in the table
    .numReadRegions = 1,
    .numWriteRegions = 1,
    .numEraseRegions = 1,
    .geometryTable = &gSramGeometryTable
};

// SRAM Driver Initialization Data
const DRV_SRAM_INIT drvSramInit =
{
    .mediaStartAddress = DRV_SRAM_MEDIA_START_ADDRESS,
    .sramMediaGeometry = &gSramGeometry
};

objectHandle = DRV_SRAM_Initialize(DRV_SRAM_INDEX_0, (SYS_MODULE_INIT*)&drvSRAMInit);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized.

init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_SRAM_Initialize
(
const SYS_MODULE_INDEX index,
const SYS_MODULE_INIT * const init
);

DRV_SRAM_IsAttached Function

Returns the physical attach status of the SRAM.

File

drv_sram.h

C

```
bool DRV_SRAM_IsAttached(const DRV_HANDLE handle);
```

Returns

Returns false if the handle is invalid otherwise returns true.

Description

This function returns the physical attach status of the SRAM.

Remarks

None.

Preconditions

The DRV_SRAM_Initialize() routine must have been called for the specified SRAM driver instance.
The DRV_SRAM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// The SRAM media is always attached and so the below always returns true.
bool isSRAMAttached;
isSRAMAttached = DRV_SRAM_isAttached(drvSRAMHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open

Function

bool DRV_SRAM_IsAttached
(
const DRV_HANDLE handle
);

DRV_SRAM_IsWriteProtected Function

Returns the write protect status of the SRAM.

File

drv_sram.h

C

```
bool DRV_SRAM_IsWriteProtected(const DRV_HANDLE handle);
```

Returns

Always returns false.

Description

This function returns the physical attach status of the SRAM. This function always returns false.

Remarks

None.

Preconditions

The DRV_SRAM_Initialize() routine must have been called for the specified SRAM driver instance.

The DRV_SRAM_Open() routine must have been called to obtain a valid opened device handle.

Example

```
// The SRAM media is treated as always writeable.
bool isWriteProtected;
isWriteProtected = DRV_SRAM_IsWriteProtected(drvSRAMHandle);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function

Function

bool DRV_SRAM_IsWriteProtected

(

const DRV_HANDLE handle

);

DRV_SRAM_Open Function

Opens the specified SRAM driver instance and returns a handle to it

File

drv_sram.h

C

```
DRV_HANDLE DRV_SRAM_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, DRV_HANDLE_INVALID is returned. Errors can occur under the following circumstances:

- if the number of client objects allocated via DRV_SRAM_CLIENTS_NUMBER

is insufficient

- if the client is trying to open the driver but driver has been opened

exclusively by another client

- if the client is trying to open the driver exclusively, but has

already been opened in a non exclusive mode by another client.

- if the driver hardware instance being opened is invalid

Description

This routine opens the specified SRAM driver instance and provides a handle. This handle must be provided to all other client-level operations to identify the caller and the instance of the driver.

Remarks

The handle returned is valid until the DRV_SRAM_Close routine is called. This routine will NEVER block waiting for hardware. If the driver has has already been opened, it cannot be opened exclusively.

Preconditions

DRV_SRAM_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;
```

```
handle = DRV_SRAM_Open(DRV_SRAM_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_SRAM_Open

(

const SYS_MODULE_INDEX index,

const DRV_IO_INTENT ioIntent

);

DRV_SRAM_Read Function

Reads blocks of data from the specified block start address.

File

drv_sram.h

C

```
void DRV_SRAM_Read(const DRV_HANDLE handle, DRV_SRAM_COMMAND_HANDLE * commandHandle, void * targetBuffer,
uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_SRAM_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This routine reads blocks of data from the specified block start address. This operation is blocking and returns with the required data in the target buffer. If a event handler has been registered to receive the driver events then the event handler will be called from within this function. The function returns DRV_SRAM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the driver handle is invalid
- if the driver state is not ready
- if the target buffer pointer is NULL
- if the number of blocks to be read is zero or more than the actual number

of blocks available

- if the client opened the driver in write only mode

Remarks

None.

Preconditions

The DRV_SRAM_Initialize routine must have been called for the specified SRAM driver instance.

DRV_SRAM_Open must have been called with DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE as the ioIntent to obtain a valid opened device handle.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart = 0;
uint32_t nBlock = 2;
DRV_SRAM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySRAMHandle is the handle returned by the DRV_SRAM_Open function.
```

```
DRV_SRAM_EventHandlerSet(mySRAMHandle, APP_SRAMEventHandler, (uintptr_t)&myAppObj);
DRV_SRAM_Read(mySRAMHandle, &commandHandle, &myBuffer, blockStart, nBlock);
if(DRV_SRAM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}
else
{
    // Read operation completed successfully.
}

// Event is invoked from within the DRV_SRAM_Read function when the read
// operation processing is complete.

void APP_SRAMEventHandler
(
    DRV_SRAM_EVENT event,
    DRV_SRAM_COMMAND_HANDLE commandHandle,
    uintptr_t contextHandle
)
{
    // contextHandle points to myAppObj.
    switch(event)
    {
        case DRV_SRAM_EVENT_COMMAND_COMPLETE:
            // This means the data was transferred.
            break;

        case DRV_SRAM_EVENT_COMMAND_ERROR:
            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle
targetBuffer	Buffer into which the data read from the SRAM memory will be placed
blockStart	SRAM media's start block address from where the read should begin.
nBlock	Total number of blocks to be read.

Function

```
void DRV_SRAM_Read
(
const       DRV_HANDLE handle,
    DRV_SRAM_COMMAND_HANDLE * commandHandle,
void * targetBuffer,
uint32_t blockStart,
uint32_t nBlock
);
```

DRV_SRAM_Status Function

Gets the current status of the SRAM driver module.

File

drv_sram.h

C

```
SYS_STATUS DRV_SRAM_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is ready and accept requests for new operations.

SYS_STATUS_UNINITIALIZED - Indicates the driver is not initialized.

Description

This routine provides the current status of the SRAM driver module.

Remarks

None.

Preconditions

Function DRV_SRAM_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_SRAM_Initialize
SYS_STATUS          SRAMStatus;

SRAMStatus = DRV_SRAM_Status(object);
if (SRAMStatus == SYS_STATUS_READY)
{
    // Driver is ready to process read/write operations.
}
else
{
    // Driver is not ready.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_SRAM_Initialize routine

Function

SYS_STATUS DRV_SRAM_Status

(

SYS_MODULE_OBJ object

);

DRV_SRAM_Write Function

Writes blocks of data starting from the specified block start address of the SRAM media.

File

drv_sram.h

C

```
void DRV_SRAM_Write(const DRV_HANDLE handle, DRV_SRAM_COMMAND_HANDLE * commandHandle, void * sourceBuffer,
uint32_t blockStart, uint32_t nBlock);
```

Returns

The buffer handle is returned in the commandHandle argument. It will be DRV_SRAM_COMMAND_HANDLE_INVALID if the request was not successful.

Description

This routine writes blocks of data starting at the specified block start address. This operation is blocking and returns after having written the data. If a event handler has been registered to receive the driver events then the event handler will be called from within this function. The function returns DRV_SRAM_COMMAND_HANDLE_INVALID in the commandHandle argument under the following circumstances:

- if the driver handle is invalid

- if the driver state is not ready
- if the source buffer pointer is NULL
- if the number of blocks to be written is zero or more than the actual

number of blocks available

- if the client opened the driver in read only mode

Remarks

None

Preconditions

The DRV_SRAM_Initialize() routine must have been called for the specified SRAM driver instance.

DRV_SRAM_Open() routine must have been called to obtain a valid opened device handle. DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified as a parameter to this routine.

Example

```
uint8_t myBuffer[MY_BUFFER_SIZE];
uint32_t blockStart = 2;
uint32_t nBlock = 2;
DRV_SRAM_COMMAND_HANDLE commandHandle;
MY_APP_OBJ myAppObj;

// mySRAMHandle is the handle returned by the DRV_SRAM_Open function.
// Client registers an event handler with driver

DRV_SRAM_EventHandlerSet(mySRAMHandle, APP_SRAMEventHandler, (uintptr_t)&myAppObj);
DRV_SRAM_Write(mySRAMHandle, &commandHandle, &myBuffer, blockStart, nBlock);

if(DRV_SRAM_COMMAND_HANDLE_INVALID == commandHandle)
{
    // Error handling here
}
else
{
    // Write completed successfully.
}

// Event is received from within the DRV_SRAM_Write function when the
// buffer is processed.

void APP_SRAMEventHandler
(
    DRV_SRAM_EVENT event,
    DRV_SRAM_COMMAND_HANDLE commandHandle,
    uintptr_t contextHandle
)
{
    // contextHandle points to myAppObj.
    switch(event)
    {
        case DRV_SRAM_EVENT_COMMAND_COMPLETE:
            // This means the data was transferred.
            break;

        case DRV_SRAM_EVENT_COMMAND_ERROR:
            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open function
commandHandle	Pointer to an argument that will contain the return buffer handle

sourceBuffer	The source buffer containing data to be programmed into SRAM memory
blockStart	Start block address of SRAM media from where the write should begin.
nBlock	Total number of blocks to be written.

Function

```
void DRV_SRAM_Write
(
const       DRV_HANDLE handle,
    DRV_SRAM_COMMAND_HANDLE * commandHandle,
void * sourceBuffer,
uint32_t blockStart,
uint32_t nBlock
);
```

c) Data Types and Constants

DRV_SRAM_COMMAND_HANDLE Type

Handle identifying commands queued in the driver.

File

drv_sram.h

C

```
typedef SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE DRV_SRAM_COMMAND_HANDLE;
```

Description

SRAM Driver command handle.

A command handle is returned by a call to the Read or Write functions. This handle allows the application to track the completion of the operation. This command handle is also returned to the client along with the event that has occurred with respect to the command. This allows the application to connect the event to a specific command in case where multiple commands are queued.

The command handle associated with the command request expires when the client has been notified of the completion of the command (after event handler function that notifies the client returns) or after the command has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_SRAM_COMMAND_STATUS Enumeration

Specifies the status of the command for the read and write operations.

File

drv_sram.h

C

```
typedef enum {
  DRV_SRAM_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED,
  DRV_SRAM_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED,
  DRV_SRAM_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS,
  DRV_SRAM_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN
} DRV_SRAM_COMMAND_STATUS;
```

Members

Members	Description
DRV_SRAM_COMMAND_COMPLETED = SYS_FS_MEDIA_COMMAND_COMPLETED	Done OK and ready
DRV_SRAM_COMMAND_QUEUED = SYS_FS_MEDIA_COMMAND_QUEUED	Scheduled but not started
DRV_SRAM_COMMAND_IN_PROGRESS = SYS_FS_MEDIA_COMMAND_IN_PROGRESS	Currently being in transfer

DRV_SRAM_COMMAND_ERROR_UNKNOWN = SYS_FS_MEDIA_COMMAND_UNKNOWN	Unknown Command

Description

SRAM Driver Command Status

SRAM Driver command Status

This type specifies the status of the command for the read and write operations.

Remarks

None.

DRV_SRAM_EVENT Enumeration

Identifies the possible events that can result from a request.

File

drv_sram.h

C

```
typedef enum {
  DRV_SRAM_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE,
  DRV_SRAM_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR
} DRV_SRAM_EVENT;
```

Members

Members	Description
DRV_SRAM_EVENT_COMMAND_COMPLETE = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE	Operation has been completed successfully.
DRV_SRAM_EVENT_COMMAND_ERROR = SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR	There was an error during the operation

Description

SRAM Driver Events

This enumeration identifies the possible events that can result from a read or a write request caused by the client.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the driver by calling the DRV_SRAM_EventHandlerSet function when a request is completed.

DRV_SRAM_EVENT_HANDLER Type

Pointer to a SRAM Driver Event handler function

File

drv_sram.h

C

```
typedef SYS_FS_MEDIA_EVENT_HANDLER DRV_SRAM_EVENT_HANDLER;
```

Returns

None.

Description

SRAM Driver Event Handler Function Pointer

This data type defines the required function signature for the SRAM event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event callbacks from the driver.

The parameters and return values are described here and a partial example implementation is provided.

Remarks

If the event is DRV_SRAM_EVENT_COMMAND_COMPLETE, it means that the read or write operation was completed successfully.

If the event is DRV_SRAM_EVENT_COMMAND_ERROR, it means that the scheduled operation was not completed successfully.

The context parameter contains the handle to the client context, provided at the time the event handling function was registered using the DRV_SRAM_EventHandlerSet function. This context handle value is passed back to the client as the "context" parameter. It can be any value necessary to identify the client context or instance (such as a pointer to the client's data) instance of the client that made the read/write request.

The event handler function executes in the driver's context. It is recommended of the application to not perform process intensive or blocking operations within this function.

Example

```
void APP_MySramEventHandler
(
    DRV_SRAM_EVENT event,
    DRV_SRAM_COMMAND_HANDLE commandHandle,
    uintptr_t context
)
{
    MY_APP_DATA_STRUCT pAppData = (MY_APP_DATA_STRUCT) context;

    switch(event)
    {
        case DRV_SRAM_EVENT_COMMAND_COMPLETE:

            // Handle the completed buffer.
            break;

        case DRV_SRAM_EVENT_COMMAND_ERROR:
        default:

            // Handle error.
            break;
    }
}
```

Parameters

Parameters	Description
event	Identifies the type of event
commandHandle	Handle returned from the Read/Write requests
context	Value identifying the context of the application that registered the event handling function

DRV_SRAM_INIT Structure

Defines the data required to initialize the SRAM driver

File

drv_sram.h

C

```
typedef struct {
  bool registerWithFs;
  uint8_t* mediaStartAddress;
  const SYS_FS_MEDIA_GEOMETRY * sramMediaGeometry;
} DRV_SRAM_INIT;
```

Members

Members	Description
bool registerWithFs;	Flag to indicate if the driver is to be registered with the file system.
uint8_t* mediaStartAddress;	SRAM Media start address. The driver treats this address as block 0 • address for read and write operations.
const SYS_FS_MEDIA_GEOMETRY * sramMediaGeometry;	SRAM Media geometry object.

Description

SRAM Driver Initialization Data

This data type defines the data required to initialize the SRAM driver.

Remarks

None.

_DRV_SRAM_H Macro

File

drv_sram.h

C

```
#define _DRV_SRAM_H
```

Description

This is macro _DRV_SRAM_H.

DRV_SRAM_COMMAND_HANDLE_INVALID Macro

This value defines the SRAM Driver's Invalid Command Handle.

File

drv_sram.h

C

```
#define DRV_SRAM_COMMAND_HANDLE_INVALID SYS_FS_MEDIA_BLOCK_COMMAND_HANDLE_INVALID
```

Description

SRAM Driver Invalid Command Handle.

This value defines the SRAM Driver Invalid Command Handle. This value is returned by read/write routines when the command request is not accepted.

Remarks

None.

DRV_SRAM_INDEX_0 Macro

SRAM driver index definitions

File

drv_sram.h

C

```
#define DRV_SRAM_INDEX_0 0
```

Description

Driver SRAM Module Index reference

These constants provide SRAM driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_SRAM_Initialize and DRV_SRAM_Open routines to identify the driver instance in use.

DRV_SRAM_INDEX_1 Macro

File

drv_sram.h

C

```
#define DRV_SRAM_INDEX_1 1
```

Description

This is macro DRV_SRAM_INDEX_1.

Files

Files

Name	Description
drv_sram.h	SRAM Driver Interface Definition

Description

This section will list only the library's interface header file(s).

drv_sram.h

SRAM Driver Interface Definition

Enumerations

	Name	Description
	DRV_SRAM_COMMAND_STATUS	Specifies the status of the command for the read and write operations.
	DRV_SRAM_EVENT	Identifies the possible events that can result from a request.

Functions

	Name	Description
	DRV_SRAM_AddressGet	Returns the SRAM media start address
	DRV_SRAM_Close	Closes an opened-instance of the SRAM driver
	DRV_SRAM_CommandStatus	Gets the current status of the command.
	DRV_SRAM_Deinitialize	Deinitializes the specified instance of the SRAM driver module
	DRV_SRAM_EventHandlerSet	Allows a client to identify an event handling function for the driver to call back when an operation has completed.
	DRV_SRAM_GeometryGet	Returns the geometry of the device.
	DRV_SRAM_Initialize	Initializes the SRAM instance for the specified driver index.
	DRV_SRAM_IsAttached	Returns the physical attach status of the SRAM.
	DRV_SRAM_IsWriteProtected	Returns the write protect status of the SRAM.
	DRV_SRAM_Open	Opens the specified SRAM driver instance and returns a handle to it
	DRV_SRAM_Read	Reads blocks of data from the specified block start address.
	DRV_SRAM_Status	Gets the current status of the SRAM driver module.
	DRV_SRAM_Write	Writes blocks of data starting from the specified block start address of the SRAM media.

Macros

	Name	Description
	_DRV_SRAM_H	This is macro _DRV_SRAM_H.
	DRV_SRAM_COMMAND_HANDLE_INVALID	This value defines the SRAM Driver's Invalid Command Handle.
	DRV_SRAM_INDEX_0	SRAM driver index definitions
	DRV_SRAM_INDEX_1	This is macro DRV_SRAM_INDEX_1.

Structures

	Name	Description
	DRV_SRAM_INIT	Defines the data required to initialize the SRAM driver

Types

	Name	Description
	DRV_SRAM_COMMAND_HANDLE	Handle identifying commands queued in the driver.
	DRV_SRAM_EVENT_HANDLER	Pointer to a SRAM Driver Event handler function

Description

SRAM Driver Interface Definition

The SRAM driver provides a simple interface to manage the SRAM Memory on Microchip microcontrollers. This file defines the interface definition for the SRAM driver.

File Name

drv_sram.h

Company

Microchip Technology Inc.

Timer Driver Library

This section describes the Timer Driver Library.

Introduction

This library provides an interface to manage the Timer module on the Microchip family of microcontrollers during different modes of operation.

Description

Timers are useful for generating accurate time based periodic interrupts for software application or real time operating systems. Other uses include counting external pulses or accurate timing measurement of external events using the timer's gate functions and accurate hardware delays.

Note: Not all features are available on all devices. Please refer to the specific device data sheet to determine availability.

Using the Library

This topic describes the basic architecture of the Timer Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_tmr.h`

The interface to the Timer Driver Library is defined in the `drv_tmr.h` header file.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

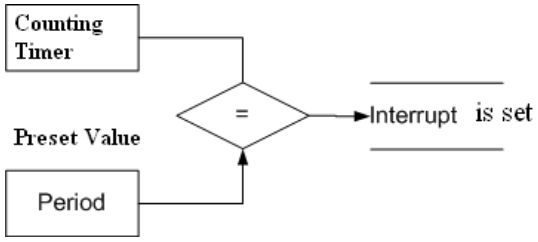
Abstraction Model

The Timer Driver abstracts the hardware by providing the capability to register callback functions to the application.

Description

Abstraction Model

The abstraction model of the Timer Driver is explained in the following diagram:



The core functionality of the Timer allows access to both the counter and the period values.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Timer Driver Library.

Library Interface Section	Description
Configuration	Provides macros for configuring the system. It is required that the system configures the driver to build correctly by choosing appropriate configuration options as listed in this section. These macros enable different features or modes of the timer peripheral.
System Interaction Functions	Provides interfaces to system layer to initialize, deinitialize and reinitialize the module. This section also describes functions to query the status of the module.
Core Functions	Provides interfaces for core functionality of the driver.
Alarm Functions	Provides interfaces to handle alarm features, if alarm functionality is enabled.
Period Functions	Provides interfaces to control the periodicity of the timers.
Counter Control Functions	Provides interfaces to update the counter values.
Miscellaneous Functions	Provides interfaces to get the version information, timer tick and operating frequencies.

How the Library Works

The library provides interfaces to support:

- System Interaction
- Sync Mode Setup
- Period Modification
- Counter Modification
- Client Core Functionality
- Client Alarm Functionality (optional function, enabled using configuration options)
- Other Optional Functionality (enabled using configuration options)

Note: Any code segment pertaining to the driver interfaces will work for both the static or dynamic configurations. It is not necessary to modify the code to move from one configuration to the other (i.e., from static or dynamic or static-multi).

System Interaction

This section describes Timer initialization and reinitialization.

Description

Initialization and Reinitialization

The system performs the initialization of the device driver with settings that affect only the instance of the device that is being initialized.

The DRV_TMR_Initialize function returns an object handle of the type SYS_MODULE_OBJ. After this, the object handle returned by the Initialize interface would be used by the other system interfaces such as DRV_TMR_Deinitialize and DRV_TMR_Status, DRV_TMR_Tasks.

Example: Timer Initialization

```
DRV_TMR_INIT     init;
SYS_MODULE_OBJ   object;
SYS_STATUS       tmrStatus;

// populate the TMR init configuration structure
init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
init.tmrId            = TMR_ID_2;
init.clockSource      = TMR_CLOCK_SOURCE_PERIPHERAL_CLOCK;
init.prescale         = TMR_PRESCALE_VALUE_256;
init.interruptSource  = INT_SOURCE_TIMER_2;
init.mode             = DRV_TMR_OPERATION_MODE_16_BIT;
init.asyncWriteEnable = false;

object = DRV_TMR_Initialize (DRV_TMR_INDEX_0, (SYS_MODULE_INIT *)&init);

if (object == SYS_MODULE_OBJ_INVALID)
{
    // Handle error
}
```

Deinitialization

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This routine may block if the driver is running in an OS environment that supports blocking operations and the driver requires system resources access. However, the routine will never block for hardware Timer access.

Status

Timer status is available to query the module state after initialization and reinitialization.

Tasks Routine

The interface DRV_TMR_Tasks needs to be called by the system task service in a polled environment and in an interrupt-based system.

Example: Polling

```
int main( void )
{
    SYS_MODULE_OBJ object;
    object = DRV_TMR_Initialize( DRV_TMR_INDEX_0, (SYS_MODULE_INIT *) &initConf );

    if( SYS_STATUS_READY != DRV_TMR_Status( object ) )
        return 0;

    while (1)
    {
        DRV_TMR_Tasks (object);
    }
}
```

Example: Interrupt

```
int main( void )
{
    SYS_MODULE_OBJ object;
    object = DRV_TMR_Initialize( DRV_TMR_INDEX_0, (SYS_MODULE_INIT *) &initConf );

    if( SYS_STATUS_READY != DRV_TMR_Status( object ) )
        return 0;

    while (1);
}

/* Sample interrupt routine not specific to any device family */
void ISR T1Interrupt(void)
{
    //Call the Timer Tasks routine
    DRV_TMR_Tasks(object);
}
```

Client Interaction

This section describes general client operation.

Description

General Client Operation

For the application to begin using an instance of the Timer module, it must call the DRV_TMR_Open function. This provides the configuration required to open the Timer instance for operation.

The Timer Driver supports only the 'DRV_IO_INTENT_EXCLUSIVE' IO_INTENT.

Example:

```
DRV_HANDLE handle;

// Configure the instance DRV_TMR_INDEX_1 with the configuration
handle = DRV_TMR_Open(DRV_TMR_INDEX_1, DRV_IO_INTENT_EXCLUSIVE);

if( handle == DRV_HANDLE_INVALID )
{
    // Client cannot open the instance.
}
```

The function DRV_TMR_Close closes an already opened instance of the Timer Driver, invalidating the handle. DRV_TMR_Open must have been

called to obtain a valid opened device handle.

Example:
```
DRV_HANDLE handle;

// Configure the instance DRV_TMR_INDEX_1 with the configuration
handle = DRV_TMR_Open(DRV_TMR_INDEX_1, DRV_IO_INTENT_EXCLUSIVE);

/*...*/

DRV_TMR_Close( handle );
```

The client has the option to check the status through the interface DRV_TMR_ClientStatus.

Example:
```
DRV_HANDLE handle;

// Configure the instance DRV_TMR_INDEX_1 with the configuration
handle = DRV_TMR_Open(DRV_TMR_INDEX_1, DRV_IO_INTENT_EXCLUSIVE);

if ( DRV_TMR_CLIENT_STATUS_READY != DRV_TMR_ClientStatus( handle ) )
   return 0;
```

Modification

This section describes Period modification for the different types of Timers (i.e., 16-/32-bit).

Description

These set of functions help modify the Timer periodicity at the client level.

Period Modification

Periodicity of Timer (16/32-bit) can be modified using DRV_TMR_AlarmPeriodSet and the current period can be obtained using DRV_TMR_AlarmPeriodGet.

Example:
```
DRV_HANDLE handle;
/* Open the client */
handle = DRV_TMR_Open( DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE );

/* ... */

/* Update the new period */
DRV_TMR_AlarmPeriodSet( handle, 0xC350);
```

Counter Modification

This section describes counter modification for the different types of Timers (i.e., 8-/16-/32-bit).

Description

These set of functions help modify the initial value of the Timer counters to help adjust any errors in the periodicity.

Counter Modification

The Timer initial value can be modified using DRV_TMR_CounterValueSet and the current counter value can be obtained using DRV_TMR_CounterValueGet.

Example:
```
DRV_HANDLE handle;
/* Open the client */
handle = DRV_TMR_Open( DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE );

/* ... */

/* Update the counter value */
/* Following code updates the initial value from 0x0000 to 0x0010
   to cover up any error in the previously set periodicity */

DRV_TMR_CounterValueSet( handle, 0x0010);
```

Core Functionality

This section describes core functionality of the Timer Driver.

Description

Core functionality provides an extremely basic interface for the driver operation.

Applications using the Timer core functionality need to perform the following:

1. The system should have completed the necessary initialization and DRV_TMR_Tasks should be called in a polled/interrupt environment.
2. Open_the driver using DRV_TMR_Open. The Timer Driver only supports exclusive access.
3. The Timer can be updated using DRV_TMR_AlarmPeriodSet. The previously set value can be retrieved using DRV_TMR_AlarmPeriodGet.
4. Start the driver using DRV_TMR_Start.
5. Poll for the elapsed alarm status using DRV_TMR_AlarmHasElapsed.
6. The client will be able to stop the started Timer instance using DRV_TMR_Stop at any time and will be able to close it using DRV_TMR_Close when it is no longer required.

Example:

```
/* Open the client */
handle = DRV_TMR_Open( DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE );
DRV_TMR_Start (handle);
unsigned int alarmCount = 0;
while (1)
{
    if (true == DRV_TMR_AlarmHasElapsed (handle))
    {
        alarmCount++;
        // Do something
    }
}
```

Notes:

1. The user needs to stop the Timer before any updates on the counter or period and restart it later.
2. The Timer alarm count gets reset after any call to DRV_TMR_AlarmHasElapsed.
3. The Timer alarm status remains unchanged if the user stops the timer and restarts later.

Alarm Functionality

This section describes the Timer Driver alarm functionality.

Description

The Timer Driver provides alarm functionality.

Applications using the Timer alarm functionality, need to perform the following:

1. The system should have completed the necessary initialization and DRV_TMR_Tasks should be running in either a polled environment or in an interrupt environment.
2. Open the driver using DRV_TMR_Open. The Timer Driver supports exclusive access only.
3. Configure the alarm using DRV_TMR_AlarmRegister.
4. Start the driver using DRV_TMR_Start.
5. If a callback is supplied, the Timer Driver will call the callback function when the alarm expires.
6. The client will be able to stop the started Timer module instance using DRV_TMR_Stop at any time and will be able to close it using DRV_TMR_Close when it is no longer required.
7. The client can deregister the callback by using DRV_TMR_AlarmDeregister.

Example:

```
DRV_HANDLE handle;
/* Open the client */
handle = DRV_TMR_Open (DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
/* Configure the timer alarm feature */
uint32_t myFreq = 1000; // 1KHz
uint32_t clkFreq = DRV_TMR_CounterFrequencyGet(tmrHandle);   // timer running frequency

// calculate the divider needed
uint32_t divider = clkFreq / myFreq;

// Start the alarm
if(!DRV_TMR_AlarmRegister ( tmrHandle, divider, true, 0, CallBackFreq ))
```

```
{
    // divider value could not be obtain;
    // handle the error
    //
    return;
}

DRV_TMR_Start (handle);

// The driver tasks function calls the client registered callback after the alarm expires.
void CallBackFreq (uintptr_t context, uint32_t alarmCount)
{
    // Do something specific on an alarm event trigger
}
```

Optional Interfaces

This section describes additional/optional client interfaces.

Description

Additional/Optional client interfaces include the following:

Get Operating Frequency

The function DRV_TMR_CounterFrequencyGet provides the client with the information on the Timer operating frequency.

Example:

```
DRV_HANDLE handle;
uint32_t freq;

/* Open the client */
handle = DRV_TMR_Open (DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);

freq = DRV_TMR_OperatingFrequencyGet (handle);
```

Example Usage of the Timer Driver

This section describes typical usage of the Timer Driver for various Timer modules in polling/interrupt advanced/core modes.

Description

The user can pass NULL to the driver initialize interface. However, the respective configuration parameters need to be configured in the correct manner.

Example:

```
//Polled mode under 32-bit count mode for a PIC32 device using the alarm feature
SYS_MODULE_OBJ object;

// main
DRV_TMR_INIT init;
DRV_HANDLE handle;

init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
init.tmrId            = TMR_ID_2;
init.clockSource      = TMR_CLKSOURCE_INTERNAL;
init.prescale         = TMR_PRESCALE_TX_VALUE_256;
init.interruptSource  = INT_SOURCE_TIMER_3;
init.mode             = DRV_TMR_OPERATION_MODE_16_BIT;init.asyncWriteEnable = false;

object = DRV_TMR_Initialize (DRV_TMR_INDEX_0, (SYS_MODULE_INIT *)&init);
if ( SYS_STATUS_READY != DRV_TMR_Status(object))
    return 0;

handle = DRV_TMR_Open (DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if ( DRV_TMR_CLIENT_STATUS_READY != DRV_TMR_ClientStatus(handle))
    return 0;

if(!DRV_TMR_AlarmRegister ( tmrHandle, divider, true, 0, AlarmCallback ))
{
    // divider value could not be obtain;
```

```
    // handle the error
}

DRV_TMR_Start (handle);

while (1)
{
    DRV_TMR_Tasks (object);
}

DRV_TMR_Stop (handle);

DRV_TMR_Close (handle);
if ( DRV_TMR_CLIENT_STATUS_INVALID != DRV_TMR_ClientStatus(handle))
    return 0;

DRV_TMR_Deinitialize (object);
// end main

void AlarmCallback (uintptr_t context, uint32_t alarmCount)
{
  // Do something specific on an alarm event trigger
}
```

Configuring the Library

Macros

	Name	Description
	DRV_TMR_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_TMR_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_TMR_CLOCK_PRESCALER	Sets the default timer driver clock prescaler.
	DRV_TMR_MODE	Sets the default timer driver clock operating mode.
	DRV_TMR_MODULE_ID	Sets the default timer module ID to be used by the timer driver.
	DRV_TMR_MODULE_INIT	Sets the default module init value for the timer driver.
	DRV_TMR_INTERRUPT_SOURCE	Sets the default timer driver clock interrupt source
	DRV_TMR_ASYNC_WRITE_ENABLE	Controls Asynchronous Write mode of the Timer.
	DRV_TMR_CLOCK_SOURCE	Sets the default timer driver clock source.
	DRV_TMR_CLIENTS_NUMBER	Sets up the maximum number of clients that can be supported by an instance of the dynamic driver.

Description

The configuration of the Timer Driver Library is based on the file `system_config.h`.

This header file contains the configuration selection for the Timer Driver Library build. Based on the selections made here and the system setup, the Timer Driver may support the selected features. These configuration settings will apply to all instances of the driver.

This header can be placed anywhere in the application-specific folders and the path of this header needs to be presented to the include search for a successful build. Refer to the Applications Help section for more details.

DRV_TMR_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported by the dynamic driver.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_INSTANCES_NUMBER 5
```

Description

Hardware instances support

This definition sets up the maximum number of hardware instances that can be supported by the dynamic driver.

Remarks

None

DRV_TMR_INTERRUPT_MODE Macro

Controls operation of the driver in the interrupt or polled mode.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_INTERRUPT_MODE true
```

Description

TMR Interrupt And Polled Mode Operation Control

This macro controls the operation of the driver in the interrupt mode of operation. The possible values of this macro are:

- true - Select if interrupt mode of timer operation is desired
- false - Select if polling mode of timer operation is desired

Not defining this option to true or false will result in a build error.

Remarks

None.

DRV_TMR_CLOCK_PRESCALER Macro

Sets the default timer driver clock prescaler.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_CLOCK_PRESCALER (TMR_PRESCALE_VALUE_256)
```

Description

Default timer driver clock prescaler

This macro sets the default timer driver clock prescaler.

Remarks

This value can be overridden by a run time initialization value.

DRV_TMR_MODE Macro

Sets the default timer driver clock operating mode.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_MODE (DRV_TMR_OPERATION_MODE_16_BIT)
```

Description

Default timer driver clock operating mode

This macro sets the default timer driver clock operating mode.

Remarks

This value can be overridden by a run time initialization value.

DRV_TMR_MODULE_ID Macro

Sets the default timer module ID to be used by the timer driver.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_MODULE_ID (TMR_ID_2)
```

Description

Default timer driver index

This macro sets the default timer module ID to be used by the timer driver.

Remarks

This value can be overridden by a run time initialization value.

DRV_TMR_MODULE_INIT Macro

Sets the default module init value for the timer driver.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_MODULE_INIT (SYS_MODULE_POWER_RUN_FULL)
```

Description

Default module init object configuration

This macro sets the default module init value for the timer driver.

Remarks

This value can be overridden by a run time initialization value.

DRV_TMR_INTERRUPT_SOURCE Macro

Sets the default timer driver clock interrupt source

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_INTERRUPT_SOURCE (INT_SOURCE_TIMER_2)
```

Description

Default timer driver clock interrupt source

This macro sets the default timer driver clock interrupt source

Remarks

This value can be overridden by a run time initialization value.

DRV_TMR_ASYNC_WRITE_ENABLE Macro

Controls Asynchronous Write mode of the Timer.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_ASYNC_WRITE_ENABLE false
```

Description

TMR Asynchronous write mode configuration

This macro controls the Asynchronous Write mode of the Timer. This macro accepts the following values:

- true - Configures the Timer to enable asynchronous write control
- false - Configures the Timer to disable asynchronous write control
- DRV_CONFIG_NOT_SUPPORTED - When the feature is not supported on the instance.

Remarks

This feature is not available in all modules/devices. Refer to the specific device data sheet for more information.

DRV_TMR_CLOCK_SOURCE Macro

Sets the default timer driver clock source.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_CLOCK_SOURCE (DRV_TMR_CLKSOURCE_INTERNAL)
```

Description

Default timer driver clock source

This macro sets the default timer driver clock source.

Remarks

This value can be overridden by a run time initialization value.

DRV_TMR_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be supported by an instance of the dynamic driver.

File

drv_tmr_config_template.h

C

```
#define DRV_TMR_CLIENTS_NUMBER 1
```

Description

Client instances support

This definition sets up the maximum number of clients that can be supported by an instance of the dynamic driver.

Remarks

Currently each client is required to get exclusive access to the timer module. Therfore the DRV_TMR_CLIENTS_NUMBER should always be set to 1.

Building the Library

This section lists the files that are available in the Timer Driver Library.

Description

This section list the files that are available in the `\src` folder of the Timer Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/tmr`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_tmr.h	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_tmr_dynamic.c	Basic Timer driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library

Module Dependencies

The Timer Driver Library depends on the following modules:

- Clock System Service Library
- Interrupt System Service Library
- Interrupt Peripheral Library
- Device Control System Service Library

Library Interface

a) System Interaction Functions

	Name	Description
	DRV_TMR_Deinitialize	Deinitializes the specified instance of the Timer driver. **Implementation:** Dynamic
	DRV_TMR_Initialize	Initializes the Timer driver. **Implementation:** Static/Dynamic
	DRV_TMR_Status	Provides the current status of the Timer driver. **Implementation:** Dynamic
	DRV_TMR_Tasks	Maintains the driver's state machine. **Implementation:** Dynamic
	DRV_TMR_ClockSet	Sets the timers clock by selecting the source and prescaler. **Implementation:** Dynamic
	DRV_TMR_GateModeSet	Enables the Gate mode. **Implementation:** Dynamic

b) Core Functions

	Name	Description
	DRV_TMR_ClientStatus	Gets the status of the client operation. **Implementation:** Dynamic
	DRV_TMR_Close	Closes an opened instance of the Timer driver. **Implementation:** Dynamic
	DRV_TMR_Open	Opens the specified Timer driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_TMR_Start	Starts the Timer counting. **Implementation:** Static/Dynamic
	DRV_TMR_Stop	Stops the Timer from counting. **Implementation:** Static/Dynamic

c) Alarm Functions

	Name	Description
	DRV_TMR_AlarmHasElapsed	Provides the status of Timer's period elapse. **Implementation:** Dynamic
	DRV_TMR_AlarmDisable	Disables an alarm signal. **Implementation:** Dynamic
	DRV_TMR_AlarmEnable	Re-enables an alarm signal. **Implementation:** Dynamic
	DRV_TMR_AlarmDeregister	Removes a previously set alarm. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriodGet	Provides the Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriodSet	Updates the Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmRegister	Sets up an alarm. **Implementation:** Dynamic

d) Counter Control Functions

	Name	Description
	DRV_TMR_CounterFrequencyGet	Provides the Timer input frequency. **Implementation:** Dynamic
	DRV_TMR_CounterClear	Clears the Timer's counter register. **Implementation:** Static/Dynamic
	DRV_TMR_CounterValueGet	Reads the Timer's counter register. **Implementation:** Static/Dynamic
	DRV_TMR_CounterValueSet	Updates the Timer's counter register. **Implementation:** Static/Dynamic

e) Miscellaneous Functions

	Name	Description
	DRV_TMR_GateModeClear	Enables the Gate mode. **Implementation:** Dynamic
	DRV_TMR_PrescalerGet	This function gets the currently selected prescaler. **Implementation:** Dynamic
	DRV_TMR_OperationModeGet	This function gets the currently selected operation mode. **Implementation:** Dynamic
	DRV_TMR_DividerRangeGet	Returns the Timer divider values. **Implementation:** Dynamic

f) Data Types and Constants

Name	Description
DRV_TMR_CALLBACK	Pointer to a Timer driver callback function data type.
DRV_TMR_INIT	Defines the Timer driver initialization data.
DRV_TMR_CLIENT_STATUS	Identifies the client-specific status of the Timer driver
DRV_TMR_DIVIDER_RANGE	This data structure specifies the divider values that can be obtained by the timer module.
DRV_TMR_OPERATION_MODE	Lists the operation modes available for timer driver.
DRV_TMR_INDEX_COUNT	Number of valid Timer driver indices.
DRV_TMR_INDEX_0	Timer driver index definitions
DRV_TMR_INDEX_1	This is macro DRV_TMR_INDEX_1.
DRV_TMR_INDEX_2	This is macro DRV_TMR_INDEX_2.
DRV_TMR_INDEX_3	This is macro DRV_TMR_INDEX_3.
DRV_TMR_INDEX_4	This is macro DRV_TMR_INDEX_4.
DRV_TMR_INDEX_5	This is macro DRV_TMR_INDEX_5.
DRV_TMR_INDEX_6	This is macro DRV_TMR_INDEX_6.
DRV_TMR_INDEX_7	This is macro DRV_TMR_INDEX_7.
DRV_TMR_INDEX_8	This is macro DRV_TMR_INDEX_8.

	DRV_TMR_INDEX_9	This is macro DRV_TMR_INDEX_9.
	DRV_TMR_INDEX_10	This is macro DRV_TMR_INDEX_10.
	DRV_TMR_INDEX_11	This is macro DRV_TMR_INDEX_11.

Description

This section describes the functions of the Timer Driver Library.

Refer to each section for a detailed description.

a) System Interaction Functions

DRV_TMR_Deinitialize Function

Deinitializes the specified instance of the Timer driver.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the Timer driver, disabling its operation (and any hardware). All internal data is invalidated.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

This function will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_TMR_Status operation. The system has to use DRV_TMR_Status to find out when the module is in the ready state.

Preconditions

The DRV_TMR_Initialize function must have been called before calling this function and a valid SYS_MODULE_OBJ must have been returned.

Example

```
SYS_MODULE_OBJ      tmrObject;     //  Returned from DRV_TMR_Initialize
SYS_STATUS          tmrStatus;

DRV_TMR_Deinitialize ( tmrObject );

tmrStatus = DRV_TMR_Status ( tmrObject );

if ( SYS_MODULE_UNINITIALIZED == tmrStatus )
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_TMR_Initialize

Function

void DRV_TMR_Deinitialize (SYS_MODULE_OBJ object)

DRV_TMR_Initialize Function

Initializes the Timer driver.

Implementation: Static/Dynamic

File

drv_tmr.h

C

```
SYS_MODULE_OBJ DRV_TMR_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver object. Otherwise, it returns SYS_MODULE_OBJ_INVALID. The returned object must be passed as argument to DRV_TMR_Deinitialize, DRV_TMR_Tasks and DRV_TMR_Status functions.

Description

This function initializes the Timer driver, making it ready for clients to open and use it.

Remarks

This function must be called before any other Timer driver function is called.

This function should only be called once during system initialization unless DRV_TMR_Deinitialize is called to deinitialize the driver instance.

This function will NEVER block for hardware access. The system must use DRV_TMR_Status to find out when the driver is in the ready state.

Build configuration options may be used to statically override options in the "init" structure and will take precedence over initialization data passed using this function.

Preconditions

None.

Example

```
DRV_TMR_INIT    init;
SYS_MODULE_OBJ  objectHandle;

// Populate the timer initialization structure
init.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;
init.tmrId            = TMR_ID_2;
init.clockSource      = DRV_TMR_CLKSOURCE_INTERNAL;
init.prescale         = TMR_PRESCALE_VALUE_256;
init.interruptSource  = INT_SOURCE_TIMER_2;
init.mode             = DRV_TMR_OPERATION_MODE_16_BIT;
init.asyncWriteEnable = false;

// Do something

objectHandle = DRV_TMR_Initialize ( DRV_TMR_INDEX_0, (SYS_MODULE_INIT*)&init );

if ( SYS_MODULE_OBJ_INVALID == objectHandle )
{
    // Handle error
}
```

Parameters

Parameters	Description
drvIndex	Index for the driver instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_TMR_Initialize

(

const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT * const init

)

DRV_TMR_Status Function

Provides the current status of the Timer driver.

Implementation: Dynamic

File

drv_tmr.h

C

```
SYS_STATUS DRV_TMR_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is initialized and ready for operation

Description

This function provides the current status of the Timer driver.

Remarks

Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_STATUS_ERROR - Indicates that the driver is in an error state

Any value less than SYS_STATUS_ERROR is also an error state.

SYS_MODULE_UNINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR.

The this operation can be used to determine when any of the driver's module level operations has completed.

Once the status operation returns SYS_STATUS_READY, the driver is ready for operation.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This function will NEVER block waiting for hardware.

Preconditions

The DRV_TMR_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_TMR_Initialize
SYS_STATUS          tmrStatus;

tmrStatus = DRV_TMR_Status ( object );

else if ( SYS_STATUS_ERROR >= tmrStatus )
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_TMR_Initialize

Function

SYS_STATUS DRV_TMR_Status (SYS_MODULE_OBJ object)

DRV_TMR_Tasks Function

Maintains the driver's state machine.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_Tasks(SYS_MODULE_OBJ object);
```

Returns

None

Description

This function is used to maintain the driver's internal state machine and processes the timer events..

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks)

Preconditions

The DRV_TMR_Initialize function must have been called for the specified Timer driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_TMR_Initialize

while (true)
{
    DRV_TMR_Tasks ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_TMR_Initialize)

Function

void DRV_TMR_Tasks (SYS_MODULE_OBJ object)

DRV_TMR_ClockSet Function

Sets the timers clock by selecting the source and prescaler.

Implementation: Dynamic

File

drv_tmr.h

C

```
bool DRV_TMR_ClockSet(DRV_HANDLE handle, DRV_TMR_CLK_SOURCES clockSource, TMR_PRESCALE preScale);
```

Returns

- true - if the operation is successful
- false - either the handle is invalid or the clockSource and/or prescaler are not supported

Description

This function sets the timer clock by selecting the source and prescaler. The clock sources are device specific, refer device datasheet for supported clock sources. If unsupported clock source is passed then the behaviour of this function is unpredictable.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called. Must have selected 32-Bit timer mode if mode selection is applicable.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE              tmrHandle;  // Returned from DRV_TMR_Open

DRV_TMR_ClockSet ( tmrHandle, DRV_TMR_CLKSOURCE_INTERNAL, TMR_PRESCALE_VALUE_256 );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
clockSource	Clock source of the timer
preScale	Timer's Prescaler divisor

Function

bool DRV_TMR_ClockSet
(
DRV_HANDLE handle,
DRV_TMR_CLK_SOURCES clockSource,
TMR_PRESCALE preScale
)

DRV_TMR_GateModeSet Function

Enables the Gate mode.

Implementation: Dynamic

File

drv_tmr.h

C

```
bool DRV_TMR_GateModeSet(DRV_HANDLE handle);
```

Returns

- true - if the operation is successful
- false - either the handle is invalid or the gate mode is not supported

Description

This function enables the Gated mode of Timer. User can measure the duration of an external signal in this mode. Once the Gate mode is enabled, Timer will start on the raising edge of the external signal. It will keep counting until the next falling edge.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open

DRV_TMR_GateModeSet ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

bool DRV_TMR_GateModeSet (DRV_HANDLE handle)

b) Core Functions

DRV_TMR_ClientStatus Function

Gets the status of the client operation.

Implementation: Dynamic

File

drv_tmr.h

C

```
DRV_TMR_CLIENT_STATUS DRV_TMR_ClientStatus(DRV_HANDLE handle);
```

Returns

None

Description

This function gets the status of the recently completed client level operation.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called for the specified Timer driver instance.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open
DRV_TMR_CLIENT_STATUS tmrDrvStatus;

tmrDrvStatus = DRV_TMR_ClientStatus ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_TMR_CLIENT_STATUS DRV_TMR_ClientStatus (DRV_HANDLE handle)

DRV_TMR_Close Function

Closes an opened instance of the Timer driver.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_Close(DRV_HANDLE handle);
```

Returns

None

Description

This function closes an opened instance of the Timer driver, invalidating the handle.

Remarks

After calling this function, the handle passed in "handle" must not be used with any of the remaining driver functions. A new handle must be obtained by calling DRV_TMR_Open before the caller may use the driver again.

Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_TMR_Initialize function must have been called for the specified Timer driver instance.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_TMR_Open

DRV_TMR_Close ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TMR_Close (DRV_HANDLE handle)

DRV_TMR_Open Function

Opens the specified Timer driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_tmr.h

C

```
DRV_HANDLE DRV_TMR_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Returns

If successful, the function returns a valid open instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID.

Description

This function opens the specified Timer driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. Timer driver does not support multiple clients. If two tasks want to use the timer, one should wait until the other one gets closed.

Remarks

The handle returned is valid until the DRV_TMR_Close function is called.

This function will NEVER block waiting for hardware.

If the requested intent flags are not supported, the function will return DRV_HANDLE_INVALID.

The Timer driver does not support DRV_IO_INTENT_SHARED. Only exclusive access is supported for now.

Preconditions

The DRV_TMR_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_TMR_Open ( DRV_TMR_INDEX_0, DRV_IO_INTENT_EXCLUSIVE );

if ( DRV_HANDLE_INVALID == handle )
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT ORed together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_TMR_Open

(

const SYS_MODULE_INDEX index,

const DRV_IO_INTENT intent

)

DRV_TMR_Start Function

Starts the Timer counting.

Implementation: Static/Dynamic

File

drv_tmr.h

C

```
bool DRV_TMR_Start(DRV_HANDLE handle);
```

Returns

- true - if the operation succeeded
- false - the supplied handle is invalid or the client doesn't have the needed parameters to run (alarm callback and period)

Description

This function starts the Timer counting.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Timer parameters must have been set by a call to DRV_TMR_AlarmRegister.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open

DRV_TMR_AlarmRegister(tmrHandle, 0x100, true, 0, myTmrCallback);
DRV_TMR_Start ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

bool DRV_TMR_Start (DRV_HANDLE handle)

DRV_TMR_Stop Function

Stops the Timer from counting.

Implementation: Static/Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_Stop(DRV_HANDLE handle);
```

Returns

None.

Description

This function stops the running Timer from counting.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_TMR_Open
```

```
DRV_TMR_Stop ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TMR_Stop (DRV_HANDLE handle)

c) Alarm Functions

DRV_TMR_AlarmHasElapsed Function

Provides the status of Timer's period elapse.

Implementation: Dynamic

File

drv_tmr.h

C

```
uint32_t DRV_TMR_AlarmHasElapsed(DRV_HANDLE handle);
```

Returns

Number of times timer has elapsed since the last call.

Description

This function returns the number of times Timer's period has elapsed since last call to this API has made. On calling this API, the internally maintained counter will be cleared and count will be started again from next elapse.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      tmrHandle;  // Returned from DRV_TMR_Open
bool            elapseStatus;
SYS_MODULE_OBJ  tmrObject // Returned by DRV_TMR_Initialize
unsigned int    appInternalTime = 0;

Sys_Tasks()
{
    //Timer task will be called from ISR

    APP_TimeUpdate_Task();

    //Other Tasks
}

void APP_TimeUpdate_Task ( void )
{
    //We will not miss a count even though we are late
    appInternalTime += DRV_TMR_AlarmHasElapsed ( tmrHandle );
}
```

Parameters

Parameters	Description
handle	A valid handle, returned from the DRV_TMR_Open

Function

unsigned int DRV_TMR_AlarmHasElapsed (DRV_HANDLE handle)

DRV_TMR_AlarmDisable Function

Disables an alarm signal.

Implementation: Dynamic

File

drv_tmr.h

C

```
bool DRV_TMR_AlarmDisable(DRV_HANDLE handle);
```

Returns

The current status of the alarm:

- true if the alarm was currently enabled
- false if the alarm was currently disabled

Description

This function allows the client to disable an alarm generation. Use DRV_TMR_AlarmEnable to re-enable.

Remarks

When the driver operates in interrupts this call resolves to a device interrupt disable.

Do NOT disable the timer except for very short periods of time. If the time that the interrupt is disabled is longer than a wrap around period and the interrupt is missed, the hardware has no means of recovering and the resulting timing will be inaccurate.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

A client alarm must be active.

Example

Parameters

Parameters	Description
handle	A valid handle, returned from DRV_TMR_Open

Function

bool DRV_TMR_AlarmDisable (DRV_HANDLE handle);

DRV_TMR_AlarmEnable Function

Re-enables an alarm signal.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_AlarmEnable(DRV_HANDLE handle, bool enable);
```

Returns

None

Description

This function allows the client to re-enable an alarm after it has been disabled by a DRV_TMR_AlarmDisable call.

Remarks

When the driver operates in interrupts this call resolves to a device interrupt re-enable.

Preconditions

The DRV_TMR_Initialize function must have been called. DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

Parameters

Parameters	Description
handle	A valid handle, returned from DRV_TMR_Open
enable	boolean to enable the current callback

Function

void DRV_TMR_AlarmEnable (DRV_HANDLE handle, bool enable);

DRV_TMR_AlarmDeregister Function

Removes a previously set alarm.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_AlarmDeregister(DRV_HANDLE handle);
```

Returns

None.

Description

This function removes a previously set alarm.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

DRV_TMR_AlarmRegister function must have been called before.

Example

```
// Example of a key debounce check

static unsigned int lastReadKey, readKey, keyCount, globalKeyState;
DRV_HANDLE              tmrHandle;  // Returned from DRV_TMR_Open

void keyPressDetect ()
{
    // Calculate the count to be passed on from the clock input
    DRV_TMR_AlarmRegister ( tmrHandle, 0xFF00, true, DebounceCheck );
}

void DebounceCheck ( uintptr_t context )
{
    readKey = AppReadKey();

    if ( readKey != lastReadKey )
    {
        lastReadKey = readKey;
        keyCount = 0;
    }
    else
    {
        if ( keyCount > 20 )
        {
```

```
            globalKeyState = readKey;
            DRV_TMR_AlarmDeregister ( tmrHandle );
        }
        keyCount++;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TMR_AlarmDeregister (DRV_HANDLE handle)

DRV_TMR_AlarmPeriodGet Function

Provides the Timer's period.

Implementation: Dynamic

File

drv_tmr.h

C

```
uint32_t DRV_TMR_AlarmPeriodGet(DRV_HANDLE handle);
```

Returns

Timer period value:

- a 16 bit value if the timer is configured in 16 bit mode
- a 32 bit value if the timer is configured in 32 bit mode

Description

This function gets the Timer's period.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open
uint32_t period;

period = DRV_TMR_AlarmPeriodGet ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_TMR_AlarmPeriodGet (DRV_HANDLE handle)

DRV_TMR_AlarmPeriodSet Function

Updates the Timer's period.

Implementation: Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_AlarmPeriodSet(DRV_HANDLE handle, uint32_t value);
```

Returns

None.

Description

This function updates the Timer's period.

Remarks

- The period value will be truncated to a 16 bit value if the timer is

configured in 16 bit mode.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_TMR_Open

DRV_TMR_AlarmPeriodSet ( handle, 0x1000 );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
value	Period value • a 16 bit value if the timer is configured in 16 bit mode • a 32 bit value if the timer is configured in 32 bit mode

Function

void DRV_TMR_AlarmPeriodSet (DRV_HANDLE handle, uint32_t value)

DRV_TMR_AlarmRegister Function

Sets up an alarm.

Implementation: Dynamic

File

drv_tmr.h

C

```
bool DRV_TMR_AlarmRegister(DRV_HANDLE handle, uint32_t divider, bool isPeriodic, uintptr_t context,
DRV_TMR_CALLBACK callBack);
```

Returns

- true - if the call succeeded
- false - the obtained divider could not be obtained or the passed handle was invalid

Description

This function sets up an alarm, allowing the client to receive a callback from the driver when the timer counter reaches zero. Alarms can be one-shot or periodic. A periodic alarm will reload the timer and generate alarm until stopped. The alarm frequency is: DRV_TMR_CounterFrequencyGet() / divider;

Remarks

The divider value will be truncated to a 16 bit value if the timer is configured in 16 bit mode. The timer should be started using DRV_TMR_Start API to get a callback.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

divider value has to be within the timer divider range (see DRV_TMR_DividerSpecGet).

Example

```
//Do the initialization with 'mode' set to DRV_TMR_OPERATION_MODE_16_BIT

void setupTask ()
{
    DRV_HANDLE              tmrHandle;  // Returned from DRV_TMR_Open

    uint32_t myFreq = 1000; // 1KHz
    uint32_t clkFreq = DRV_TMR_CounterFrequencyGet(tmrHandle);   // timer running frequency

    // calculate the divider needed
    uint32_t divider = clkFreq / myFreq;

    // Start the alarm
    if(!DRV_TMR_AlarmRegister ( tmrHandle, divider, true, 0, CallBackFreq ))
    {
        // divider value could not be obtain;
        // handle the error
        //
    }
}
```

Parameters

Parameters	Description
handle	A valid handle, returned from DRV_TMR_Open
divider	The value to divide the timer clock source to obtain the required alarm frequency. • a 16 bit value if the timer is configured in 16 bit mode • a 32 bit value if the timer is configured in 32 bit mode
isPeriodic	Flag indicating whether the alarm should be one-shot or periodic.
context	A reference, call back function will be called with the same reference.
callBack	A call back function which will be called on time out.

Function

bool DRV_TMR_AlarmRegister
(
    DRV_HANDLE handle,
uint32_t divider,
bool isPeriodic,
uintptr_t context,
    DRV_TMR_CALLBACK callBack
)

d) Counter Control Functions

DRV_TMR_CounterFrequencyGet Function

Provides the Timer input frequency.

Implementation: Dynamic

File

drv_tmr.h

C

```
uint32_t DRV_TMR_CounterFrequencyGet(DRV_HANDLE handle);
```

Returns

32-bit value corresponding to the running frequency. If Timer clock source is external, then this function returns 0.

Description

This function provides the Timer input frequency. Input frequency is the clock to the Timer register and it is considering the prescaler divisor.

Remarks

On most processors, the Timer's base frequency is the same as the peripheral bus clock.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open
uint32_t   clkFreqHz;

clkFreqHz = DRV_TMR_CounterFrequencyGet ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_TMR_CounterFrequencyGet (DRV_HANDLE handle)

DRV_TMR_CounterClear Function

Clears the Timer's counter register.

Implementation: Static/Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_CounterClear(DRV_HANDLE handle);
```

Returns

None.

Description

This function clears the Timer's value in the counter register.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TMR_CounterClear (DRV_HANDLE handle)

DRV_TMR_CounterValueGet Function

Reads the Timer's counter register.

Implementation: Static/Dynamic

File

drv_tmr.h

C

```
uint32_t DRV_TMR_CounterValueGet(DRV_HANDLE handle);
```

Returns

Timer current period:

- a 16 bit value if the timer is configured in 16 bit mode
- a 32 bit value if the timer is configured in 32 bit mode

Description

This function returns the Timer's value in the counter register.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
//Example to use timer for precision time measurement
//without configuring an alarm (interrupt based)
char appState = 0;
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open

switch ( appState )
{
    case 0:
        //Calculate and set the counter period
        DRV_TMR_CounterValueSet ( tmrHandle, ( 0xFFFF - 0x1000 ) );

        //counter starts
        DRV_TMR_Start ( tmrHandle );

        //Trigger an application operation
        app_trigger_operation();

        //Check for time-out in the next state
        appState++;
    case 1:
        //Overflows and stops at 0 if no alarm is set
        if ( DRV_TMR_CounterValueGet ( tmrHandle ) == 0 )
        {
            //Time-out
            return false;
        }
        else if ( app_operation_isComplete( ) )
        {
            //Operation is complete before time-out
            return true;
        }
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint32_t DRV_TMR_CounterValueGet (DRV_HANDLE handle)

DRV_TMR_CounterValueSet Function

Updates the Timer's counter register.

Implementation: Static/Dynamic

File

drv_tmr.h

C

```
void DRV_TMR_CounterValueSet(DRV_HANDLE handle, uint32_t counterPeriod);
```

Returns

None.

Description

This function updates the Timer's value in the counter register.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
counterPeriod	counter period value • a 16 bit value if the timer is configured in 16 bit mode • a 32 bit value if the timer is configured in 32 bit mode

Function

void DRV_TMR_CounterValueSet (DRV_HANDLE handle, uint32_t counterPeriod)

e) Miscellaneous Functions

DRV_TMR_GateModeClear Function

Enables the Gate mode.

Implementation: Dynamic

File

drv_tmr.h

C

```
bool DRV_TMR_GateModeClear(DRV_HANDLE handle);
```

Returns

- true - if the operation is successful
- false - either the handle is invalid or the gate mode is not supported

Description

This function enables the Gated mode of Timer. User can measure the duration of an external signal in this mode. Once the Gate mode is enabled, Timer will start on the raising edge of the external signal. It will keep counting until the next falling edge.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open

DRV_TMR_GateModeClear ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

bool DRV_TMR_GateModeClear (DRV_HANDLE handle)

DRV_TMR_PrescalerGet Function

This function gets the currently selected prescaler.

Implementation: Dynamic

File

drv_tmr.h

C

```
TMR_PRESCALE DRV_TMR_PrescalerGet(DRV_HANDLE handle);
```

Returns

Timer prescaler.

Description

This function gets the currently selected prescaler.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open
TMR_PRESCALE preScale;

preScale = DRV_TMR_PrescalerGet ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

TMR_PRESCALE DRV_TMR_PrescalerGet (DRV_HANDLE handle)

DRV_TMR_OperationModeGet Function

This function gets the currently selected operation mode.

Implementation: Dynamic

File

drv_tmr.h

C

```
DRV_TMR_OPERATION_MODE DRV_TMR_OperationModeGet(DRV_HANDLE handle);
```

Returns

A DRV_TMR_OPERATION_MODE value showing how the timer is currently configured. DRV_TMR_OPERATION_MODE_NONE is returned for an invalid client handle.

Description

This function gets the currently selected 16/32 bit operation mode.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open
DRV_TMR_OPERATION_MODE operMode;

operMode = DRV_TMR_OperationModeGet ( tmrHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_TMR_OPERATION_MODE DRV_TMR_OperationModeGet(DRV_HANDLE handle)

DRV_TMR_DividerRangeGet Function

Returns the Timer divider values.

Implementation: Dynamic

File

drv_tmr.h

C

```
DRV_TMR_OPERATION_MODE DRV_TMR_DividerRangeGet(DRV_HANDLE handle, DRV_TMR_DIVIDER_RANGE* pDivRange);
```

Returns

- A DRV_TMR_OPERATION_MODE value showing how the timer is currently configured. The pDivRange is updated with the supported range values.
- DRV_TMR_OPERATION_MODE_NONE for invalid client handle

Description

This function provides the Timer operating mode and divider range.

Remarks

None.

Preconditions

The DRV_TMR_Initialize function must have been called.

DRV_TMR_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE tmrHandle;  // Returned from DRV_TMR_Open
DRV_TMR_OPERATION_MODE  timerMode;
DRV_TMR_DIVIDER_RANGE   timerRange;
```

```
DRV_TMR_DividerRangeGet(handle, &timerRange);
uint32_t   clkFreqHz = DRV_TMR_CounterFrequencyGet ( tmrHandle );

uint32_t    maxFreqHz = clkFreqHz / timerRange.dividerMin;
uint32_t    minFreqHz = clkFreqHz / timerRange.dividerMax;
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
pDivRange	Address to store the timer divider range.

Function

DRV_TMR_OPERATION_MODE DRV_TMR_DividerRangeGet
(
DRV_HANDLE handle,
DRV_TMR_DIVIDER_RANGE* pDivRange
)

f) Data Types and Constants

DRV_TMR_CALLBACK Type

Pointer to a Timer driver callback function data type.

File

drv_tmr.h

C

```
typedef void (* DRV_TMR_CALLBACK)(uintptr_t context, uint32_t alarmCount);
```

Description

Timer Driver Callback Function Pointer

This data type defines a pointer to a Timer driver callback function.

Remarks

Useful only when timer alarm callback support is enabled by defining the DRV_TMR_ALARM_ENABLE configuration option.

DRV_TMR_INIT Structure

Defines the Timer driver initialization data.

File

drv_tmr.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  TMR_MODULE_ID tmrId;
  DRV_TMR_CLK_SOURCES clockSource;
  TMR_PRESCALE prescale;
  INT_SOURCE interruptSource;
  DRV_TMR_OPERATION_MODE mode;
  bool asyncWriteEnable;
} DRV_TMR_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization.
TMR_MODULE_ID tmrId;	Identifies timer hardware module (PLIB-level) ID
DRV_TMR_CLK_SOURCES clockSource;	Clock Source select.

TMR_PRESCALE prescale;	Prescaler Selection from the processor enumeration
INT_SOURCE interruptSource;	Interrupt Source for TMR module. If 'DRV_TMR_OPERATION_MODE_32_BIT' flag is selected the interrupt will be generated by the 2nd timer of the pair, the odd numbered one.
DRV_TMR_OPERATION_MODE mode;	Select 16/32 bit operation mode. 32 bit mode will combine two 16 bit timer modules to form a 32 bit one. This is usually only necessary for very long delays.
bool asyncWriteEnable;	Asynchronous write enable configuration. If true the asynchronous write is enabled. For timers that do not support this feature the value is ignored

Description

Timer Driver Initialize Data

This data type defines data required to initialize the Timer driver.

Remarks

Not all initialization features are available on all devices.

DRV_TMR_CLIENT_STATUS Enumeration

Identifies the client-specific status of the Timer driver

File

drv_tmr.h

C

```
typedef enum {
  DRV_TMR_CLIENT_STATUS_INVALID,
  DRV_TMR_CLIENT_STATUS_BUSY,
  DRV_TMR_CLIENT_STATUS_READY,
  DRV_TMR_CLIENT_STATUS_RUNNING
} DRV_TMR_CLIENT_STATUS;
```

Members

Members	Description
DRV_TMR_CLIENT_STATUS_INVALID	Driver is invalid (or unopened) state
DRV_TMR_CLIENT_STATUS_BUSY	An operation is currently in progress
DRV_TMR_CLIENT_STATUS_READY	Ready, no operations running
DRV_TMR_CLIENT_STATUS_RUNNING	Timer started and running, processing transactions

Description

Timer Driver Client Status

This enumeration identifies the client-specific status of the Timer driver.

Remarks

None.

DRV_TMR_DIVIDER_RANGE Structure

This data structure specifies the divider values that can be obtained by the timer module.

File

drv_tmr.h

C

```
typedef struct {
  uint32_t dividerMin;
  uint32_t dividerMax;
  uint32_t dividerStep;
} DRV_TMR_DIVIDER_RANGE;
```

Members

Members	Description
uint32_t dividerMin;	The minimum divider value that the timer module can obtain
uint32_t dividerMax;	The maximum divider value that the timer module can obtain

uint32_t dividerStep;	The divider step value, between 2 divider values Should be 1 for most timers

Description

Timer Driver divider operating specification

This data structure specifies the divider values that can be obtained by the timer hardware.

Remarks

None.

DRV_TMR_OPERATION_MODE Enumeration

Lists the operation modes available for timer driver.

File

drv_tmr.h

C

```
typedef enum {
  DRV_TMR_OPERATION_MODE_NONE,
  DRV_TMR_OPERATION_MODE_16_BIT,
  DRV_TMR_OPERATION_MODE_32_BIT
} DRV_TMR_OPERATION_MODE;
```

Members

Members	Description
DRV_TMR_OPERATION_MODE_NONE	The timer module operating mode none/invalid
DRV_TMR_OPERATION_MODE_16_BIT	The timer module operates in 16 bit mode
DRV_TMR_OPERATION_MODE_32_BIT	The timer module operates in 32 bit mode This will combine two 16 bit timer modules

Description

Timer Driver Operation mode

This enumeration lists all the available operation modes that are valid for the timer hardware.

Remarks

Not all modes are available on all devices.

DRV_TMR_INDEX_COUNT Macro

Number of valid Timer driver indices.

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_COUNT TMR_NUMBER_OF_MODULES
```

Description

Timer Driver Module Index Count

This constant identifies Timer driver index definitions.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is device-specific.

DRV_TMR_INDEX_0 Macro

Timer driver index definitions

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_0 0
```

Description

Timer Driver Module Index Numbers

These constants provide Timer driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_TMR_Initialize and DRV_TMR_Open functions to identify the driver instance in use.

DRV_TMR_INDEX_1 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_1 1
```

Description

This is macro DRV_TMR_INDEX_1.

DRV_TMR_INDEX_2 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_2 2
```

Description

This is macro DRV_TMR_INDEX_2.

DRV_TMR_INDEX_3 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_3 3
```

Description

This is macro DRV_TMR_INDEX_3.

DRV_TMR_INDEX_4 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_4 4
```

Description

This is macro DRV_TMR_INDEX_4.

DRV_TMR_INDEX_5 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_5 5
```

Description

This is macro DRV_TMR_INDEX_5.

DRV_TMR_INDEX_6 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_6 6
```

Description

This is macro DRV_TMR_INDEX_6.

DRV_TMR_INDEX_7 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_7 7
```

Description

This is macro DRV_TMR_INDEX_7.

DRV_TMR_INDEX_8 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_8 8
```

Description

This is macro DRV_TMR_INDEX_8.

DRV_TMR_INDEX_9 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_9 9
```

Description

This is macro DRV_TMR_INDEX_9.

DRV_TMR_INDEX_10 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_10 10
```

Description

This is macro DRV_TMR_INDEX_10.

DRV_TMR_INDEX_11 Macro

File

drv_tmr.h

C

```
#define DRV_TMR_INDEX_11 11
```

Description

This is macro DRV_TMR_INDEX_11.

Files

Files

Name	Description
drv_tmr.h	Timer device driver interface header file.
drv_tmr_config_template.h	Timer driver configuration definitions for the template version.

Description

This section lists the source and header files used by the Timer Driver Library.

drv_tmr.h

Timer device driver interface header file.

Enumerations

	Name	Description
	DRV_TMR_CLIENT_STATUS	Identifies the client-specific status of the Timer driver
	DRV_TMR_OPERATION_MODE	Lists the operation modes available for timer driver.

Functions

	Name	Description
	DRV_TMR_AlarmDeregister	Removes a previously set alarm. **Implementation:** Dynamic
	DRV_TMR_AlarmDisable	Disables an alarm signal. **Implementation:** Dynamic
	DRV_TMR_AlarmEnable	Re-enables an alarm signal. **Implementation:** Dynamic
	DRV_TMR_AlarmHasElapsed	Provides the status of Timer's period elapse. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriodGet	Provides the Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmPeriodSet	Updates the Timer's period. **Implementation:** Dynamic
	DRV_TMR_AlarmRegister	Sets up an alarm. **Implementation:** Dynamic

	DRV_TMR_ClientStatus	Gets the status of the client operation. **Implementation:** Dynamic
	DRV_TMR_ClockSet	Sets the timers clock by selecting the source and prescaler. **Implementation:** Dynamic
	DRV_TMR_Close	Closes an opened instance of the Timer driver. **Implementation:** Dynamic
	DRV_TMR_CounterClear	Clears the Timer's counter register. **Implementation:** Static/Dynamic
	DRV_TMR_CounterFrequencyGet	Provides the Timer input frequency. **Implementation:** Dynamic
	DRV_TMR_CounterValueGet	Reads the Timer's counter register. **Implementation:** Static/Dynamic
	DRV_TMR_CounterValueSet	Updates the Timer's counter register. **Implementation:** Static/Dynamic
	DRV_TMR_Deinitialize	Deinitializes the specified instance of the Timer driver. **Implementation:** Dynamic
	DRV_TMR_DividerRangeGet	Returns the Timer divider values. **Implementation:** Dynamic
	DRV_TMR_GateModeClear	Enables the Gate mode. **Implementation:** Dynamic
	DRV_TMR_GateModeSet	Enables the Gate mode. **Implementation:** Dynamic
	DRV_TMR_Initialize	Initializes the Timer driver. **Implementation:** Static/Dynamic
	DRV_TMR_Open	Opens the specified Timer driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_TMR_OperationModeGet	This function gets the currently selected operation mode. **Implementation:** Dynamic
	DRV_TMR_PrescalerGet	This function gets the currently selected prescaler. **Implementation:** Dynamic
	DRV_TMR_Start	Starts the Timer counting. **Implementation:** Static/Dynamic
	DRV_TMR_Status	Provides the current status of the Timer driver. **Implementation:** Dynamic
	DRV_TMR_Stop	Stops the Timer from counting. **Implementation:** Static/Dynamic
	DRV_TMR_Tasks	Maintains the driver's state machine. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_TMR_INDEX_0	Timer driver index definitions
	DRV_TMR_INDEX_1	This is macro DRV_TMR_INDEX_1.
	DRV_TMR_INDEX_10	This is macro DRV_TMR_INDEX_10.
	DRV_TMR_INDEX_11	This is macro DRV_TMR_INDEX_11.
	DRV_TMR_INDEX_2	This is macro DRV_TMR_INDEX_2.
	DRV_TMR_INDEX_3	This is macro DRV_TMR_INDEX_3.
	DRV_TMR_INDEX_4	This is macro DRV_TMR_INDEX_4.
	DRV_TMR_INDEX_5	This is macro DRV_TMR_INDEX_5.
	DRV_TMR_INDEX_6	This is macro DRV_TMR_INDEX_6.
	DRV_TMR_INDEX_7	This is macro DRV_TMR_INDEX_7.
	DRV_TMR_INDEX_8	This is macro DRV_TMR_INDEX_8.
	DRV_TMR_INDEX_9	This is macro DRV_TMR_INDEX_9.
	DRV_TMR_INDEX_COUNT	Number of valid Timer driver indices.

Structures

	Name	Description
	DRV_TMR_DIVIDER_RANGE	This data structure specifies the divider values that can be obtained by the timer module.
	DRV_TMR_INIT	Defines the Timer driver initialization data.

Types

	Name	Description
	DRV_TMR_CALLBACK	Pointer to a Timer driver callback function data type.

Description

Timer Device Driver Interface Definition

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the Timer device driver.

File Name

drv_tmr.h

Company

Microchip Technology Inc.

drv_tmr_config_template.h

Timer driver configuration definitions for the template version.

Macros

	Name	Description
	DRV_TMR_ASYNC_WRITE_ENABLE	Controls Asynchronous Write mode of the Timer.
	DRV_TMR_CLIENTS_NUMBER	Sets up the maximum number of clients that can be supported by an instance of the dynamic driver.
	DRV_TMR_CLOCK_PRESCALER	Sets the default timer driver clock prescaler.
	DRV_TMR_CLOCK_SOURCE	Sets the default timer driver clock source.
	DRV_TMR_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported by the dynamic driver.
	DRV_TMR_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_TMR_INTERRUPT_SOURCE	Sets the default timer driver clock interrupt source
	DRV_TMR_MODE	Sets the default timer driver clock operating mode.
	DRV_TMR_MODULE_ID	Sets the default timer module ID to be used by the timer driver.
	DRV_TMR_MODULE_INIT	Sets the default module init value for the timer driver.

Description

Timer Driver Configuration Definitions for the Template Version

These definitions set up the driver for the default mode of operation of the driver.

File Name

drv_tmr_config_template.h

Company

Microchip Technology Inc.

Touch Driver Libraries Help

This section describes the Touch Driver Libraries.

Generic Touch Driver API

This library help section outlines the generic Touch Driver API to be followed by anyone who wants to use a custom created touch driver to go with the MPLAB Harmony framework for their applications.

Description

This generic driver would still be used with the Touch System Service as described by the API. It provides the data structures and functions required for the touch driver to interface with the graphics library as well as the Touch System Services.

The APIs provide routines to read the touch input data from the touch screen. The driver is based on the device notifying the availability of touch input data through external interrupt.

Currently, the API and the system services only supports non-gestural single-fingered touch input.

Library Interface

Functions

	Name	Description
	DRV_TOUCH_Close	Closes an opened instance of an TOUCH module driver.
	DRV_TOUCH_Deinitialize	Deinitializes the index instance of the TOUCH module.
	DRV_TOUCH_Initialize	Initializes hardware and data for the index instance of the TOUCH module.
	DRV_TOUCH_Open	Opens the specified instance of the Touch driver for use and provides an "open-instance" handle.
	DRV_TOUCH_Read	Notifies the driver that there is current touch data to read
	DRV_TOUCH_Reinitialize	
	DRV_TOUCH_Status	Provides the current status of the index instance of the TOUCH module.
	DRV_TOUCH_Tasks	Maintains the driver's state machine and implements its task queue processing. **Implementation:** Dynamic

Data Types and Constants

	Name	Description
	DRV_TOUCH_INIT	Defines the data required to initialize or reinitialize the TOUCH driver
	DRV_TOUCH_PEN_STATE	Identifies the current state of the pen.
	DRV_TOUCH_POSITION_STATUS	Identifies the current status of the current touch point.
	DRV_TOUCH_SAMPLE_POINTS	This is type DRV_TOUCH_SAMPLE_POINTS.
	DRV_TOUCH_INDEX_0	Touch driver index definitions.
	DRV_TOUCH_INDEX_1	This is macro DRV_TOUCH_INDEX_1.
	DRV_TOUCH_INDEX_COUNT	Number of valid TOUCH driver indices.

Description

Functions

DRV_TOUCH_Close Function

Closes an opened instance of an TOUCH module driver.

File

drv_touch.h

C

```
void DRV_TOUCH_Close(DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened instance of an TOUCH module driver, making the specified handle invalid.

Preconditions

The DRV_TOUCH_Initialize routine must have been called for the specified TOUCH device instance and the DRV_TOUCH_Status must have returned SYS_STATUS_READY.

DRV_TOUCH_Open must have been called to obtain a valid opened device handle.

Example

```
myTouchHandle = DRV_TOUCH_Open(DRV_TOUCH_ID_1, DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE);

DRV_TOUCH_Close(myTouchHandle);
```

Parameters

Parameters	Description
drvHandle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TOUCH_Close (const DRV_HANDLE drvHandle)

DRV_TOUCH_Deinitialize Function

Deinitializes the index instance of the TOUCH module.

File

drv_touch.h

C

```
void DRV_TOUCH_Deinitialize(const SYS_MODULE_INDEX index);
```

Returns

None.

Description

This function deinitializes the index instance of the TOUCH module, disabling its operation (and any hardware for driver modules). It deinitializes only the specified module instance. It also resets all the internal data structures and fields for the specified instance to the default settings.

Preconditions

The DRV_TOUCH_Initialize function should have been called before calling this function.

Example

```
SYS_STATUS   touchstatus;

DRV_TOUCH_Deinitialize(DRV_TOUCH_ID_1);

touchstatus = DRV_TOUCH_Status(DRV_TOUCH_ID_1);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the TOUCH module to be deinitialized

Function

void DRV_TOUCH_Deinitialize (const SYS_MODULE_ID index)

DRV_TOUCH_Initialize Function

Initializes hardware and data for the index instance of the TOUCH module.

File

drv_touch.h

C

```
SYS_MODULE_OBJ DRV_TOUCH_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

None

Description

This function initializes hardware for the index instance of the TOUCH module, using the hardware initialization given data. It also initializes any internal driver data structures making the driver ready to be opened.

Preconditions

None.

Example

```
DRV_TOUCH_INIT_DATA         touchInitData;
SYS_STATUS                  touchStatus;

// Populate the touchInitData structure
touchInitData.moduleInit.powerState = SYS_MODULE_POWER_RUN_FULL;
touchInitData.moduleInit.moduleCode = (DRV_TOUCH_INIT_DATA_MASTER | DRV_TOUCH_INIT_DATA_SLAVE);

DRV_TOUCH_Initialize(DRV_TOUCH_ID_1, (SYS_MODULE_INIT*)&touchInitData);
touchStatus = DRV_TOUCH_Status(DRV_TOUCH_ID_1);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the TOUCH module to be initialized
data	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and the default initialization is to be used.

Function

void DRV_TOUCH_Initialize (const TOUCH_MODULE_ID index,

const SYS_MODULE_INIT *const data)

DRV_TOUCH_Open Function

Opens the specified instance of the Touch driver for use and provides an "open-instance" handle.

File

drv_touch.h

C

```
DRV_HANDLE DRV_TOUCH_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
```

Returns

If successful, the routine returns a valid open-instance handle (a value identifying both the caller and the module instance). If an error occurs, the returned value is DRV_HANDLE_INVALID.

Description

This function opens the specified instance of the Touch module for use and provides a handle that is required to use the remaining driver routines.

This function opens a specified instance of the Touch module driver for use by any client module and provides an "open-instance" handle that must be provided to any of the other Touch driver operations to identify the caller and the instance of the Touch driver/hardware module.

Preconditions

The DRV_TOUCH_Initialize routine must have been called for the specified TOUCH device instance and the DRV_TOUCH_Status must have returned SYS_STATUS_READY.

Example

```
DRV_HANDLE                  touchHandle;
DRV_TOUCH_CLIENT_STATUS     touchClientStatus;

touchHandle = DRV_TOUCH_Open(DRV_TOUCH_ID_1, DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE);
if (DRV_HANDLE_INVALID == touchHandle)
{
    // Handle open error
}
```

```
touchClientStatus = DRV_TOUCH_ClientStatus(touchHandle);

// Close the device when it is no longer needed.
DRV_TOUCH_Close(touchHandle);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the TOUCH module to be opened.
intent	Flags parameter identifying the intended usage and behavior of the driver. Multiple flags may be ORed together to specify the intended usage of the device. See the DRV_IO_INTENT definition.

Function

DRV_HANDLE DRV_TOUCH_Open (const SYS_MODULE_INDEX index,
const DRV_IO_INTENT intent)

DRV_TOUCH_Read Function

Notifies the driver that there is current touch data to read

File

drv_touch.h

C

```
size_t DRV_TOUCH_Read(DRV_HANDLE drvHandle, void * buffer, size_t size);
```

Description

Notifies the driver that there is current touch data to read

Example

Function

size_t DRV_TOUCH_Read (DRV_HANDLE drvHandle, void *buffer, size_t size)

DRV_TOUCH_Reinitialize Function

File

drv_touch.h

C

```
void DRV_TOUCH_Reinitialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT *const data);
```

Returns

None.

Preconditions

The DRV_TOUCH_Initialize function should have been called before calling this function.

Example

```
SYS_MODULE_INIT touchInit;
SYS_STATUS      touchStatus;

DRV_TOUCH_Reinitialize(DRV_TOUCH_ID_1, &touchStatus);
```

Parameters

Parameters	Description
index	Index, identifying the instance of the TOUCH module to be reinitialized
data	Pointer to the data structure containing any data necessary to reinitialize the hardware. This pointer may be null if no data is required and default configuration is to be used.

Function

void DRV_TOUCH_Reinitialize(const SYS_MODULE_ID index,
const SYS_MODULE_INIT *const data)

DRV_TOUCH_Status Function

Provides the current status of the index instance of the TOUCH module.

File

drv_touch.h

C

```
SYS_STATUS DRV_TOUCH_Status(const SYS_MODULE_INDEX index);
```

Description

This function provides the current status of the index instance of the TOUCH module.

Preconditions

The DRV_TOUCH_Initialize function should have been called before calling this function.

Function

SYS_STATUS DRV_TOUCH_Status (const TOUCH_MODULE_ID index)

DRV_TOUCH_Tasks Function

Maintains the driver's state machine and implements its task queue processing.

Implementation: Dynamic

File

drv_touch.h

C

```
void DRV_TOUCH_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal state machine and implement its command queue processing. It is always called from SYS_Tasks() function. This routine decodes the touch input data available.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks)

Preconditions

The DRV_TOUCH_Initialize routine must have been called for the specified MTCH6301 driver instance.

Example

```
SYS_MODULE_OBJ      object;   // Returned from DRV_TOUCH_MTCH6301_Initialize

void SYS_Tasks( void )
{
    DRV_TOUCH_Tasks ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_TOUCH_Initialize)

Function

void DRV_TOUCH_Tasks (SYS_MODULE_OBJ object);

Data Types and Constants

DRV_TOUCH_INIT Structure

Defines the data required to initialize or reinitialize the TOUCH driver

File

drv_touch.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  int touchId;
  SYS_MODULE_OBJ (* drvInitialize)(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
  DRV_HANDLE (* drvOpen)(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
  void (* drvCalibrationSet)(DRV_TOUCH_SAMPLE_POINTS * samplePoints);
  short (* drvTouchGetX)(uint8_t touchNumber);
  short (* drvTouchGetY)(uint8_t touchNumber);
  DRV_TOUCH_POSITION_STATUS (* drvTouchStatus)(const SYS_MODULE_INDEX index);
  void (* drvTouchDataRead)(const SYS_MODULE_INDEX index);
  DRV_TOUCH_PEN_STATE (* drvTouchPenGet)(uint8_t touchNumber);
  INT_SOURCE interruptSource;
  uint16_t orientation;
  uint16_t horizontalResolution;
  uint16_t verticalResolution;
  uint16_t (* pReadFunc)(uint32_t);
  void (* pWriteFunc)(uint16_t, uint32_t);
  void (* pSectorErase)(uint32_t);
  int32_t minTouchDetectDelta;
} DRV_TOUCH_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
int touchId;	ID
uint16_t orientation;	Orientation of the display (given in degrees of 0,90,180,270)
uint16_t horizontalResolution;	Horizontal Resolution of the displayed orientation in Pixels
uint16_t (* pReadFunc)(uint32_t);	typedef for read function pointer
void (* pWriteFunc)(uint16_t, uint32_t);	typedef for write function pointer

Description

TOUCH Driver Initialization Data

This data type defines the data required to initialize or reinitialize the TOUCH driver. If the driver is built statically, the members of this data structure are statically over-ridden by static override definitions in the system_config.h file.

Remarks

None.

DRV_TOUCH_PEN_STATE Type

Identifies the current state of the pen.

File

drv_touch.h

C

```
typedef enum DRV_TOUCH_PEN_STATE@2 DRV_TOUCH_PEN_STATE;
```

Description

TOUCH Controller Driver Pen State

Identifies the current state of the pen reported from a touch event.

Remarks

This enumeration is the return type for the TouchGetPen routine.

DRV_TOUCH_POSITION_STATUS Type

Identifies the current status of the current touch point.

File

drv_touch.h

C

```
typedef enum DRV_TOUCH_POSITION_STATUS@2 DRV_TOUCH_POSITION_STATUS;
```

Description

TOUCH Controller Driver Touch status

Identifies the current status of the current touch point.

Remarks

This enumeration is the return type for the status routine for the current touch point

DRV_TOUCH_SAMPLE_POINTS Type

File

drv_touch.h

C

```
typedef struct DRV_TOUCH_SAMPLE_POINTS@2 DRV_TOUCH_SAMPLE_POINTS;
```

Description

This is type DRV_TOUCH_SAMPLE_POINTS.

DRV_TOUCH_INDEX_0 Macro

Touch driver index definitions.

File

drv_touch.h

C

```
#define DRV_TOUCH_INDEX_0 0
```

Description

Touch Driver Module Index Numbers

These constants provide the Touch driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_TOUCH_Initialize and DRV_TOUCH_Open functions to identify the driver instance in use.

DRV_TOUCH_INDEX_1 Macro

File

drv_touch.h

C

```
#define DRV_TOUCH_INDEX_1 1
```

Description

This is macro DRV_TOUCH_INDEX_1.

DRV_TOUCH_INDEX_COUNT Macro

Number of valid TOUCH driver indices.

File

drv_touch.h

C

```
#define DRV_TOUCH_INDEX_COUNT 1
```

Description

TOUCH Driver Module Index Count

This constant identifies the number of valid TOUCH driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific header files defined as part of the peripheral libraries.

Files

Files

Name	Description
drv_touch.h	Touch device driver interface file.

Description

drv_touch.h

Touch device driver interface file.

Functions

	Name	Description
	DRV_TOUCH_Close	Closes an opened instance of an TOUCH module driver.
	DRV_TOUCH_Deinitialize	Deinitializes the index instance of the TOUCH module.
	DRV_TOUCH_Initialize	Initializes hardware and data for the index instance of the TOUCH module.
	DRV_TOUCH_Open	Opens the specified instance of the Touch driver for use and provides an "open-instance" handle.
	DRV_TOUCH_Read	Notifies the driver that there is current touch data to read
	DRV_TOUCH_Reinitialize	
	DRV_TOUCH_Status	Provides the current status of the index instance of the TOUCH module.
	DRV_TOUCH_Tasks	Maintains the driver's state machine and implements its task queue processing. **Implementation:** Dynamic

Macros

	Name	Description
	DRV_TOUCH_INDEX_0	Touch driver index definitions.
	DRV_TOUCH_INDEX_1	This is macro DRV_TOUCH_INDEX_1.
	DRV_TOUCH_INDEX_COUNT	Number of valid TOUCH driver indices.

Structures

	Name	Description
	DRV_TOUCH_INIT	Defines the data required to initialize or reinitialize the TOUCH driver

Types

	Name	Description
	DRV_TOUCH_PEN_STATE	Identifies the current state of the pen.
	DRV_TOUCH_POSITION_STATUS	Identifies the current status of the current touch point.
	DRV_TOUCH_SAMPLE_POINTS	This is type DRV_TOUCH_SAMPLE_POINTS.

Description

Touch Driver Interface

The Touch driver provides a abstraction to all touch drivers.

File Name

drv_touch.h

Company

Microchip Technology Inc.

10-bit ADC Touch Driver Library

This topic describes the 10-bit ADC Touch Driver Library.

Introduction

This library provides an interface to manage the 10-bit ADC Touch Driver module on the Microchip family of microcontrollers in different modes of operation.

Description

The MPLAB Harmony 10-bit ADC Touch Driver provides a high-level interface to the 10-bit ADC touch device. This driver provides application routines to read non-gestural single-point touch input data from the touch screen. The 10-bit ADC touch device can notify the availability of touch input data through external interrupt. The 10-bit ADC Touch Driver allows the application to map a controller pin as an external interrupt pin.

Using the Library

This topic describes the basic architecture of the 10-bit ADC Touch Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_adc10bit.h`

The interface to the 10-bit ADC Touch Driver library is defined in the `drv_adc10bit.h` header file. Any C language source (`.c`) file that uses the ADC 10-bit Touch Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the 10-bit ADC Touch Driver module.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, open, close, task, and status functions.

Configuring the Library

Macros

	Name	Description
	DRV_ADC10BIT_CALIBRATION_DELAY	Defines the calibration delay.
	DRV_ADC10BIT_CALIBRATION_INSET	Defines the calibration inset.
	DRV_ADC10BIT_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_ADC10BIT_INDEX	ADC10BIT static index selection.
	DRV_ADC10BIT_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_ADC10BIT_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_ADC10BIT_SAMPLE_POINTS	Defines the sample points.
	DRV_ADC10BIT_TOUCH_DIAMETER	Defines the touch diameter.

Description

The configuration of the 10-bit ADC Touch Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the ADC 10-bit Touch Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the 10-bit ADC Touch Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_ADC10BIT_CALIBRATION_DELAY Macro

Defines the calibration delay.

File

drv_adc10bit_config_template.h

C

```
#define DRV_ADC10BIT_CALIBRATION_DELAY 300
```

Description

ADC10BIT Calibration Delay

This macro enables the delay between calibration touch points.

Remarks

None.

DRV_ADC10BIT_CALIBRATION_INSET Macro

Defines the calibration inset.

File

drv_adc10bit_config_template.h

C

```
#define DRV_ADC10BIT_CALIBRATION_INSET 25
```

Description

ADC10BIT Calibration Inset

This macro defines the calibration inset.

Remarks

None.

DRV_ADC10BIT_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_adc10bit_config_template.h

C

```
#define DRV_ADC10BIT_CLIENTS_NUMBER 1
```

Description

ADC10BIT client number

This macro selects the maximum number of clients.

This definition selected the maximum number of clients that the ADC10BIT driver can support at run-time.

Remarks

None.

DRV_ADC10BIT_INDEX Macro

ADC10BIT static index selection.

File

drv_adc10bit_config_template.h

C

```
#define DRV_ADC10BIT_INDEX DRV_ADC10BIT_INDEX_0
```

Description

ADC10BIT Static Index Selection

This macro specifies the static index selection for the driver object reference.

Remarks

This index is required to make a reference to the driver object.

DRV_ADC10BIT_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported.

File

drv_adc10bit_config_template.h

C

```
#define DRV_ADC10BIT_INSTANCES_NUMBER 1
```

Description

ADC10BIT hardware instance configuration

This macro sets up the maximum number of hardware instances that can be supported.

Remarks

None.

DRV_ADC10BIT_INTERRUPT_MODE Macro

Controls operation of the driver in the interrupt or polled mode.

File

drv_adc10bit_config_template.h

C

```
#define DRV_ADC10BIT_INTERRUPT_MODE false
```

Description

ADC10BIT Interrupt And Polled Mode Operation Control

This macro controls the operation of the driver in the interrupt mode of operation. The possible values of this macro are:

- true - Select if interrupt mode of ADC10BIT operation is desired
- false - Select if polling mode of ADC10BIT operation is desired

Not defining this option to true or false will result in a build error.

Remarks

None.

DRV_ADC10BIT_SAMPLE_POINTS Macro

Defines the sample points.

File

drv_adc10bit_config_template.h

C

```
#define DRV_ADC10BIT_SAMPLE_POINTS 4
```

Description

ADC10BIT Sample Points

This macro defines the sample points.

Remarks

None.

DRV_ADC10BIT_TOUCH_DIAMETER Macro

Defines the touch diameter.

File

drv_adc10bit_config_template.h

C

```
#define DRV_ADC10BIT_TOUCH_DIAMETER 10
```

Description

ADC10BIT Touch Diameter

This macro defines the touch diameter.

Remarks

None.

Building the Library

This section lists the files that are available in the 10-bit ADC Touch Driver Library.

Description

This section list the files that are available in the `\src` folder of the 10-bit ADC Touch Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/touch/adc10bit`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_adc10bit.h	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/drv_adc10bit.c	Basic 10-bit ADC Touch Driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The 10-bit ADC Touch Driver Library depends on the following modules:

- Interrupt System Service Library
- Ports System Service Library
- Touch System Service Library
- I2C Driver Library

Library Interface

a) System Functions

	Name	Description
	DRV_TOUCH_ADC10BIT_CalibrationSet	Loads calibration parameters from Non-volatile Memory.
	DRV_TOUCH_ADC10BIT_Close	Closes an opened instance of the 10-bit ADC Driver.
	DRV_TOUCH_ADC10BIT_Deinitialize	Deinitializes the specified instance of the ADC10BIT driver module.
	DRV_TOUCH_ADC10BIT_Initialize	Initializes the 10-bit ADC Driver instance for the specified driver index
	DRV_TOUCH_ADC10BIT_Open	Opens the specified ADC10BIT driver instance and returns a handle to it.
	DRV_TOUCH_ADC10BIT_Status	Provides the current status of the ADC10BIT driver module.
	DRV_TOUCH_ADC10BIT_Tasks	Maintains the driver's state machine and implements its ISR.
	DRV_TOUCH_ADC10BIT_TouchGetRawX	Returns raw x coordinate status when the touch screen is pressed.
	DRV_TOUCH_ADC10BIT_TouchGetRawY	Returns raw y coordinate status when the touch screen is pressed.
	DRV_TOUCH_ADC10BIT_TouchGetX	Returns x coordinate status when the touch screen is pressed.
	DRV_TOUCH_ADC10BIT_TouchStoreCalibration	Stores calibration parameters into Non-volatile Memory.
	DRV_TOUCH_ADC10BIT_PositionDetect	None.
	DRV_TOUCH_ADC10BIT_TouchGetY	Returns y coordinate status when the touch screen is pressed.
	DRV_TOUCH_ADC10BIT_TouchDataRead	Notifies the driver that the current touch data has been read
	DRV_TOUCH_ADC10BIT_TouchStatus	Returns the status of the current touch input.

b) Data Types and Constants

	Name	Description
	_DRV_TOUCH_ADC10BIT_CLIENT_DATA	Defines the data that can be changed per client.
	_DRV_TOUCH_ADC10BIT_INIT	Defines the data required to initialize or reinitialize the 10-bit ADC Driver.
	DRV_ADC10BIT_MODULE_ID	This is type DRV_ADC10BIT_MODULE_ID.
	DRV_TOUCH_ADC10BIT_CLIENT_DATA	Defines the data that can be changed per client.
	DRV_TOUCH_ADC10BIT_HANDLE	Driver handle.
	DRV_TOUCH_ADC10BIT_INIT	Defines the data required to initialize or reinitialize the 10-bit ADC Driver.
	DRV_TOUCH_ADC10BIT_HANDLE_INVALID	Definition of an invalid handle.
	DRV_TOUCH_ADC10BIT_INDEX_0	ADC10BIT driver index definitions.
	DRV_TOUCH_ADC10BIT_INDEX_1	This is macro DRV_TOUCH_ADC10BIT_INDEX_1.
	DRV_TOUCH_ADC10BIT_INDEX_COUNT	Number of valid ADC10BIT driver indices.

Description

This section describes the API functions of the 10-bit ADC Touch Driver library.

Refer to each section for a detailed description.

a) System Functions

DRV_TOUCH_ADC10BIT_CalibrationSet Function

Loads calibration parameters from Non-volatile Memory.

File

drv_adc10bit.h

C

```
void DRV_TOUCH_ADC10BIT_CalibrationSet(DRV_TOUCH_SAMPLE_POINTS * samplePoints);
```

Returns

None.

Description

This function loads calibration parameters from Non-volatile Memory.

Preconditions

The NVM initialization function must be called before calling this function.

Function

void DRV_TOUCH_ADC10BIT_TouchLoadCalibration(void)

DRV_TOUCH_ADC10BIT_Close Function

Closes an opened instance of the 10-bit ADC Driver.

File

drv_adc10bit.h

C

```
void DRV_TOUCH_ADC10BIT_Close(DRV_HANDLE handle);
```

Returns

None

Description

This function closes an opened instance of the 10-bit ADC Driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_TOUCH_ADC10BIT_Open before the caller may use the driver again. This function is thread safe in a RTOS application.

Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

DRV_TOUCH_ADC10BIT_Initialize must have been called for the specified ADC10BIT driver instance.

DRV_TOUCH_ADC10BIT_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_TOUCH_ADC10BIT_Open

DRV_TOUCH_ADC10BIT_Close ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TOUCH_ADC10BIT_Close (DRV_HANDLE handle)

DRV_TOUCH_ADC10BIT_Deinitialize Function

Deinitializes the specified instance of the ADC10BIT driver module.

File

drv_adc10bit.h

C

```
void DRV_TOUCH_ADC10BIT_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the specified instance of the 10-bit ADC Driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

This function will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_TOUCH_ADC10BIT_Status operation. The system has to use DRV_TOUCH_ADC10BIT_Status to determine when the module is in the ready state.

Preconditions

DRV_TOUCH_ADC10BIT_Initialize must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_TOUCH_ADC10BIT_Initialize
SYS_STATUS          status;

DRV_TOUCH_ADC10BIT_Deinitialize ( object );

status = DRV_TOUCH_ADC10BIT_Status( object );
if( SYS_MODULE_UNINITIALIZED == status )
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_TOUCH_ADC10BIT_Initialize

Function

void DRV_TOUCH_ADC10BIT_Deinitialize (SYS_MODULE_OBJ object)

DRV_TOUCH_ADC10BIT_Initialize Function

Initializes the 10-bit ADC Driver instance for the specified driver index

File

drv_adc10bit.h

C

```
SYS_MODULE_OBJ DRV_TOUCH_ADC10BIT_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const
init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes the 10-bit ADC Driver instance for the specified driver index, making it ready for clients to open and use it. The initialization

data is specified by the 'init' parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the 10-bit ADC Driver module ID. For example, driver instance 0 can be assigned to ADC10BIT2. If the driver is built statically, then some of the initialization parameters are overridden by configuration macros. Refer to the description of the DRV_TOUCH_ADC10BIT_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This routine must be called before any other ADC10BIT routine is called.

This routine should only be called once during system initialization unless DRV_TOUCH_ADC10BIT_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

None.

Example

```
DRV_TOUCH_ADC10BIT_INIT        init;
SYS_MODULE_OBJ      objectHandle;

// Populate the ADC10BIT initialization structure
init.spiId              = ADC10BIT_ID_1;

objectHandle = DRV_TOUCH_ADC10BIT_Initialize(DRV_TOUCH_ADC10BIT_INDEX_1, (SYS_MODULE_INIT*)usartInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized. Please note this is not the 10-bit ADC Driver ID. The hardware 10-bit ADC Driver ID is set in the initialization structure. This is the index of the driver index to use.
init	Pointer to a data structure containing any data necessary to initialize the driver. If this pointer is NULL, the driver uses the static initialization override macros for each member of the initialization data structure.

Function

SYS_MODULE_OBJ DRV_TOUCH_ADC10BIT_Initialize(const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init)

DRV_TOUCH_ADC10BIT_Open Function

Opens the specified ADC10BIT driver instance and returns a handle to it.

File

drv_adc10bit.h

C

```
DRV_HANDLE DRV_TOUCH_ADC10BIT_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. An error can occur when the following is true:

- if the number of client objects allocated via DRV_TOUCH_ADC10BIT_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid

Description

This function opens the specified USART driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options additionally affect the behavior of the DRV_USART_Read() and DRV_USART_Write() functions. If the ioIntent is DRV_IO_INTENT_NONBLOCKING, then these function will not block even if the required amount of data could not be processed. If the ioIntent is DRV_IO_INTENT_BLOCKING, these functions will block until the required amount of data is processed.

If ioIntent is DRV_IO_INTENT_READ, the client will only be read from the driver. If ioIntent is DRV_IO_INTENT_WRITE, the client will only be able to write to the driver. If the ioIntent in DRV_IO_INTENT_READWRITE, the client will be able to do both, read and write.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_TOUCH_ADC10BIT_Close routine is called. This routine will NEVER block waiting for hardware. If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

DRV_TOUCH_ADC10BIT_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_TOUCH_ADC10BIT_Open( DRV_TOUCH_ADC10BIT_INDEX_0, DRV_IO_INTENT_EXCLUSIVE );

if( DRV_HANDLE_INVALID == handle )
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Index of the driver initialized with DRV_TOUCH_ADC10BIT_Initialize. Please note this is not the SPI id.
intent	Zero or more of the values from the enumeration DRV_IO_INTENT ORed together to indicate the intended use of the driver

Function

DRV_HANDLE DRV_TOUCH_ADC10BIT_Open (const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT intent)

DRV_TOUCH_ADC10BIT_Status Function

Provides the current status of the ADC10BIT driver module.

File

drv_adc10bit.h

C

```
SYS_STATUS DRV_TOUCH_ADC10BIT_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system level operation and cannot start another

Description

This function provides the current status of the ADC10BIT driver module.

Remarks

Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_MODULE_UNINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR.

This function can be used to determine when any of the driver's module level operations has completed.

If the status operation returns SYS_STATUS_BUSY, the previous operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This function will NEVER block waiting for hardware.

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

DRV_TOUCH_ADC10BIT_Initialize must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_TOUCH_ADC10BIT_Initialize
SYS_STATUS          status;

status = DRV_TOUCH_ADC10BIT_Status( object );
if( SYS_STATUS_READY != status )
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_TOUCH_ADC10BIT_Initialize

Function

SYS_STATUS DRV_TOUCH_ADC10BIT_Status (SYS_MODULE_OBJ object)

DRV_TOUCH_ADC10BIT_Tasks Function

Maintains the driver's state machine and implements its ISR.

File

drv_adc10bit.h

C

```
void DRV_TOUCH_ADC10BIT_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal state machine and implement its transmit ISR for interrupt-driven implementations. In polling mode, this function should be called from the SYS_Tasks function. In Interrupt mode, this function should be called in the transmit interrupt service routine of the USART that is associated with this USART driver hardware instance.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This function may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

DRV_TOUCH_ADC10BIT_Initialize must have been called for the specified 10-bit ADC Driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_TOUCH_ADC10BIT_Initialize

while( true )
{
    DRV_TOUCH_ADC10BIT_Tasks ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_TOUCH_ADC10BIT_Initialize)

Function

void DRV_TOUCH_ADC10BIT_Tasks (SYS_MODULE_OBJ object);

DRV_TOUCH_ADC10BIT_TouchGetRawX Function

Returns raw x coordinate status when the touch screen is pressed.

File

drv_adc10bit.h

C

```
short DRV_TOUCH_ADC10BIT_TouchGetRawX();
```

Returns

- raw x coordinate - Indicates the touch screen was pressed
- -1 - Indicates the touch screen was not pressed

Description

This function returns the raw x coordinate status when the touch screen is pressed.

Remarks

None.

Preconditions

None.

Function

short DRV_TOUCH_ADC10BIT_TouchGetRawX()

DRV_TOUCH_ADC10BIT_TouchGetRawY Function

Returns raw y coordinate status when the touch screen is pressed.

File

drv_adc10bit.h

C

```
short DRV_TOUCH_ADC10BIT_TouchGetRawY();
```

Returns

- raw y coordinate - Indicates the touch screen was pressed
- -1 - Indicates the touch screen was not pressed

Description

This function returns the raw y coordinate status when the touch screen is pressed.

Remarks

None.

Preconditions

None.

Function

short DRV_TOUCH_ADC10BIT_TouchGetRawY()

DRV_TOUCH_ADC10BIT_TouchGetX Function

Returns x coordinate status when the touch screen is pressed.

File

drv_adc10bit.h

C

```
short DRV_TOUCH_ADC10BIT_TouchGetX(uint8_t touchNumber);
```

Returns

- x coordinate - Indicates the touch screen was pressed
- -1 - Indicates the touch screen was not pressed

Description

This function returns the x coordinate status when the touch screen is pressed.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
touchNumber	touch input index.

Function

short DRV_TOUCH_ADC10BIT_TouchGetX(uint8_t touchNumber)

DRV_TOUCH_ADC10BIT_TouchStoreCalibration Function

Stores calibration parameters into Non-volatile Memory.

File

drv_adc10bit.h

C

```
void DRV_TOUCH_ADC10BIT_TouchStoreCalibration();
```

Returns

None.

Description

This function stores calibration parameters into Non-volatile Memory.

Remarks

This API is deprecated and its funcationality is handled via SYSTEM_INITIALIZATION

Preconditions

The NVM initialization function must be called before calling this function.

Function

void DRV_TOUCH_ADC10BIT_TouchStoreCalibration(void)

DRV_TOUCH_ADC10BIT_PositionDetect Function

None.

File

drv_adc10bit.h

C

```
short DRV_TOUCH_ADC10BIT_PositionDetect();
```

Returns

None.

Description

None.

Preconditions

None.

Function

void DRV_TOUCH_ADC10BIT_TouchLoadCalibration(void)

DRV_TOUCH_ADC10BIT_TouchGetY Function

Returns y coordinate status when the touch screen is pressed.

File

drv_adc10bit.h

C

```
short DRV_TOUCH_ADC10BIT_TouchGetY(uint8_t touchNumber);
```

Returns

- y coordinate - Indicates the touch screen was pressed
- -1 - Indicates the touch screen was not pressed

Description

This function returns the y coordinate status when the touch screen is pressed.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
handle	driver client handle.
touchNumber	touch input index.

Function

short DRV_TOUCH_ADC10BIT_TouchGetY(DRV_HANDLE handle, uint8_t touchNumber)

DRV_TOUCH_ADC10BIT_TouchDataRead Function

Notifies the driver that the current touch data has been read

File

drv_adc10bit.h

C

```
void DRV_TOUCH_ADC10BIT_TouchDataRead(const SYS_MODULE_INDEX index);
```

Returns

None.

Description

Notifies the driver that the current touch data has been read

Function

void DRV_TOUCH_ADC10BIT_TouchDataRead(const SYS_MODULE_INDEX index)

DRV_TOUCH_ADC10BIT_TouchStatus Function

Returns the status of the current touch input.

File

drv_adc10bit.h

C

```
DRV_TOUCH_POSITION_STATUS DRV_TOUCH_ADC10BIT_TouchStatus(const SYS_MODULE_INDEX index);
```

Returns

It returns the status of the current touch input.

Description

It returns the status of the current touch input.

Function

DRV_TOUCH_POSITION_SINGLE DRV_TOUCH_ADC10BIT_TouchStatus(const SYS_MODULE_INDEX index)

b) Data Types and Constants

DRV_ADC10BIT_MODULE_ID Enumeration

File

drv_adc10bit.h

C

```
typedef enum {
  ADC10BIT_ID_1 = 0,
  ADC10BIT_NUMBER_OF_MODULES
} DRV_ADC10BIT_MODULE_ID;
```

Description

This is type DRV_ADC10BIT_MODULE_ID.

DRV_TOUCH_ADC10BIT_CLIENT_DATA Structure

Defines the data that can be changed per client.

File

drv_adc10bit.h

C

```
typedef struct _DRV_TOUCH_ADC10BIT_CLIENT_DATA {
} DRV_TOUCH_ADC10BIT_CLIENT_DATA;
```

Description

Macro: ADC10BIT Driver Client Specific Configuration

This data type defines the data can be configured per client. This data can be per client, and overrides the configuration data contained inside of DRV_TOUCH_ADC10BIT_INIT.

Remarks

None.

DRV_TOUCH_ADC10BIT_HANDLE Type

Driver handle.

File

drv_adc10bit.h

C

```
typedef uintptr_t DRV_TOUCH_ADC10BIT_HANDLE;
```

Description

Macro: ADC10BIT Driver Handle

Touch screen controller interfacing with the 10-bit Analog-to-Digital (ADC) converter device.

Remarks

None

DRV_TOUCH_ADC10BIT_INIT Structure

Defines the data required to initialize or reinitialize the 10-bit ADC Driver.

File

drv_adc10bit.h

C

```
typedef struct _DRV_TOUCH_ADC10BIT_INIT {
  SYS_MODULE_INIT moduleInit;
  DRV_ADC10BIT_MODULE_ID adc10bitId;
} DRV_TOUCH_ADC10BIT_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
DRV_ADC10BIT_MODULE_ID adc10bitId;	Identifies peripheral (PLIB-level) ID

Description

Macro: ADC10BIT Driver Initialization Data

This data type defines the data required to initialize or reinitialize the 10-bit ADC Driver. If the driver is built statically, the members of this data structure are statically over-ridden by static override definitions in the system_config.h file.

Remarks

None.

DRV_TOUCH_ADC10BIT_HANDLE_INVALID Macro

Definition of an invalid handle.

File

drv_adc10bit.h

C

```
#define DRV_TOUCH_ADC10BIT_HANDLE_INVALID ((DRV_TOUCH_ADC10BIT_HANDLE)(-1))
```

Description

Macro: ADC10BIT Driver Invalid Handle

This is the definition of an invalid handle. An invalid handle is returned by DRV_ADC10BIT_RawRead and DRV_ADC10BIT_RawRead functions if the request was not successful.

Remarks

None.

DRV_TOUCH_ADC10BIT_INDEX_0 Macro

ADC10BIT driver index definitions.

File

drv_adc10bit.h

C

```
#define DRV_TOUCH_ADC10BIT_INDEX_0 0
```

Description

Macro: ADC10BIT Driver Module Index Numbers

These constants provide the 10-bit ADC Driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the DRV_ADC10BIT_Initialize and DRV_ADC10BIT_Open functions to identify the driver instance in use.

DRV_TOUCH_ADC10BIT_INDEX_1 Macro

File

drv_adc10bit.h

C

```
#define DRV_TOUCH_ADC10BIT_INDEX_1 1
```

Description

This is macro DRV_TOUCH_ADC10BIT_INDEX_1.

DRV_TOUCH_ADC10BIT_INDEX_COUNT Macro

Number of valid ADC10BIT driver indices.

File

drv_adc10bit.h

C

```
#define DRV_TOUCH_ADC10BIT_INDEX_COUNT 2
```

Description

Macro: ADC10BIT Driver Module Index Count

This constant identifies the number of valid 10-bit ADC Driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific header files defined as part of the peripheral libraries.

Files

Files

Name	Description
drv_adc10bit.h	10-bit ADC Touch Driver interface definitions
drv_adc10bit_config_template.h	10-bit ADC Touch Driver configuration template.

Description

This section lists the source and header files used by the 10-bit ADC Touch Driver Library.

drv_adc10bit.h

10-bit ADC Touch Driver interface definitions

Enumerations

	Name	Description
	DRV_ADC10BIT_MODULE_ID	This is type DRV_ADC10BIT_MODULE_ID.

Functions

	Name	Description
	DRV_TOUCH_ADC10BIT_CalibrationSet	Loads calibration parameters from Non-volatile Memory.
	DRV_TOUCH_ADC10BIT_Close	Closes an opened instance of the 10-bit ADC Driver.
	DRV_TOUCH_ADC10BIT_Deinitialize	Deinitializes the specified instance of the ADC10BIT driver module.
	DRV_TOUCH_ADC10BIT_Initialize	Initializes the 10-bit ADC Driver instance for the specified driver index
	DRV_TOUCH_ADC10BIT_Open	Opens the specified ADC10BIT driver instance and returns a handle to it.
	DRV_TOUCH_ADC10BIT_PositionDetect	None.
	DRV_TOUCH_ADC10BIT_Status	Provides the current status of the ADC10BIT driver module.
	DRV_TOUCH_ADC10BIT_Tasks	Maintains the driver's state machine and implements its ISR.
	DRV_TOUCH_ADC10BIT_TouchDataRead	Notifies the driver that the current touch data has been read
	DRV_TOUCH_ADC10BIT_TouchGetRawX	Returns raw x coordinate status when the touch screen is pressed.
	DRV_TOUCH_ADC10BIT_TouchGetRawY	Returns raw y coordinate status when the touch screen is pressed.
	DRV_TOUCH_ADC10BIT_TouchGetX	Returns x coordinate status when the touch screen is pressed.
	DRV_TOUCH_ADC10BIT_TouchGetY	Returns y coordinate status when the touch screen is pressed.
	DRV_TOUCH_ADC10BIT_TouchStatus	Returns the status of the current touch input.
	DRV_TOUCH_ADC10BIT_TouchStoreCalibration	Stores calibration parameters into Non-volatile Memory.

Macros

	Name	Description
	DRV_TOUCH_ADC10BIT_HANDLE_INVALID	Definition of an invalid handle.
	DRV_TOUCH_ADC10BIT_INDEX_0	ADC10BIT driver index definitions.
	DRV_TOUCH_ADC10BIT_INDEX_1	This is macro DRV_TOUCH_ADC10BIT_INDEX_1.
	DRV_TOUCH_ADC10BIT_INDEX_COUNT	Number of valid ADC10BIT driver indices.

Structures

	Name	Description
	_DRV_TOUCH_ADC10BIT_CLIENT_DATA	Defines the data that can be changed per client.
	_DRV_TOUCH_ADC10BIT_INIT	Defines the data required to initialize or reinitialize the 10-bit ADC Driver.
	DRV_TOUCH_ADC10BIT_CLIENT_DATA	Defines the data that can be changed per client.
	DRV_TOUCH_ADC10BIT_INIT	Defines the data required to initialize or reinitialize the 10-bit ADC Driver.

Types

	Name	Description
	DRV_TOUCH_ADC10BIT_HANDLE	Driver handle.

Description

10-bit ADC Touch Driver Interface Definition

This is a resistive touch screen driver that is using the Microchip Graphics Library. The calibration values are automatically checked (by reading a specific memory location on the non-volatile memory) when initializing the module if the function pointers to the read and write callback functions are initialized. If the read value is invalid calibration will automatically be executed. Otherwise, the calibration values will be loaded and used. The driver assumes that the application side provides the read and write routines to a non-volatile memory. If the callback functions are not initialized, the calibration routine will always be called at start-up to initialize the global calibration values. This driver assumes that the Graphics Library is initialized and will be using the default font of the library.

File Name

drv_adc10bit.h

Company

Microchip Technology Inc.

drv_adc10bit_config_template.h

10-bit ADC Touch Driver configuration template.

Macros

	Name	Description
	DRV_ADC10BIT_CALIBRATION_DELAY	Defines the calibration delay.
	DRV_ADC10BIT_CALIBRATION_INSET	Defines the calibration inset.
	DRV_ADC10BIT_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_ADC10BIT_INDEX	ADC10BIT static index selection.
	DRV_ADC10BIT_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_ADC10BIT_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_ADC10BIT_SAMPLE_POINTS	Defines the sample points.
	DRV_ADC10BIT_TOUCH_DIAMETER	Defines the touch diameter.

Description

10-bit ADC Touch Driver Configuration Template

This header file contains the build-time configuration selections for the 10-bit ADC Touch Driver. This is the template file which give all possible configurations that can be made. This file should not be included in any project.

File Name

drv_adc10bit_config_template.h

Company

Microchip Technology Inc.

ADC Touch Driver Library

This topic describes the ADC Touch Driver Library.

Introduction

This library provides an interface to manage the ADC Touch Driver module on the Microchip family of microcontrollers in different modes of operation.

Description

The MPLAB Harmony ADC Touch Driver provides a high-level interface to the ADC touch device. This driver provides application routines to read the touch input data from the touch screen. . The ADC touch device can notify the availability of touch input data through external interrupt. The ADC Touch Driver allows the application to map a controller pin as an external interrupt pin.

Using the Library

This topic describes the basic architecture of the ADC Touch Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_touch_adc.h`

The interface to the ADC Touch Driver library is defined in the `drv_touch_adc.h` header file. Any C language source (`.c`) file that uses the ADC Touch Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

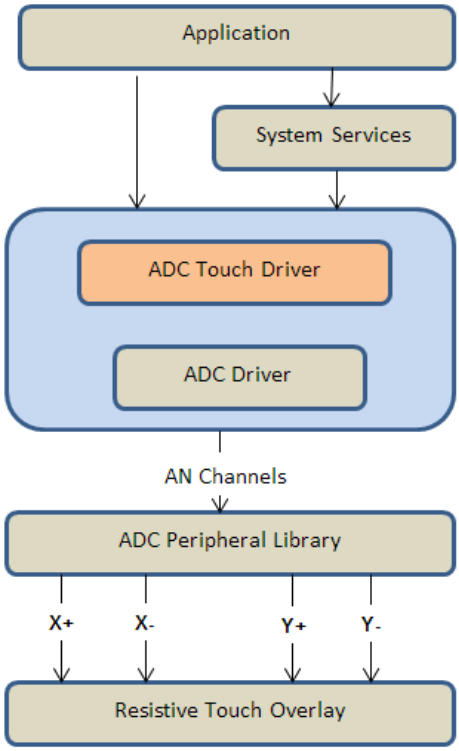
This library provides a low-level abstraction of the ADC Touch Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The ADC Touch Driver has routines to perform the following operations. The driver initialization routines allow the application to initialize the driver. The driver must be initialized before it can be used by application. Once driver is initialized the driver open routine allows retrieving the client handle. Once the touch input is available a touch input read request is sent and input data is retrieved in a buffer. The buffer data is then decoded

to get the x and y coordinate of the touch screen in the form of the number of pixels.

ADC Touch Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the ADC Touch Driver module.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, open, close, task, and status functions.

How the Library Works

The library provides interfaces to support system functions, which provide system module interfaces, device initialization, deinitialization, open, close, task, and status functions.

During steady state operation, the DRV_TOUCH_ADC_Tasks is called continuously in System_Tasks to save the current touch position, or -1 if no touch was detected.

At any time, DRV_TOUCH_ADC_TouchGetX and DRV_TOUCH_ADC_TouchGetY are called to retrieve the last touch position. Touch positions are not queued.

Touch samples are configurable, the default is 300. The return integer can have the value between 0-screenWidth and 0-screenHeight.

Initializing the Driver

Before the ADC Touch Driver can be opened, it must be configured and initialized. The driver build time configuration is defined by the configuration macros. Refer to the Building the Library section for the location of and more information on the various configuration macros and how these macros should be designed. The driver initialization is configured through the DRV_TOUCH_INIT data structure that is passed to the DRV_TOUCH_ADC_Initialize function. The initialization parameters include the interrupt source, interrupt pin remap configuration and touch screen resolution. The following code shows an example of initializing the ADC Touch Touch Driver.

```
/* The following code shows an example of designing the
 *  DRV_TOUCH_INIT data structure. It also shows how an example
 *  usage of the DRV_TOUCH_ADC_Initialize function.
 */
```

```
DRV_TOUCH_INIT drvTouchInitData;
SYS_MODULE_OBJ objectHandle;

/* Touch Module Id*/
drvTouchInitData.touchId                    = DRV_TOUCH_INDEX_0;

/* I2C Bus driver open */
drvTouchInitData.drvOpen                    = DRV_I2C_Open;

/* Interrupt Source for Touch */
drvTouchInitData.interruptSource            = INT_SOURCE_EXTERNAL_1;

/* Interrupt Pin function mapping */
drvTouchInitData.interruptPort.inputFunction = INPUT_FUNC_INT1;

/* Pin to be mapped as interrupt pin */
drvTouchInitData.interruptPort.inputPin     = INPUT_PIN_RPE8;

/* Analog pin number */
drvTouchInitData.interruptPort.analogPin    = PORTS_ANALOG_PIN_25;

/* Pin Mode of analog pin */
drvTouchInitData.interruptPort.pinMode      = PORTS_PIN_MODE_DIGITAL;

/* Interrupt pin port */
drvTouchInitData.interruptPort.channel      = PORT_CHANNEL_E;

/* Interrupt pin port maskl */
drvTouchInitData.interruptPort.dataMask     = 0x8;

/* Touch screen orientation */
drvTouchInitData.orientation                = DISP_ORIENTATION;

/* Touch screen horizontal resolution */
drvTouchInitData.horizontalResolution           = DISP_HOR_RESOLUTION;

/* Touch screen vertical resolution */
drvTouchInitData.verticalResolution         = DISP_VER_RESOLUTION;

/* Driver initialization */
objectHandle = DRV_TOUCH_ADC_Initialize(DRV_TOUCH_INDEX_0,
                                        (SYS_MODULE_INIT*)drvTouchInitData);
    if (SYS_MODULE_OBJ_INVALID == objectHandle)
    {
        // Handle error
    }
```

Opening the Driver

To use the ADC Touch Driver, the application must open the driver. This is done by calling the DRV_TOUCH_ADC_Open function. If successful, the DRV_TOUCH_ADC_Open function will return a handle to the driver. This handle records the association between the client and the driver instance that was opened. The DRV_TOUCH_ADC_Open function may return DRV_HANDLE_INVALID in the situation where the driver is not ready to be opened. When this occurs, the application can try opening the driver again. Note that the open function may return an invalid handle in other (error) cases as well. The following code shows an example of the driver being opened.

```
DRV_HANDLE handle;
handle = DRV_TOUCH_ADC_Open( DRV_TOUCH_ADC_INDEX_0,
DRV_IO_INTENT_EXCLUSIVE );
if( DRV_HANDLE_INVALID == handle )
{
// Unable
}
```

Tasks Routine

To use the ADC Touch Driver, the application must open the driver. This is done by calling the DRV_TOUCH_ADC_Open function. If successful, the DRV_TOUCH_ADC_Open function will return a handle to the driver. This handle records the association between the client and the driver instance that was opened. The DRV_TOUCH_ADC_Open function may return DRV_HANDLE_INVALID in the situation where the driver is not ready to be opened. When this occurs, the application can try opening the driver again. Note that the open function may return an invalid handle in

other (error) cases as well. The following code shows an example of the driver being opened.

```
DRV_HANDLE handle;
handle = DRV_TOUCH_ADC_Open( DRV_TOUCH_ADC_INDEX_0,
DRV_IO_INTENT_EXCLUSIVE );
if( DRV_HANDLE_INVALID == handle )
{
// Unable
}
```

Configuring the Library

The configuration of the ADC Touch Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the ADC 10-bit Touch Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the ADC Touch Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Building the Library

This section lists the files that are available in the ADC Touch Driver Library.

Description

This section list the files that are available in the `\src` folder of the ADC Touch Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/touch/touch_adc`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_touch_adc.h`	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/drv_touch_adc.c`	Basic ADC Touch Driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The ADC Touch Driver Library depends on the following modules:

- Interrupt System Service Library
- Ports System Service Library
- Touch System Service Library
- I2C Driver Library

Library Interface

This section describes the API functions of the ADC Touch Driver library.

Refer to each section for a detailed description.

a) System Functions

b) Data Types and Constants

Files

This section lists the source and header files used by the ADC Touch Driver Library.

AR1021 Touch Driver Library

This topic describes the AR1021 Touch Driver Library.

Introduction

This library provides a low-level abstraction of the AR1021 Touch Driver Library that is available on the Microchip family of microcontrollers with a convenient C language interface. It can be used to simplify low-level access to the module without the necessity of interacting directly with the module's registers, thereby hiding differences from one microcontroller variant to another.

Description

The AR1021 Touch Driver Library, in conjunction with the Microchip AR1021 Resistive Touch Screen Controller module, allows an application to:

- Calibrate touch points
- Receive touch points

The following application services are provided by the AR1021 Touch Driver Library:

- Configuring the AR1021 controller (TouchThreshold, PenUpDelay, PenStateReportDelaylist, SensitivityFilter, etc.)
- Saving touch points to EEPROM

The operational services are not typically accessible to the application as this portion of the code resides within the Touch System Service Library software layer and is used by the Graphics Library stack services to receive touch point data.

Using the Library

This topic describes the basic architecture of the AR1021 Touch Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_ar1021.h`

The interface to the AR1021 Touch Driver library is defined in the `drv_ar1021.h` header file. Any C language source (`.c`) file that uses the AR1021 Touch Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the AR1021 Touch Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

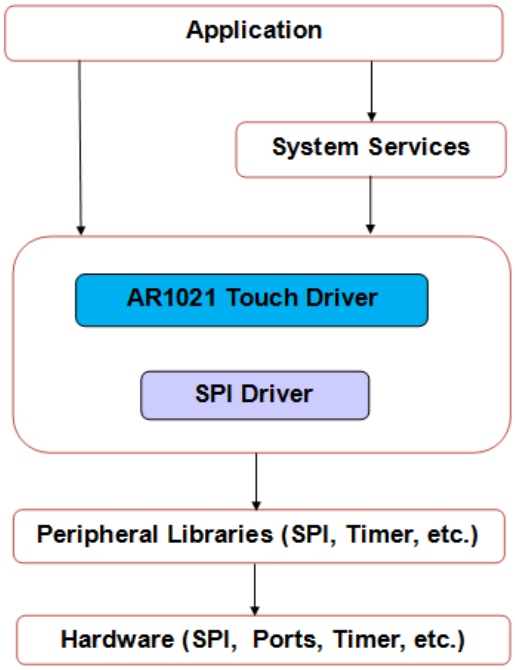
Description

The AR1021 Touch Driver Library provides the following functionality:

- AR1021 library initialization
- AR1021 controller configuration
- AR1021 controller connectivity
- AR1021 polling for pen-down and pen-up touch point events

The abstraction model shown in the following diagram depicts how the AR1021 Touch Driver is positioned in the MPLAB Harmony framework. The AR1021 Touch Driver Library uses the SPI Driver for control and touch data transfers to the AR1021 module.

AR1021 Touch Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the AR1021 Touch Driver module.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, task, and status functions.
Client Functions	Provides functions to open, close, and calibrate the AR1021 Touch Driver.

How the Library Works

The library provides interfaces to support:

- System functions, which provide system module interfaces, device initialization, deinitialization, task, touch, and status functions
- Client functions, which open, close, and calibrate the AR1021 Touch Driver

Initializing the Driver

Before the AR1021 Touch Driver can be opened, it must be configured and initialized. The driver build time configuration is defined by the configuration macros. Refer to the Building the Library section for the location of and more information on the various configuration macros and how these macros should be designed. The driver initialization is configured through the DRV_TOUCH_INIT data structure that is passed to the DRV_TOUCH_AR1021_Initialize function. The initialization parameters include the interrupt source, interrupt pin remap configuration and touch screen resolution. The following code shows an example of initializing the AR1021 Touch Driver.

Example:

```
/* The following code shows an example of designing the
 *  DRV_TOUCH_INIT data structure. It also shows how an example
 *  usage of the DRV_TOUCH_AR1021_Initialize function.
 */

DRV_TOUCH_INIT drvTouchInitData;
SYS_MODULE_OBJ objectHandle;

/* Driver initialization */
objectHandle = DRV_TOUCH_AR1021_Initialize(DRV_TOUCH_INDEX_0,
```

```
                                        (SYS_MODULE_INIT*)drvTouchInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Opening the Driver

To use the AR1021 Touch Driver, the application must open the driver. This is done by calling the DRV_TOUCH_AR1021_Open function.

If successful, the DRV_TOUCH_AR1021_Open function will return a handle to the driver. This handle records the association between the client and the driver instance that was opened. The DRV_TOUCH_AR1021_Open function may return DRV_HANDLE_INVALID in the situation where the driver is not ready to be opened. When this occurs, the application can try opening the driver again. Note that the open function may return an invalid handle in other (error) cases as well. The following code shows an example of the driver being opened.

```
DRV_HANDLE  handle;

handle = DRV_TOUCH_AR1021_Open( DRV_TOUCH_AR1021_INDEX_0,
                                DRV_IO_INTENT_EXCLUSIVE );

if( DRV_HANDLE_INVALID == handle )
{
    // Unable to open the driver
}
```

Tasks Routine

This routine processes the AR1021 Touch Driver commands from the command queue. If the state of the command is initialize or done it returns. If the read request registration is successful the state of command is to decode input. The tasks routine decodes the input and updates the global variables storing the touch input data in form of x and y coordinates. The AR1021 Touch Driver task routine is to be called from SYS_Tasks. The following code shows an example:

```
SYS_MODULE_OBJ      object;   // Returned from DRV_TOUCH_AR1021_Initialize

void SYS_Tasks( void )
{
    DRV_TOUCH_AR1021_Tasks ( object );

    // Do other tasks
}
```

Configuring the Library

Macros

	Name	Description
	DRV_AR1021_CALIBRATION_DELAY	Define the calibration delay.
	DRV_AR1021_CALIBRATION_INSET	Define the calibration inset.
	DRV_AR1021_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_AR1021_INDEX	AR1021 static index selection.
	DRV_AR1021_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_AR1021_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_AR1021_SAMPLE_POINTS	Define the sample points.
	DRV_AR1021_TOUCH_DIAMETER	Define the touch diameter.

Description

The configuration of the AR1021 Touch Driver is accomplished through AR1021 Touch Driver selections in the MPLAB Harmony Configurator (MHC). Based on the selections made, a specific AR1021 Touch Driver is established automatically to execute all system configuration, initialization, and steady-state touch acquisitions.

Refer to Volume III: MPLAB Harmony Configurator (MHC) for more details on system configuration. Refer to the Applications Help section for additional information.

DRV_AR1021_CALIBRATION_DELAY Macro

Define the calibration delay.

File

drv_ar1021_config_template.h

C

```
#define DRV_AR1021_CALIBRATION_DELAY 300
```

Description

AR1021 Calibration Delay

This macro enables the delay between calibration touch points.

Remarks

None.

DRV_AR1021_CALIBRATION_INSET Macro

Define the calibration inset.

File

drv_ar1021_config_template.h

C

```
#define DRV_AR1021_CALIBRATION_INSET 25
```

Description

AR1021 Calibration Inset

This macro define the calibration inset.

Remarks

None.

DRV_AR1021_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_ar1021_config_template.h

C

```
#define DRV_AR1021_CLIENTS_NUMBER 1
```

Description

AR1021 Maximum Number of Clients

This definition selected the maximum number of clients that the AR1021 driver can support at run time.

Remarks

None.

DRV_AR1021_INDEX Macro

AR1021 static index selection.

File

drv_ar1021_config_template.h

C

```
#define DRV_AR1021_INDEX DRV_AR1021_INDEX_0
```

Description

AR1021 Static Index Selection

AR1021 static index selection for the driver object reference.

Remarks

This index is required to make a reference to the driver object.

DRV_AR1021_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported.

File

drv_ar1021_config_template.h

C

```
#define DRV_AR1021_INSTANCES_NUMBER 1
```

Description

AR1021 hardware instance configuration

This macro sets up the maximum number of hardware instances that can be supported.

Remarks

None.

DRV_AR1021_INTERRUPT_MODE Macro

Controls operation of the driver in the interrupt or polled mode.

File

drv_ar1021_config_template.h

C

```
#define DRV_AR1021_INTERRUPT_MODE false
```

Description

AR1021 Interrupt And Polled Mode Operation Control

This macro controls the operation of the driver in the interrupt mode of operation. The possible values of this macro are:

- true - Select if interrupt mode of AR1021 operation is desired
- false - Select if polling mode of AR1021 operation is desired

Not defining this option to true or false will result in a build error.

Remarks

None.

DRV_AR1021_SAMPLE_POINTS Macro

Define the sample points.

File

drv_ar1021_config_template.h

C

```
#define DRV_AR1021_SAMPLE_POINTS 4
```

Description

AR1021 Sample Points

AR1021 sample points

Remarks

None.

DRV_AR1021_TOUCH_DIAMETER Macro

Define the touch diameter.

File

drv_ar1021_config_template.h

C

```
#define DRV_AR1021_TOUCH_DIAMETER 10
```

Description

AR1021 Touch Diameter

This macro defines the touch diameter

Remarks

None.

Building the Library

This section lists the files that are available in the AR1021 Touch Driver Library.

Description

This section list the files that are available in the `\src` folder of the AR1021 Touch Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/touch/ar1021`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_ar1021.h`	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/drv_ar1021.c`	Basic AR1021 Touch Driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The AR1021 Touch Driver Library depends on the following modules:

- Interrupt System Service Library
- Ports System Service Library
- Touch System Service Library
- I2C Driver Library

Library Interface

a) System Functions

	Name	Description
	DRV_TOUCH_AR1021_Deinitialize	De-initializes the specified instance of the AR1021 driver module.
	DRV_TOUCH_AR1021_FactoryDefaultSet	Set AR1021 controller to factory default configuration settings.
	DRV_TOUCH_AR1021_Initialize	Initializes the AR1021 instance for the specified driver index

	DRV_TOUCH_AR1021_RegisterConfigWrite	Write a value to the given AR1021 configuration register.
	DRV_TOUCH_AR1021_Status	Provides the current status of the AR1021 driver module.
	DRV_TOUCH_AR1021_Tasks	Maintains the driver's state machine and implements its task queue processing. **Implementation:** Dynamic
	DRV_TOUCH_AR1021_TouchDataRead	Notifies the driver that the current touch data has been read
	DRV_TOUCH_AR1021_TouchGetX	Returns the x coordinate of touch input. **Implementation:** Dynamic
	DRV_TOUCH_AR1021_TouchGetY	Returns the y coordinate of touch input. **Implementation:** Dynamic
	DRV_TOUCH_AR1021_TouchPenGet	Returns the PEN state of the touch event.
	DRV_TOUCH_AR1021_TouchStatus	Returns the status of the current touch input.

b) Client Functions

	Name	Description
	DRV_TOUCH_AR1021_Calibrate	Calibrate the touch screen
	DRV_TOUCH_AR1021_CalibrationSet	Set calibration with pre-defined points..
	DRV_TOUCH_AR1021_Close	Closes an opened instance of the AR1021 driver
	DRV_TOUCH_AR1021_Open	Opens the specified AR1021 driver instance and returns a handle to it. **Implementation:** Dynamic

c) Data Types and Constants

	Name	Description
	DRV_TOUCH_AR1021_CALIBRATION_PROMPT_CALLBACK	Defines the callback functions required to inform the user of touch and release targets.
	DRV_TOUCH_AR1021_HANDLE	Touch screen controller AR1021 driver handle.
	DRV_TOUCH_AR1021_MODULE_ID	This is type DRV_TOUCH_AR1021_MODULE_ID.
	DRV_TOUCH_AR1021_TASK_STATE	Enumeration defining AR1021 touch controller driver task state.
	DRV_TOUCH_AR1021_HANDLE_INVALID	Definition of an invalid handle.
	DRV_TOUCH_AR1021_INDEX_0	AR1021 driver index definitions.
	DRV_TOUCH_AR1021_INDEX_COUNT	Number of valid AR1021 driver indices.

Description

This section describes the API functions of the AR1021 Touch Driver Library.

a) System Functions

DRV_TOUCH_AR1021_Deinitialize Function

De-initializes the specified instance of the AR1021 driver module.

File

drv_ar1021.h

C

```
void DRV_TOUCH_AR1021_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

De-initializes the specified instance of the AR1021 driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again.

This function will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_TOUCH_AR1021_Status operation. The system has to use DRV_TOUCH_AR1021_Status to find out when the module is in the ready state.

Preconditions

Function DRV_TOUCH_AR1021_Initialize must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_TOUCH_AR1021_Initialize
SYS_STATUS          status;

DRV_TOUCH_AR1021_Deinitialize ( object );

status = DRV_TOUCH_AR1021_Status( object );
if( SYS_MODULE_UNINITIALIZED == status )
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_TOUCH_AR1021_Initialize

Function

void DRV_TOUCH_AR1021_Deinitialize (SYS_MODULE_OBJ object)

DRV_TOUCH_AR1021_FactoryDefaultSet Function

Set AR1021 controller to factory default configuration settings.

File

drv_ar1021.h

C

```
void DRV_TOUCH_AR1021_FactoryDefaultSet();
```

Returns

None

Description

This function returns the AR1021 to operate on factory default configuration settings.

Remarks

A power cycle is required to run on the default settings.

Preconditions

The DRV_TOUCH_AR1021_Open routine must have been called for the specified AR1021 driver instance.

Example

```
DRV_TOUCH_AR1021_FactoryDefaultSet ( void );
```

Function

void DRV_TOUCH_AR1021_FactoryDefaultSet(void)

DRV_TOUCH_AR1021_Initialize Function

Initializes the AR1021 instance for the specified driver index

File

drv_ar1021.h

C

```
SYS_MODULE_OBJ DRV_TOUCH_AR1021_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const
init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the AR1021 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the 'init' parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the AR1021 module ID. For example, driver instance 0 can be assigned to AR10212. If the driver is built statically, then some of the initialization parameters are overridden by configuration macros. Refer to the description of the DRV_TOUCH_AR1021_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This routine must be called before any other AR1021 routine is called.

This routine should only be called once during system initialization unless DRV_TOUCH_AR1021_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

None.

Example

```
DRV_TOUCH_INIT      drvAr1021InitData;
SYS_MODULE_OBJ      objectHandle;

objectHandle = DRV_TOUCH_AR1021_Initialize(DRV_TOUCH_AR1021_INDEX_1, (SYS_MODULE_INIT*)drvAr1021InitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized. Please note this is not the AR1021 id. The hardware AR1021 id is set in the initialization structure. This is the index of the driver index to use.
init	Pointer to a data structure containing any data necessary to initialize the driver. If this pointer is NULL, the driver uses the static initialization override macros for each member of the initialization data structure.

Function

SYS_MODULE_OBJ DRV_TOUCH_AR1021_Initialize(const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init)

DRV_TOUCH_AR1021_RegisterConfigWrite Function

Write a value to the given AR1021 configuration register.

File

drv_ar1021.h

C

```
void DRV_TOUCH_AR1021_RegisterConfigWrite(uint16_t regOffset, uint8_t Value);
```

Returns

None

Description

This function set a value to the given AR1021 configuration register.

Remarks

none

Preconditions

The DRV_TOUCH_AR1021_Open routine must have been called for the specified AR1021 driver instance.

Example

```
DRV_TOUCH_AR1021_RegisterConfigWrite(uint16_t regOffset, uint8_t Value);
```

Function

void DRV_TOUCH_AR1021_RegisterConfigWrite(uint16_t regOffset, uint8_t Value)

DRV_TOUCH_AR1021_Status Function

Provides the current status of the AR1021 driver module.

File

drv_ar1021.h

C

```
SYS_STATUS DRV_TOUCH_AR1021_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system level operation and cannot start another

Description

This function provides the current status of the AR1021 driver module.

Remarks

Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_MODULE_UNINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR.

This function can be used to determine when any of the driver's module level operations has completed.

If the status operation returns SYS_STATUS_BUSY, the previous operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This function will NEVER block waiting for hardware.

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The DRV_TOUCH_AR1021_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_TOUCH_AR1021_Initialize
SYS_STATUS          status;

status = DRV_TOUCH_AR1021_Status( object );
if( SYS_STATUS_READY != status )
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_TOUCH_AR1021_Initialize

Function

SYS_STATUS DRV_TOUCH_AR1021_Status (SYS_MODULE_OBJ object)

DRV_TOUCH_AR1021_Tasks Function

Maintains the driver's state machine and implements its task queue processing.

Implementation: Dynamic

File

drv_ar1021.h

C

```
void DRV_TOUCH_AR1021_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal state machine and implement its command queue processing. It is always called from SYS_Tasks() function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks)

Preconditions

The DRV_TOUCH_AR1021_Initialize routine must have been called for the specified AR1021 driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_TOUCH_AR1021_Initialize

while( true )
{
    DRV_TOUCH_AR1021_Tasks ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_TOUCH_AR1021_Initialize)

Function

void DRV_TOUCH_AR1021_Tasks (SYS_MODULE_OBJ object);

DRV_TOUCH_AR1021_TouchDataRead Function

Notifies the driver that the current touch data has been read

File

drv_ar1021.h

C

```
void DRV_TOUCH_AR1021_TouchDataRead(const SYS_MODULE_INDEX index);
```

Returns

None.

Description

Notifies the driver that the current touch data has been read

Function

void DRV_TOUCH_AR1021_TouchDataRead(const SYS_MODULE_INDEX index)

DRV_TOUCH_AR1021_TouchGetX Function

Returns the x coordinate of touch input.

Implementation: Dynamic

File

drv_ar1021.h

C

```
short DRV_TOUCH_AR1021_TouchGetX(uint8_t touchNumber);
```

Returns

It returns the x coordinate of the touch input in terms of number of pixels.

Description

It returns the x coordinate in form of number of pixels for a touch input denoted by touchNumber.

Parameters

Parameters	Description
touchNumber	index to the touch input.

Function

short DRV_TOUCH_AR1021_TouchGetX(uint8 touchNumber)

DRV_TOUCH_AR1021_TouchGetY Function

Returns the y coordinate of touch input.

Implementation: Dynamic

File

drv_ar1021.h

C

```
short DRV_TOUCH_AR1021_TouchGetY(uint8_t touchNumber);
```

Returns

It returns the y coordinate of the touch input in terms of number of pixels.

Description

It returns the y coordinate in form of number of pixes for a touch input denoted by touchNumber.

Parameters

Parameters	Description
touchNumber	index to the touch input.

Function

short DRV_TOUCH_AR1021_TouchGetY(uint8 touchNumber)

DRV_TOUCH_AR1021_TouchPenGet Function

Returns the PEN state of the touch event.

File

drv_ar1021.h

C

```
DRV_TOUCH_PEN_STATE DRV_TOUCH_AR1021_TouchPenGet(uint8_t touchNumber);
```

Returns

It returns DRV_TOUCH_PEN_STATE

Description

It returns the PEN state of the last touch event corresponding to the x and y position.

Parameters

Parameters	Description
touchNumber	index to the touch input.

Function

DRV_TOUCH_PEN_STATE DRV_TOUCH_AR1021_TouchPenGet(uint8_t touchNumber)

DRV_TOUCH_AR1021_TouchStatus Function

Returns the status of the current touch input.

File

drv_ar1021.h

C

```
DRV_TOUCH_POSITION_STATUS DRV_TOUCH_AR1021_TouchStatus(const SYS_MODULE_INDEX index);
```

Returns

It returns the status of the current touch input.

Description

It returns the status of the current touch input.

Function

DRV_TOUCH_POSITION_SINGLE DRV_TOUCH_AR1021_TouchStatus(const SYS_MODULE_INDEX index)

b) Client Functions

DRV_TOUCH_AR1021_Calibrate Function

Calibrate the touch screen

File

drv_ar1021.h

C

```
void DRV_TOUCH_AR1021_Calibrate(const DRV_TOUCH_AR1021_CALIBRATION_PROMPT_CALLBACK * prompt);
```

Returns

None

Description

This function display calibration points on the display to enable calibration.

Remarks

None

Preconditions

The DRV_TOUCH_AR1021_Initialize routine must have been called for the specified AR1021 driver instance.

Example

```
DRV_TOUCH_AR1021_Calibrate ( handle );
```

Function

void DRV_TOUCH_AR1021_Calibrate ((const DRV_TOUCH_AR1021_CALIBRATION_PROMPT_CALLBACK * prompt))

DRV_TOUCH_AR1021_CalibrationSet Function

Set calibration with pre-defined points..

File

drv_ar1021.h

C

```
void DRV_TOUCH_AR1021_CalibrationSet(DRV_TOUCH_SAMPLE_POINTS * samplePoints);
```

Returns

None

Description

This function allows for the setting of pre-loaded calibration points.

Remarks

None

Preconditions

The DRV_TOUCH_AR1021_Open routine must have been called for the specified AR1021 driver instance.

Example

```
DRV_TOUCH_AR1021_CalibrationSet ( void );
```

Function

void DRV_TOUCH_AR1021_CalibrationSet(void)

DRV_TOUCH_AR1021_Close Function

Closes an opened instance of the AR1021 driver

File

drv_ar1021.h

C

```
void DRV_TOUCH_AR1021_Close(DRV_HANDLE handle);
```

Returns

None

Description

This function closes an opened instance of the AR1021 driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_TOUCH_AR1021_Open before the caller may use the driver again. This function is thread safe in a RTOS application.

Usually there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_TOUCH_AR1021_Initialize routine must have been called for the specified AR1021 driver instance.

DRV_TOUCH_AR1021_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_TOUCH_AR1021_Open

DRV_TOUCH_AR1021_Close ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TOUCH_AR1021_Close (DRV_HANDLE handle)

DRV_TOUCH_AR1021_Open Function

Opens the specified AR1021 driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_ar1021.h

C

```
DRV_HANDLE DRV_TOUCH_AR1021_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. An error can occur when the following is true:

- if the number of client objects allocated via DRV_TOUCH_AR1021_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid

Description

This routine opens the specified AR1021 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The current version of driver does not support the DRV_IO_INTENT feature. The driver is by default non-blocking. The driver can perform both read and write to the AR1021 device. The driver supports single client only.

Remarks

The handle returned is valid until the DRV_TOUCH_AR1021_Close routine is called. This routine will NEVER block waiting for hardware. If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

The DRV_TOUCH_AR1021_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_TOUCH_AR1021_Open( DRV_TOUCH_AR1021_INDEX_0, DRV_IO_INTENT_EXCLUSIVE );

if( DRV_HANDLE_INVALID == handle )
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Index of the driver initialized with DRV_TOUCH_AR1021_Initialize().
intent	Zero or more of the values from the enumeration DRV_IO_INTENT ORed together to indicate the intended use of the driver. The current version of driver does not support the selective IO intent feature.

Function

DRV_HANDLE DRV_TOUCH_AR1021_Open (const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT intent)

c) Data Types and Constants

DRV_TOUCH_AR1021_CALIBRATION_PROMPT_CALLBACK Structure

Defines the callback functions required to inform the user of touch and release targets.

File

drv_ar1021.h

C

```
typedef struct {
  void (* firstPromptCallback)(void);
  void (* secondPromptCallback)(void);
  void (* thirdPromptCallback)(void);
  void (* fourthPromptCallback)(void);
  void (* completeCallback)(void);
} DRV_TOUCH_AR1021_CALIBRATION_PROMPT_CALLBACK;
```

Members

Members	Description
void (* firstPromptCallback)(void);	first calibration target
void (* secondPromptCallback)(void);	second calibration target
void (* thirdPromptCallback)(void);	third calibration target
void (* fourthPromptCallback)(void);	fourth calibration target
void (* completeCallback)(void);	complete calibration

Description

TOUCH Driver Calibration Initialization Data

This data type defines the callback function pointers required to inform of touch and release targets. The driver will invoke each callback in sequential order. The host code can display graphic and/or textual content to direct the user when a where on the LCD display to touch and release.

Remarks

None.

DRV_TOUCH_AR1021_HANDLE Type

Touch screen controller AR1021 driver handle.

File

drv_ar1021.h

C

```
typedef uintptr_t DRV_TOUCH_AR1021_HANDLE;
```

Description

AR1021 Driver Handle

Touch controller AR1021 driver handle is a handle for the driver client object. Each driver with successful open call will return a new handle to the client object.

Remarks

None.

DRV_TOUCH_AR1021_MODULE_ID Enumeration

File

drv_ar1021.h

C

```
typedef enum {
  AR1021_ID_1 = 0,
  AR1021_NUMBER_OF_MODULES
} DRV_TOUCH_AR1021_MODULE_ID;
```

Description

This is type DRV_TOUCH_AR1021_MODULE_ID.

DRV_TOUCH_AR1021_TASK_STATE Enumeration

Enumeration defining AR1021 touch controller driver task state.

File

drv_ar1021.h

C

```
typedef enum {
  DRV_TOUCH_AR1021_TASK_STATE_INIT = 0,
  DRV_TOUCH_AR1021_TASK_STATE_DONE
} DRV_TOUCH_AR1021_TASK_STATE;
```

Members

Members	Description
DRV_TOUCH_AR1021_TASK_STATE_INIT = 0	Task initialize state
DRV_TOUCH_AR1021_TASK_STATE_DONE	Task complete state

Description

AR1021 Touch Controller Driver Task State

This enumeration defines the AR1021 touch controller driver task state. The task state helps to synchronize the operations of initialization the the task, adding the read input task to the task queue once the touch controller notifies the available touch input and a decoding the touch input received.

Remarks

None.

DRV_TOUCH_AR1021_HANDLE_INVALID Macro

Definition of an invalid handle.

File

drv_ar1021.h

C

```
#define DRV_TOUCH_AR1021_HANDLE_INVALID ((DRV_TOUCH_AR1021_HANDLE)(-1))
```

Description

AR1021 Driver Invalid Handle

This is the definition of an invalid handle. An invalid handle is is returned by DRV_TOUCH_AR1021_Open() and DRV_AR1021_Close() functions if the request was not successful.

Remarks

None.

DRV_TOUCH_AR1021_INDEX_0 Macro

AR1021 driver index definitions.

File

drv_ar1021.h

C

```
#define DRV_TOUCH_AR1021_INDEX_0 0
```

Description

AR1021 Driver Module Index Numbers

These constants provide the AR1021 driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_AR1021_Initialize and

DRV_AR1021_Open functions to identify the driver instance in use.

DRV_TOUCH_AR1021_INDEX_COUNT Macro

Number of valid AR1021 driver indices.

File

drv_ar1021.h

C

```
#define DRV_TOUCH_AR1021_INDEX_COUNT 1
```

Description

AR1021 Driver Module Index Count

This constant identifies the number of valid AR1021 driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific header files defined as part of the peripheral libraries.

Files

Files

Name	Description
drv_ar1021.h	Touch controller AR1021 driver implementation.

Description

This section lists the source and header files used by the AR1021 Touch Driver Library.

drv_ar1021.h

Touch controller AR1021 driver implementation.

Enumerations

	Name	Description
	DRV_TOUCH_AR1021_MODULE_ID	This is type DRV_TOUCH_AR1021_MODULE_ID.
	DRV_TOUCH_AR1021_TASK_STATE	Enumeration defining AR1021 touch controller driver task state.

Functions

	Name	Description
	DRV_TOUCH_AR1021_Calibrate	Calibrate the touch screen
	DRV_TOUCH_AR1021_CalibrationSet	Set calibration with pre-defined points..
	DRV_TOUCH_AR1021_Close	Closes an opened instance of the AR1021 driver
	DRV_TOUCH_AR1021_Deinitialize	De-initializes the specified instance of the AR1021 driver module.
	DRV_TOUCH_AR1021_FactoryDefaultSet	Set AR1021 controller to factory default configuration settings.
	DRV_TOUCH_AR1021_Initialize	Initializes the AR1021 instance for the specified driver index
	DRV_TOUCH_AR1021_Open	Opens the specified AR1021 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_TOUCH_AR1021_RegisterConfigWrite	Write a value to the given AR1021 configuration register.
	DRV_TOUCH_AR1021_Status	Provides the current status of the AR1021 driver module.
	DRV_TOUCH_AR1021_Tasks	Maintains the driver's state machine and implements its task queue processing. **Implementation:** Dynamic
	DRV_TOUCH_AR1021_TouchDataRead	Notifies the driver that the current touch data has been read
	DRV_TOUCH_AR1021_TouchGetX	Returns the x coordinate of touch input. **Implementation:** Dynamic
	DRV_TOUCH_AR1021_TouchGetY	Returns the y coordinate of touch input. **Implementation:** Dynamic
	DRV_TOUCH_AR1021_TouchPenGet	Returns the PEN state of the touch event.

	DRV_TOUCH_AR1021_TouchStatus	Returns the status of the current touch input.

Macros

	Name	Description
	DRV_TOUCH_AR1021_HANDLE_INVALID	Definition of an invalid handle.
	DRV_TOUCH_AR1021_INDEX_0	AR1021 driver index definitions.
	DRV_TOUCH_AR1021_INDEX_COUNT	Number of valid AR1021 driver indices.

Structures

	Name	Description
	DRV_TOUCH_AR1021_CALIBRATION_PROMPT_CALLBACK	Defines the callback functions required to inform the user of touch and release targets.

Types

	Name	Description
	DRV_TOUCH_AR1021_HANDLE	Touch screen controller AR1021 driver handle.

Description

Touch controller AR1021 driver file

This file consist of touch controller AR1021 driver interfaces. It implements the driver interfaces which read the touch input data from AR1021 through SPI bus.

File Name

drv_ar1021.c

MTCH6301 Touch Driver Library

This topic describes the MTCH6301 Touch Driver Library.

Introduction

This library provides an interface to manage the MTCH6301 Touch Driver module on the Microchip family of microcontrollers in different modes of operation.

Description

The MPLAB Harmony MTCH6301 Touch Driver provides a high-level interface to the MTCH6301 touch controller device. This driver provides application routines to read the touch input data from the touch screen. The MTCH6301 device can notify the availability of touch input data through external interrupt. The MTCH6301 driver allows the application to map a controller pin as an external interrupt pin.

Currently, the MTCH6301 Touch Driver only supports non-gestural single-fingered touch input.

Using the Library

This topic describes the basic architecture of the MTCH6301 Touch Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_mtch6301.h`

The interface to the MTCH6301 Touch Driver library is defined in the `drv_mtch6301.h` header file. Any C language source (`.c`) file that uses the MTCH6301 Touch Driver library should include this header.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the MTCH6301 Touch Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

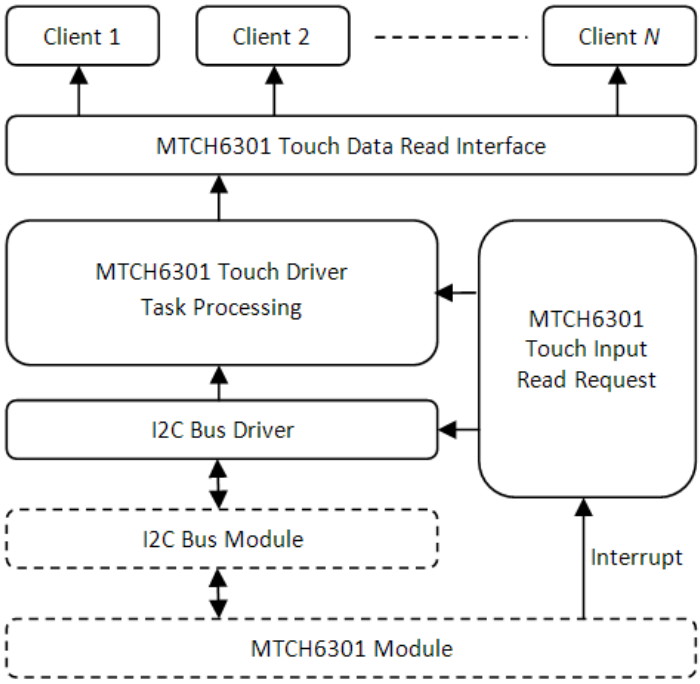
The MTCH6301 Touch Driver has routines to perform the following operations:

- Sending read request
- Reading the touch input data

- Access to the touch input data

The driver initialization routines allow the application to initialize the driver. The driver must be initialized before it can be used by application. Once the driver is initialized the driver open routine allows to retrieve the client handle. Once the touch input is available a touch input read request is sent and input data is retrieved in a buffer. The buffer data is then decoded to get the x and y coordinate of the touch screen in the form of the number of pixels.

MTCH6301 Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the MTCH6301 Touch Driver.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, open, close, task, and status functions.

How the Library Works

The library provides interfaces to support:

- System functions, which provide system module interfaces, device initialization, deinitialization, open, close, task, and status functions.
- Read Request function, which provides Touch input data read request function
- Read Touch Input function, which provides functions retrieving updated Touch input in the form x and y coordinates.

Initializing the Driver

Before the MTCH6301 Touch Driver can be opened, it must be configured and initialized. The driver build time configuration is defined by the configuration macros. Refer to the Building the Library section for the location of and more information on the various configuration macros and how these macros should be designed. The driver initialization is configured through the DRV_TOUCH_INIT data structure that is passed to the DRV_TOUCH_MTCH6301_Initialize function. The initialization parameters include the interrupt source, interrupt pin remap configuration and touch screen resolution. The following code shows an example of initializing the MTCH6301 Touch Driver.

Example:

```
/* The following code shows an example of designing the
 *  DRV_TOUCH_INIT data structure. It also shows how an example
 *  usage of the DRV_TOUCH_MTCH6301_Initialize function.
 */
```

```
DRV_TOUCH_INIT drvTouchInitData;
SYS_MODULE_OBJ objectHandle;

/* Touch Module Id*/
drvTouchInitData.touchId                    = DRV_TOUCH_INDEX_0;

/* I2C Bus driver open */
drvTouchInitData.drvOpen                    = DRV_I2C_Open;

/* Interrupt Source for Touch */
drvTouchInitData.interruptSource            = INT_SOURCE_EXTERNAL_1;

/* Interrupt Pin function mapping */
drvTouchInitData.interruptPort.inputFunction = INPUT_FUNC_INT1;

/* Pin to be mapped as interrupt pin */
drvTouchInitData.interruptPort.inputPin     = INPUT_PIN_RPE8;

/* Analog pin number */
drvTouchInitData.interruptPort.analogPin    = PORTS_ANALOG_PIN_25;

/* Pin Mode of analog pin */
drvTouchInitData.interruptPort.pinMode      = PORTS_PIN_MODE_DIGITAL;

/* Interrupt pin port */
drvTouchInitData.interruptPort.channel      = PORT_CHANNEL_E;

/* Interrupt pin port maskl */
drvTouchInitData.interruptPort.dataMask     = 0x8;

/* Touch screen orientation */
drvTouchInitData.orientation                = DISP_ORIENTATION;

/* Touch screen horizontal resolution */
drvTouchInitData.horizontalResolution           = DISP_HOR_RESOLUTION;

/* Touch screen vertical resolution */
drvTouchInitData.verticalResolution         = DISP_VER_RESOLUTION;

/* Driver initialization */
objectHandle = DRV_TOUCH_MTCH6301_Initialize(DRV_TOUCH_INDEX_0,
                                             (SYS_MODULE_INIT*)drvTouchInitData);
    if (SYS_MODULE_OBJ_INVALID == objectHandle)
    {
        // Handle error
    }
```

Opening the Driver

To use the MTCH6301 Touch Driver, the application must open the driver. This is done by calling the DRV_TOUCH_MTCH6301_Open function.

If successful, the DRV_TOUCH_MTCH6301_Open function will return a handle to the driver. This handle records the association between the client and the driver instance that was opened. The DRV_TOUCH_MTCH6301_Open function may return DRV_HANDLE_INVALID in the situation where the driver is not ready to be opened. When this occurs, the application can try opening the driver again. Note that the open function may return an invalid handle in other (error) cases as well. The following code shows an example of the driver being opened.

```
    DRV_HANDLE  handle;

    handle = DRV_TOUCH_MTCH6301_Open( DRV_TOUCH_MTCH6301_INDEX_0,
                                      DRV_IO_INTENT_EXCLUSIVE );

    if( DRV_HANDLE_INVALID == handle )
    {
        // Unable to open the driver
    }
```

Touch Input Read Request

To read the touch input from the MTCH6301 touch controller device, a read request must be registered. This is done by calling the DRV_TOUCH_MTCH6301_ReadRequest. If successful it registers a buffer read request to the I2C command queue. It also adds a input decode command to the MTCH6301 command queue once the I2C returns with touch input data. It can return error if the driver instance object is invalid or the MTCH6301 command queue is full. The read request is to be called from the MTCH6301 ISR. This ISR is triggered once the touch input is available. The following code shows an example of a MTCH6301 read request registration:

```
SYS_MODULE_OBJ      object;   // Returned from DRV_TOUCH_MTCH6301_Initialize

void __ISR(_EXTERNAL_INT_VECTOR, ipl5) _IntHandlerDrvMtch6301(void)
{
    DRV_TOUCH_MTCH6301_ReadRequest ( object );

    // Do other tasks
}
```

Tasks Routine

This routine processes the MTCH6301 commands from the command queue. If the state of the command is initialize or done it returns. If the read request registration is successful the state of command is to decode input. The tasks routine decodes the input and updates the global variables storing the touch input data in form of x and y coordinates. The MTCH6301 Touch Driver task routine is to be called from SYS_Tasks. The following code shows an example:

```
SYS_MODULE_OBJ      object;   // Returned from DRV_TOUCH_MTCH6301_Initialize

void SYS_Tasks( void )
{
    DRV_TOUCH_MTCH6301_Tasks ( object );

    // Do other tasks
}
```

Configuring the Library

Macros

	Name	Description
	DRV_MTCH6301_CALIBRATION_DELAY	Defines the calibration delay.
	DRV_MTCH6301_CALIBRATION_INSET	Defines the calibration inset.
	DRV_MTCH6301_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_MTCH6301_INDEX	MTCH6301 static index selection.
	DRV_MTCH6301_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_MTCH6301_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_MTCH6301_SAMPLE_POINTS	Define the sample points.
	DRV_MTCH6301_TOUCH_DIAMETER	Defines the touch diameter.

Description

The configuration of the MTCH6301 Touch Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the MTCH6301 Touch Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the MTCH6301 Touch Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_MTCH6301_CALIBRATION_DELAY Macro

Defines the calibration delay.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_CALIBRATION_DELAY 300
```

Description

MTCH6301 Calibration Delay

This macro enables the delay between calibration touch points.

Remarks

None.

DRV_MTCH6301_CALIBRATION_INSET Macro

Defines the calibration inset.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_CALIBRATION_INSET 25
```

Description

MTCH6301 Calibration Inset

This macro defines the calibration inset.

Remarks

None.

DRV_MTCH6301_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_CLIENTS_NUMBER 1
```

Description

MTCH6301 maximum number of clients

This macro selects the maximum number of clients.

This definition selected the maximum number of clients that the MTCH6301 driver can support at run time.

Remarks

None.

DRV_MTCH6301_INDEX Macro

MTCH6301 static index selection.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_INDEX DRV_MTCH6301_INDEX_0
```

Description

MTCH6301 Static Index Selection

This macro specifies the static index selection for the driver object reference.

Remarks

This index is required to make a reference to the driver object.

DRV_MTCH6301_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_INSTANCES_NUMBER 1
```

Description

MTCH6301 hardware instance configuration

This macro sets up the maximum number of hardware instances that can be supported.

Remarks

None.

DRV_MTCH6301_INTERRUPT_MODE Macro

Controls operation of the driver in the interrupt or polled mode.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_INTERRUPT_MODE false
```

Description

MTCH6301 Interrupt And Polled Mode Operation Control

This macro controls the operation of the driver in the interrupt mode of operation. The possible values of this macro are:

- true - Select if interrupt mode of MTCH6301 operation is desired
- false - Select if polling mode of MTCH6301 operation is desired

Not defining this option to true or false will result in a build error.

Remarks

None.

DRV_MTCH6301_SAMPLE_POINTS Macro

Define the sample points.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_SAMPLE_POINTS 4
```

Description

MTCH6301 Sample Points

MTCH6301 sample points

Remarks

None.

DRV_MTCH6301_TOUCH_DIAMETER Macro

Defines the touch diameter.

File

drv_mtch6301_config_template.h

C

```
#define DRV_MTCH6301_TOUCH_DIAMETER 10
```

Description

MTCH6301 Touch Diameter

This macro defines the touch diameter

Remarks

None.

Building the Library

This section lists the files that are available in the MTCH6301 Touch Driver Library.

Description

This section list the files that are available in the \src folder of the MTCH6301 Touch Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/touch/mtch6301`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_mtch6301.h`	Header file that exports the driver API.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/drv_mtch6301.c`	Basic MTCH6301 Touch Driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The MTCH6301 Touch Driver Library depends on the following modules:

- Interrupt System Service Library
- Ports System Service Library
- Touch System Service Library
- I2C Driver Library

Library Interface

a) System Functions

	Name	Description
	DRV_TOUCH_MTCH6301_Close	Closes an opened instance of the MTCH6301 driver. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Deinitialize	Deinitializes the specified instance of the MTCH6301 driver module. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Initialize	Initializes the MTCH6301 instance for the specified driver index. **Implementation:** Dynamic

	DRV_TOUCH_MTCH6301_Open	Opens the specified MTCH6301 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Status	Provides the current status of the MTCH6301 driver module. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Tasks	Maintains the driver's state machine and implements its task queue processing. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_ReadRequest	Sends a read request to I2C bus driver and adds the read task to queue. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_TouchGetX	Returns the x coordinate of touch input. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_TouchGetY	Returns the y coordinate of touch input. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_TouchDataRead	Notifies the driver that the current touch data has been read
	DRV_TOUCH_MTCH6301_TouchStatus	Returns the status of the current touch input.

b) Data Types and Constants

	Name	Description
	_DRV_MTCH6301_CLIENT_OBJECT	MTCH6301 Driver client object maintaining client data.
	DRV_TOUCH_MTCH6301_HANDLE	Touch screen controller MTCH6301 driver handle.
	DRV_TOUCH_MTCH6301_MODULE_ID	Number of valid MTCH6301 driver indices.
	DRV_TOUCH_MTCH6301_HANDLE_INVALID	Definition of an invalid handle.
	DRV_TOUCH_MTCH6301_I2C_READ_FRAME_SIZE	I2C Frame size for reading MTCH6301 touch input.
	DRV_TOUCH_MTCH6301_CLIENT_OBJECT	MTCH6301 Driver client object maintaining client data.
	DRV_TOUCH_MTCH6301_INDEX_0	MTCH6301 driver index definitions.
	DRV_TOUCH_MTCH6301_INDEX_1	This is macro DRV_TOUCH_MTCH6301_INDEX_1.
	DRV_TOUCH_MTCH6301_INDEX_COUNT	Number of valid Touch controller MTCH6301 driver indices.
	DRV_TOUCH_MTCH6301_OBJECT	Defines the data structure maintaining MTCH6301 driver instance object.
	DRV_TOUCH_MTCH6301_TASK_QUEUE	Defines the MTCH6301 Touch Controller driver task data structure.
	DRV_TOUCH_MTCH6301_TASK_STATE	Enumeration defining MTCH6301 touch controller driver task state.
	DRV_TOUCH_MTCH6301_I2C_MASTER_READ_ID	MTCH6301 input read, I2C address from where master reads touch input data.
	DRV_TOUCH_MTCH6301_I2C_MASTER_WRITE_ID	MTCH6301 command register write, I2C address where master sends the commands.

Description

This section describes the API functions of the MTCH6301 Touch Driver library.

Refer to each section for a detailed description.

a) System Functions

DRV_TOUCH_MTCH6301_Close Function

Closes an opened instance of the MTCH6301 driver.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
void DRV_TOUCH_MTCH6301_Close(DRV_HANDLE handle);
```

Returns

None

Description

This function closes an opened instance of the MTCH6301 driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_TOUCH_MTCH6301_Open before the caller may use the driver again. This function is thread safe in a RTOS application.

Usually, there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_TOUCH_MTCH6301_Initialize routine must have been called for the specified MTCH6301 driver instance.

DRV_TOUCH_MTCH6301_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_TOUCH_MTCH6301_Open

DRV_TOUCH_MTCH6301_Close ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_TOUCH_MTCH6301_Close (DRV_HANDLE handle)

DRV_TOUCH_MTCH6301_Deinitialize Function

Deinitializes the specified instance of the MTCH6301 driver module.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
void DRV_TOUCH_MTCH6301_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the MTCH6301 driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again.

This function will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_TOUCH_MTCH6301_Status operation. The system has to use DRV_TOUCH_MTCH6301_Status to determine when the module is in the ready state.

Preconditions

Function DRV_TOUCH_MTCH6301_Initialize must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Parameter: object - Driver object handle, returned from DRV_TOUCH_MTCH6301_Initialize

Example

```
SYS_MODULE_OBJ      object;    //Returned from DRV_TOUCH_MTCH6301_Initialize
SYS_STATUS          status;

DRV_TOUCH_MTCH6301_Deinitialize ( object );

status = DRV_TOUCH_MTCH6301_Status( object );
if( SYS_MODULE_UNINITIALIZED == status )
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Function

void DRV_TOUCH_MTCH6301_Deinitialize (SYS_MODULE_OBJ object)

DRV_TOUCH_MTCH6301_Initialize Function

Initializes the MTCH6301 instance for the specified driver index.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
SYS_MODULE_OBJ DRV_TOUCH_MTCH6301_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the MTCH6301 driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the 'init' parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the MTCH6301 module ID. For example, driver instance 0 can be assigned to MTCH63012. If the driver is built statically, then some of the initialization parameters are overridden by configuration macros. Refer to the description of the DRV_TOUCH_MTCH6301_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This routine must be called before any other MTCH6301 routine is called.

This routine should only be called once during system initialization unless DRV_TOUCH_MTCH6301_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

None.

Example

```
DRV_TOUCH_MTCH6301_INIT        init;
SYS_MODULE_OBJ      objectHandle;

// Populate the MTCH6301 initialization structure
// Touch Module Id
init.touchId                      = DRV_TOUCH_INDEX_0;

// I2C Bus driver open
init.drvOpen                      = DRV_I2C_Open;

// Interrupt Source for Touch
init.interruptSource              = INT_SOURCE_EXTERNAL_1;

// Interrupt Pin function mapping
init.interruptPort.inputFunction  = INPUT_FUNC_INT1;

// Pin to be mapped as interrupt pin
init.interruptPort.inputPin       = INPUT_PIN_RPE8;

// Analog pin number
init.interruptPort.analogPin      = PORTS_ANALOG_PIN_25;

// Pin Mode of analog pin
init.interruptPort.pinMode        = PORTS_PIN_MODE_DIGITAL;

// Interrupt pin port
init.interruptPort.channel        = PORT_CHANNEL_E;

// Interrupt pin port maskl
init.interruptPort.dataMask       = 0x8;
```

```
// Touch screen orientation
init.orientation                = DISP_ORIENTATION;

// Touch screen horizontal resolution
init.horizontalResolution       = DISP_HOR_RESOLUTION;

// Touch screen vertical resolution
init.verticalResolution         = DISP_VER_RESOLUTION;


objectHandle = DRV_TOUCH_MTCH6301_Initialize(DRV_TOUCH_INDEX_0,
                                             (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized. Please note this is not the MTCH6301 ID. The hardware MTCH6301 ID is set in the initialization structure. This is the index of the driver index to use.
init	Pointer to a data structure containing any data necessary to initialize the driver. If this pointer is NULL, the driver uses the static initialization override macros for each member of the initialization data structure.

Function

SYS_MODULE_OBJ DRV_TOUCH_MTCH6301_Initialize(const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init)

DRV_TOUCH_MTCH6301_Open Function

Opens the specified MTCH6301 driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
DRV_HANDLE DRV_TOUCH_MTCH6301_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. An error can occur when the following is true:

- if the number of client objects allocated via DRV_TOUCH_MTCH6301_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid

Description

This routine opens the specified MTCH6301 driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The current version of driver does not support the DRV_IO_INTENT feature. The driver is by default non-blocking. The driver can perform both read and write to the MTCH6301 device. The driver supports single client only.

Remarks

The handle returned is valid until the DRV_TOUCH_MTCH6301_Close routine is called. This routine will NEVER block waiting for hardware. If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

The DRV_TOUCH_MTCH6301_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  handle;
```

```
handle = DRV_TOUCH_MTCH6301_Open( DRV_TOUCH_MTCH6301_INDEX_0,
                                  DRV_IO_INTENT_EXCLUSIVE );

if( DRV_HANDLE_INVALID == handle )
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Index of the driver initialized with DRV_TOUCH_MTCH6301_Initialize().
intent	Zero or more of the values from the enumeration DRV_IO_INTENT ORed together to indicate the intended use of the driver. The current version of driver does not support the selective IO intent feature.

Function

DRV_HANDLE DRV_TOUCH_MTCH6301_Open (const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT intent)

DRV_TOUCH_MTCH6301_Status Function

Provides the current status of the MTCH6301 driver module.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
SYS_STATUS DRV_TOUCH_MTCH6301_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system-level operation and cannot start another

Description

This function provides the current status of the MTCH6301 driver module.

Remarks

Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_MODULE_UNINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR.

This function can be used to determine when any of the driver's module level operations has completed.

If the status operation returns SYS_STATUS_BUSY, the previous operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This function will NEVER block waiting for hardware.

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The DRV_TOUCH_MTCH6301_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;  // Returned from DRV_TOUCH_MTCH6301_Initialize
SYS_STATUS          status;

status = DRV_TOUCH_MTCH6301_Status( object );
if( SYS_STATUS_READY != status )
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_TOUCH_MTCH6301_Initialize

Function

SYS_STATUS DRV_TOUCH_MTCH6301_Status (SYS_MODULE_OBJ object)

DRV_TOUCH_MTCH6301_Tasks Function

Maintains the driver's state machine and implements its task queue processing.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
void DRV_TOUCH_MTCH6301_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal state machine and implement its command queue processing. It is always called from SYS_Tasks() function. This routine decodes the touch input data available in drvI2CReadFrameData.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks)

Preconditions

The DRV_TOUCH_MTCH6301_Initialize routine must have been called for the specified MTCH6301 driver instance.

Example

```
SYS_MODULE_OBJ      object;   // Returned from DRV_TOUCH_MTCH6301_Initialize

void SYS_Tasks( void )
{
    DRV_TOUCH_MTCH6301_Tasks ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_TOUCH_MTCH6301_Initialize)

Function

void DRV_TOUCH_MTCH6301_Tasks (SYS_MODULE_OBJ object);

DRV_TOUCH_MTCH6301_ReadRequest Function

Sends a read request to I2C bus driver and adds the read task to queue.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
void DRV_TOUCH_MTCH6301_ReadRequest(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to send a touch input read request to the I2C bus driver and adding the input read decode task to the queue. It is always called from MTCH6301 interrupt ISR routine.

Remarks

This function is normally not called directly by an application. It is called by the MTCH6301 ISR routine.

Preconditions

The DRV_TOUCH_MTCH6301_Initialize routine must have been called for the specified MTCH6301 driver instance.

Example

```
SYS_MODULE_OBJ      object;   // Returned from DRV_TOUCH_MTCH6301_Initialize

void __ISR(_EXTERNAL_INT_VECTOR, ipl5) _IntHandlerDrvMtch6301(void)
{
    DRV_TOUCH_MTCH6301_ReadRequest ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_TOUCH_MTCH6301_Initialize)

Function

void DRV_TOUCH_MTCH6301_ReadRequest(SYS_MODULE_OBJ object)

DRV_TOUCH_MTCH6301_TouchGetX Function

Returns the x coordinate of touch input.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
short DRV_TOUCH_MTCH6301_TouchGetX(uint8_t touchNumber);
```

Returns

It returns the x coordinate of the touch input in terms of number of pixels.

Description

It returns the x coordinate in form of number of pixes for a touch input denoted by touchNumber.

Parameters

Parameters	Description
touchNumber	index to the touch input.

Function

short DRV_TOUCH_MTCH6301_TouchGetX(uint8 touchNumber)

DRV_TOUCH_MTCH6301_TouchGetY Function

Returns the y coordinate of touch input.

Implementation: Dynamic

File

drv_mtch6301.h

C

```
short DRV_TOUCH_MTCH6301_TouchGetY(uint8_t touchNumber);
```

Returns

It returns the y coordinate of the touch input in terms of number of pixels.

Description

It returns the y coordinate in form of number of pixes for a touch input denoted by touchNumber.

Parameters

Parameters	Description
touchNumber	index to the touch input.

Function

short DRV_TOUCH_MTCH6301_TouchGetY(uint8 touchNumber)

DRV_TOUCH_MTCH6301_TouchDataRead Function

Notifies the driver that the current touch data has been read

File

drv_mtch6301.h

C

```
void DRV_TOUCH_MTCH6301_TouchDataRead(const SYS_MODULE_INDEX index);
```

Returns

None.

Description

Notifies the driver that the current touch data has been read

Function

void DRV_TOUCH_MTCH6301_TouchDataRead(const SYS_MODULE_INDEX index)

DRV_TOUCH_MTCH6301_TouchStatus Function

Returns the status of the current touch input.

File

drv_mtch6301.h

C

```
DRV_TOUCH_POSITION_STATUS DRV_TOUCH_MTCH6301_TouchStatus(const SYS_MODULE_INDEX index);
```

Returns

It returns the status of the current touch input.

Description

It returns the status of the current touch input.

Function

DRV_TOUCH_POSITION_SINGLE DRV_TOUCH_MTCH6301_TouchStatus(const SYS_MODULE_INDEX index)

b) Data Types and Constants

DRV_TOUCH_MTCH6301_HANDLE Type

Touch screen controller MTCH6301 driver handle.

File

drv_mtch6301.h

C

```
typedef uintptr_t DRV_TOUCH_MTCH6301_HANDLE;
```

Description

MTCH6301 Driver Handle

Touch controller MTCH6301 driver handle is a handle for the driver client object. Each driver with succesful open call will return a new handle to the client object.

Remarks

None.

DRV_TOUCH_MTCH6301_MODULE_ID Enumeration

Number of valid MTCH6301 driver indices.

File

drv_mtch6301.h

C

```
typedef enum {
  MTCH6301_ID_1 = 0,
  MTCH6301_NUMBER_OF_MODULES
} DRV_TOUCH_MTCH6301_MODULE_ID;
```

Description

MTCH6301 Driver Module Index Count

This constant identifies the number of valid MTCH6301 driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific header files defined as part of the peripheral libraries.

DRV_TOUCH_MTCH6301_HANDLE_INVALID Macro

Definition of an invalid handle.

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_HANDLE_INVALID ((DRV_TOUCH_MTCH6301_HANDLE)(-1))
```

Description

MTCH6301 Driver Invalid Handle

This is the definition of an invalid handle. An invalid handle is is returned by DRV_TOUCH_MTCH6301_Open() and DRV_MTCH6301_Close() functions if the request was not successful.

Remarks

None.

DRV_TOUCH_MTCH6301_I2C_READ_FRAME_SIZE Macro

I2C Frame size for reading MTCH6301 touch input.

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_I2C_READ_FRAME_SIZE 7
```

Description

MTCH6301 Driver Module I2C Frame Size

This constant identifies the size of I2C frame required to read from MTCH6301 touch controller. MTCH6301 notifies the availability of input data through interrupt pin.

Remarks

This constant should be used in place of hard-coded numeric literals. This value is derived from device-specific data sheets.

DRV_TOUCH_MTCH6301_CLIENT_OBJECT Structure

MTCH6301 Driver client object maintaining client data.

File

drv_mtch6301.h

C

```
typedef struct _DRV_MTCH6301_CLIENT_OBJECT {
  DRV_TOUCH_MTCH6301_OBJECT* driverObject;
  DRV_IO_INTENT intent;
  struct DRV_TOUCH_MTCH6301_CLIENT_OBJECT* pNext;
} DRV_TOUCH_MTCH6301_CLIENT_OBJECT;
```

Members

Members	Description
DRV_TOUCH_MTCH6301_OBJECT* driverObject;	Driver Object associated with the client
DRV_IO_INTENT intent;	The intent with which the client was opened
struct DRV_TOUCH_MTCH6301_CLIENT_OBJECT* pNext;	Next driver client object

Description

MTCH6301 Driver client object

This defines the object required for the maintenance of the software clients instance. This object exists once per client instance.

Remarks

None.

DRV_TOUCH_MTCH6301_INDEX_0 Macro

MTCH6301 driver index definitions.

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_INDEX_0 0
```

Description

MTCH6301 Driver Module Index Numbers

These constants provide the MTCH6301 driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_MTCH6301_Initialize and DRV_MTCH6301_Open functions to identify the driver instance in use.

DRV_TOUCH_MTCH6301_INDEX_1 Macro

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_INDEX_1 1
```

Description

This is macro DRV_TOUCH_MTCH6301_INDEX_1.

DRV_TOUCH_MTCH6301_INDEX_COUNT Macro

Number of valid Touch controller MTCH6301 driver indices.

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_INDEX_COUNT 2
```

Description

MTCH6301 Driver Module Index Count

This constant identifies the number of valid Touch Controller MTCH6301 driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals. This value is derived from device-specific header files defined as part of the peripheral libraries.

DRV_TOUCH_MTCH6301_OBJECT Structure

Defines the data structure maintaining MTCH6301 driver instance object.

File

drv_mtch6301.h

C

```
typedef struct {
  SYS_STATUS status;
  int touchId;
  SYS_MODULE_INDEX drvIndex;
  bool inUse;
  bool isExclusive;
  uint8_t numClients;
  INT_SOURCE interruptSource;
  uint16_t orientation;
  uint16_t horizontalResolution;
  uint16_t verticalResolution;
  DRV_HANDLE (* drvOpen)(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
  int32_t readRequest;
  DRV_TOUCH_MTCH6301_TASK_QUEUE* taskQueue;
  DRV_HANDLE drvI2CHandle;
  DRV_TOUCH_POSITION_STATUS touchStatus;
} DRV_TOUCH_MTCH6301_OBJECT;
```

Members

Members	Description
SYS_STATUS status;	The status of the driver
int touchId;	The peripheral Id associated with the object
SYS_MODULE_INDEX drvIndex;	Save the index of the driver. Important to know this as we are using reference based accessing
bool inUse;	Flag to indicate instance in use
bool isExclusive;	Flag to indicate module used in exclusive access mode

uint8_t numClients;	Number of clients possible with the hardware instance
INT_SOURCE interruptSource;	Touch input interrupt source
uint16_t orientation;	Orientation of the display (given in degrees of 0,90,180,270)
uint16_t horizontalResolution;	Horizontal Resolution of the displayed orientation in Pixels
uint16_t verticalResolution;	Vertical Resolution of the displayed orientaion in Pixels
DRV_HANDLE (* drvOpen)(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);	Callback for I2C Driver Open call
int32_t readRequest;	Touch Input read request counter
DRV_TOUCH_MTCH6301_TASK_QUEUE* taskQueue;	Head of the task queue
DRV_HANDLE drvI2CHandle;	I2C bus driver handle
DRV_TOUCH_POSITION_STATUS touchStatus;	Touch status

Description

MTCH6301 Driver Instance Object.

This data structure maintains the MTCH6301 driver instance object. The object exists once per hardware instance.

Remarks

None.

DRV_TOUCH_MTCH6301_TASK_QUEUE Structure

Defines the MTCH6301 Touch Controller driver task data structure.

File

drv_mtch6301.h

C

```
typedef struct {
  bool inUse;
  DRV_TOUCH_MTCH6301_TASK_STATE taskState;
  DRV_I2C_BUFFER_HANDLE drvI2CReadBufferHandle;
  uint8_t drvI2CReadFrameData[DRV_TOUCH_MTCH6301_I2C_READ_FRAME_SIZE];
} DRV_TOUCH_MTCH6301_TASK_QUEUE;
```

Members

Members	Description
bool inUse;	Flag denoting the allocation of task
DRV_TOUCH_MTCH6301_TASK_STATE taskState;	Enum maintaining the task state
DRV_I2C_BUFFER_HANDLE drvI2CReadBufferHandle;	I2C Buffer handle
uint8_t drvI2CReadFrameData[DRV_TOUCH_MTCH6301_I2C_READ_FRAME_SIZE];	Response to Read Touch Input Command • Response = { MTCH6301 Read Address, • Input Data Size, • Touch Id, Pen status, • Touch X coordinate (0 to 6), • Touch X coordinate (7 to 11), • Touch Y coordinate (0 to 6), • Touch Y coordinate (7 to 11) }

Description

MTCH6301 Touch Controller driver task data structure.

This data type defines the data structure maintaing task context in the task queue. The inUse flag denotes the task context allocation for a task. The enum variable taskState maintains the current task state. The I2C buffer handle drvI2CReadBufferHandle maintains the I2C driver buffer handle returned by the I2C driver read request. The byte array variable drvI2CReadFrameData maintains the I2C frame data sent by MTCH6301 after a successful read request.

Remarks

None.

DRV_TOUCH_MTCH6301_TASK_STATE Enumeration

Enumeration defining MTCH6301 touch controller driver task state.

File

drv_mtch6301.h

C

```
typedef enum {
  DRV_TOUCH_MTCH6301_TASK_STATE_INIT = 0,
  DRV_TOUCH_MTCH6301_TASK_STATE_READ_INPUT,
  DRV_TOUCH_MTCH6301_TASK_STATE_DECODE_INPUT,
  DRV_TOUCH_MTCH6301_TASK_STATE_DONE
} DRV_TOUCH_MTCH6301_TASK_STATE;
```

Members

Members	Description
DRV_TOUCH_MTCH6301_TASK_STATE_INIT = 0	Task initialize state
DRV_TOUCH_MTCH6301_TASK_STATE_READ_INPUT	Task read touch input request state
DRV_TOUCH_MTCH6301_TASK_STATE_DECODE_INPUT	Task touch input decode state
DRV_TOUCH_MTCH6301_TASK_STATE_DONE	Task complete state

Description

MTCH6301 Touch Controller Driver Task State

This enumeration defines the MTCH6301 touch controller driver task state. The task state helps to synchronize the operations of initialization the the task, adding the read input task to the task queue once the touch controller notifies the available touch input and a decoding the touch input received.

Remarks

None.

DRV_TOUCH_MTCH6301_I2C_MASTER_READ_ID Macro

MTCH6301 input read, I2C address from where master reads touch input data.

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_I2C_MASTER_READ_ID 0x4B
```

Description

MTCH6301 Driver Module Master Input Read I2C address

This constant defines the MTCH6301 touch input read I2C address. This address is used as I2C address to read Touch input from MTCH6301 Touch controller.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific data sheets.

DRV_TOUCH_MTCH6301_I2C_MASTER_WRITE_ID Macro

MTCH6301 command register write, I2C address where master sends the commands.

File

drv_mtch6301.h

C

```
#define DRV_TOUCH_MTCH6301_I2C_MASTER_WRITE_ID 0x4A
```

Description

MTCH6301 Driver Module Master Command Write I2C Address

This constant defines the MTCH6301 command register I2C write address. This address is used as I2C address to write commands into MTCH6301 Touch controller register.

Remarks

This constant should be used in place of hard-coded numeric literals. This value is derived from device-specific data sheets.

Files

Files

Name	Description
drv_mtch6301.h	Touch controller MTCH6301 Driver interface header file.
drv_mtch6301_config_template.h	MTCH6301 Touch Driver configuration template.

Description

This section lists the source and header files used by the MTCH6301 Touch Driver Library.

drv_mtch6301.h

Touch controller MTCH6301 Driver interface header file.

Enumerations

	Name	Description
	DRV_TOUCH_MTCH6301_MODULE_ID	Number of valid MTCH6301 driver indices.
	DRV_TOUCH_MTCH6301_TASK_STATE	Enumeration defining MTCH6301 touch controller driver task state.

Functions

	Name	Description
	DRV_TOUCH_MTCH6301_Close	Closes an opened instance of the MTCH6301 driver. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Deinitialize	Deinitializes the specified instance of the MTCH6301 driver module. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Initialize	Initializes the MTCH6301 instance for the specified driver index. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Open	Opens the specified MTCH6301 driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_ReadRequest	Sends a read request to I2C bus driver and adds the read task to queue. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Status	Provides the current status of the MTCH6301 driver module. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_Tasks	Maintains the driver's state machine and implements its task queue processing. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_TouchDataRead	Notifies the driver that the current touch data has been read
	DRV_TOUCH_MTCH6301_TouchGetX	Returns the x coordinate of touch input. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_TouchGetY	Returns the y coordinate of touch input. **Implementation:** Dynamic
	DRV_TOUCH_MTCH6301_TouchStatus	Returns the status of the current touch input.

Macros

	Name	Description
	DRV_TOUCH_MTCH6301_HANDLE_INVALID	Definition of an invalid handle.
	DRV_TOUCH_MTCH6301_I2C_MASTER_READ_ID	MTCH6301 input read, I2C address from where master reads touch input data.
	DRV_TOUCH_MTCH6301_I2C_MASTER_WRITE_ID	MTCH6301 command register write, I2C address where master sends the commands.

	DRV_TOUCH_MTCH6301_I2C_READ_FRAME_SIZE	I2C Frame size for reading MTCH6301 touch input.
	DRV_TOUCH_MTCH6301_INDEX_0	MTCH6301 driver index definitions.
	DRV_TOUCH_MTCH6301_INDEX_1	This is macro DRV_TOUCH_MTCH6301_INDEX_1.
	DRV_TOUCH_MTCH6301_INDEX_COUNT	Number of valid Touch controller MTCH6301 driver indices.

Structures

	Name	Description
	_DRV_MTCH6301_CLIENT_OBJECT	MTCH6301 Driver client object maintaining client data.
	DRV_TOUCH_MTCH6301_CLIENT_OBJECT	MTCH6301 Driver client object maintaining client data.
	DRV_TOUCH_MTCH6301_OBJECT	Defines the data structure maintaining MTCH6301 driver instance object.
	DRV_TOUCH_MTCH6301_TASK_QUEUE	Defines the MTCH6301 Touch Controller driver task data structure.

Types

	Name	Description
	DRV_TOUCH_MTCH6301_HANDLE	Touch screen controller MTCH6301 driver handle.

Description

Touch Controller MTCH6301 Driver Interface File

This header file describes the macros, data structure and prototypes of the touch controller MTCH6301 driver interface.

File Name

drv_mtch6301.c

drv_mtch6301_config_template.h

MTCH6301 Touch Driver configuration template.

Macros

	Name	Description
	DRV_MTCH6301_CALIBRATION_DELAY	Defines the calibration delay.
	DRV_MTCH6301_CALIBRATION_INSET	Defines the calibration inset.
	DRV_MTCH6301_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_MTCH6301_INDEX	MTCH6301 static index selection.
	DRV_MTCH6301_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_MTCH6301_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_MTCH6301_SAMPLE_POINTS	Define the sample points.
	DRV_MTCH6301_TOUCH_DIAMETER	Defines the touch diameter.

Description

MTCH6301 Touch Driver Configuration Template

This header file contains the build-time configuration selections for the MTCH6301 Touch Driver. This is the template file which give all possible configurations that can be made. This file should not be included in any project.

File Name

drv_mtch6301_config_template.h

Company

Microchip Technology Inc.

MTCH6303 Touch Driver Library

This topic describes the MTCH6303 Touch Driver Library.

Introduction

This library provides an interface to manage the MTCH6303 Touch Driver module on the Microchip family of microcontrollers in different modes of operation.

Description

The MPLAB Harmony MTCH6303 Touch Driver provides a high-level interface to the MTCH6303 touch controller device. This driver provides application routines to read the touch input data from the touch screen. The MTCH6303 device can notify the availability of touch input data through external interrupt. The MTCH6303 driver allows the application to map a controller pin as an external interrupt pin.

Currently, the MTCH6303 Touch Driver only supports non-gestural single-finger touch screen input.

Using the Library

This topic describes the basic architecture of the MTCH6303 Touch Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_mtch6303_static.h`

The interface to the MTCH6303 Touch Driver Library is defined in the `drv_mtch6303_static.h` header file. This file is generated by the MPLAB Harmony Configurator (MHC) during application code generation. It is included in `system_definitions.h` by MHC during application code generation. Any configuration macros required for MTCH6303 Driver are included in `system_config.h` by MHC during code generation. Any C language source (`.c`) file that uses the MTCH6303 Touch Driver Library should include `system_config.h` and `system_definitions.h`, respectively.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the MTCH6303 Touch Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

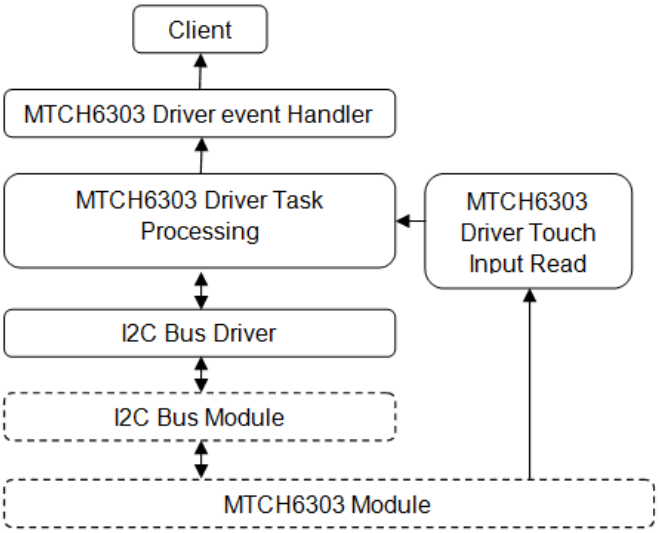
The MTCH6303 Touch Driver has routines to perform the following operations:

- MTCH6303 register read and write
- MTCh6303 message read and write
- MTCH6303 touch input read
- Mapping of the touch input to screen resolution

The driver Initialization routine allows the application to initialize the driver. The driver must be initialized before it can be used by the application. Once the driver is initialized, the driver Open function allows retrieval of the client handle. If the client handle is valid, an event handler routine needs to be registered by the application. The MTCH6303 Touch Driver triggers an interrupt once touch input is available to be read from the MTCH6303 registers. A touch input Read function is called from the interrupt handler to initiate the touch input read task. An Event Handler function is called once the touch input read task is completed. A valid touch input will be available only after the event handler routine is triggered. The touch input must be read inside of the event handler function.

The touch input data is a raw value and needs to be mapped to the target screen resolution. At zero degree orientation, touch input is mapped on the x axis from zero at the left and the maximum value at the right. At zero degree orientation, touch input is mapped on the y axis from zero at the top and the maximum value at the bottom.

MTCH6303 Driver Abstraction Model



Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the MTCH6303 Touch Driver.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks, and status functions.
Client Setup Functions	Provides open, close, status, and other setup function.
Read and Write Functions	Provides functions to read and write to the MTCH6303 registers, messages, and touch data.
Miscellaneous Functions	Provides miscellaneous functions.

How the Library Works

The library provides interfaces to support:

- System functions, which provide system module interfaces, device initialization, deinitialization, task, and status functions
- Client setup functions, which provide client interfaces such as open, close and event handler registration
- Read and write functions, initiate the touch input or register or message read and write tasks
- Miscellaneous functions such as touch input map function are provided to process the raw touch input data

Configuring the Library

The configuration of the MTCH6303 Touch Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the MTCH6303 Touch Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the MTCH6303 Touch Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Building the Library

This section lists the files that are available in the MTCH6303 Touch Driver Library.

Description

This section list the files that are available in the `/src` folder of the MTCH6303 Touch Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/touch/mtch6303`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_mtch6303_static.h`	Header file that exports the driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/drv_mtch6303_static.c`	Basic MTCH6303 Touch Driver implementation file.
`/src/drv_mtch6303_buffer_queue_i2c_static.c`	MTCH6303 I2C buffer queue implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
/drv_mtch6303_buffer_queue_touch_static.c	MTCH6303 message buffer queue implementation file.

Module Dependencies

The MTCH6303 Touch Driver Library depends on the following modules:

- Interrupt System Service Library
- Ports System Service Library
- I2C Driver Library

Library Interface

a) System Functions

	Name	Description
	DRV_MTCH6303_Deinitialize	Deinitializes the instance of the MTCH6303 driver module.
	DRV_MTCH6303_Initialize	Initializes the MTCH6303 static single instance.
	DRV_MTCH6303_Status	Gets the current status of the MTCH6303 driver module.
	DRV_MTCH6303_Tasks	Maintains the driver's register read/write state machine and implements its ISR.

b) Client Setup Functions

	Name	Description
	DRV_MTCH6303_Close	Closes an opened-instance of the MTCH6303 driver.
	DRV_MTCH6303_ErrorGet	This function returns the error associated with the last client request.
	DRV_MTCH6303_Open	Opens the MTCH6303 driver instance and returns a handle to it.

c) Read and Write Functions

	Name	Description
	DRV_MTCH6303_AddRegisterRead	Schedules a non-blocking register read request to read I2C accessible MTCH6303 registers.
	DRV_MTCH6303_AddRegisterWrite	Schedule a non-blocking driver register write operation to write I2C accessible MTCH6303 registers.
	DRV_MTCH6303_TOUCH_AddMessageCommandWrite	Schedule a non-blocking driver command message write operation to write command message to MTCH6303 registers.
	DRV_MTCH6303_TOUCH_AddMessageReportRead	Schedules a non-blocking report message read request to read the report message from MTCH6303 device.
	DRV_MTCH6303_TOUCH_AddTouchInputRead	Schedules a non-blocking read buffer request to read touch input from MTCH6303.
	DRV_MTCH6303_TOUCH_BufferEventHandlerSet	Allows a client to identify a buffer event handling function for the driver to call back when queued message transfers have finished.
	DRV_MTCH6303_TOUCH_Tasks	Maintains the driver's message state machine and implements its ISR.
	DRV_MTCH6303_TouchInputMap	Maps the raw touch input to display resolution.
	DRV_MTCH6303_TouchInputRead	Schedules a non-blocking read buffer request to read touch input from MTCH6303.
	DRV_MTCH6303_BufferEventHandlerSet	Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

d) Data Types and Constants

	Name	Description
	DRV_MTCH6303_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_MTCH6303_TOUCH_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_MTCH6303_TOUCH_NUM_INPUTS	Definition of number of touch input packets can be identified by MTCH6303.
	DRV_MTCH6303_BUFFER_EVENT	Lists the different conditions that happens during a buffer transfer.
	DRV_MTCH6303_BUFFER_EVENT_HANDLER	Points to a callback after completion of an register read -write or message stream read - write.
	DRV_MTCH6303_BUFFER_HANDLE	Handle identifying a read or write buffer passed to the driver.

	DRV_MTCH6303_CLIENT_STATUS	Defines the client-specific status of the MTCH6303 driver.
	DRV_MTCH6303_ERROR	Defines the possible errors that can occur during driver operation.
	DRV_MTCH6303_TOUCH_BUFFER_EVENT	Lists the different conditions that happens during a touch message buffer transfer.
	DRV_MTCH6303_TOUCH_BUFFER_EVENT_HANDLER	Points to a callback after completion of an message report read or message command write.
	DRV_MTCH6303_TOUCH_BUFFER_HANDLE	Handle identifying a read or write touch message buffer passed to the driver.
	DRV_MTCH6303_TOUCH_DATA	Defines MTCH6303 I2C Touch Data
	DRV_MTCH6303_TOUCH_INPUT	Defines MTCH6303 Touch Input Packet
	DRV_MTCH6303_TOUCH_MESSAGE	Defines MTCH6303 Touch Message.
	DRV_MTCH6303_TOUCH_MESSAGE_HEADER	Defines Touch Message Header.
	DRV_MTCH6303_TOUCH_NIBBLE_0	Defines the I2C Nibble 0 of MTCH6303 Touch input packet.
	DRV_MTCH6303_TOUCH_STATUS	Defines the I2C touch status register bits
	DRV_TOUCH_MTCH6303_MSG_ID	List of report or command message identification.
	DRV_TOUCH_MTCH6303_I2C_REGISTER_MAP	List of MTCH6303 I2C Accessible Register Identification.

Description

This section describes the API functions of the MTCH6303 Touch Driver library.

Refer to each section for a detailed description.

a) System Functions

DRV_MTCH6303_Deinitialize Function

Deinitializes the instance of the MTCH6303 driver module.

File

drv_mtch6303.h

C

```
void DRV_MTCH6303_Deinitialize();
```

Returns

None.

Description

Deinitializes the instance of the MTCH6303 driver module, disabling its operation. Invalidates all the internal data.

Remarks

once the initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. this routine will NEVER block waiting for hardware.

Preconditions

Function DRV_MTCH6303_Initialize should have been called before calling this function.

Example

```
SYS_STATUS    status;

DRV_MTCH6303_Deinitialize();

status = DRV_MTCH6303_Status();
if(SYS_MODULE_DEINITIALIZED != status)
{
    //check again later if you need to know
    //when the driver is deinitialized
}
```

Function

void DRV_MTCH6303_Deinitialize(void)

DRV_MTCH6303_Initialize Function

Initializes the MTCH6303 static single instance.

File

drv_mtch6303.h

C

```
SYS_MODULE_OBJ DRV_MTCH6303_Initialize();
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the MTCH6303 static driver instance. It makes the instance ready for a client to open and use it. The instance parameters are initialized by values set by MPLAB Harmony Configurator.

Preconditions

None.

Example

```
// The following code snippet shows an example MTCH6303 driver initialization.

SYS_MODULE_OBJ    objectHandle;

objectHandle = DRV_MTCH6303_Initialize();
if( SYS_MODULE_OBJ_INVALID == objectHandle )
{
   // Handle error
}
Remarks:   This routine must be called before any other MTCH6303 routine is called.

   This routine should only be called once during system initialization unless
   DRV_MTCH6303_Deinitialize is called to deinitialize the driver instance.
   This routine will NEVER block for hardware access.
```

Function

SYS_MODULE_OBJ DRV_MTCH6303_Initialize (void)

DRV_MTCH6303_Status Function

Gets the current status of the MTCH6303 driver module.

File

drv_mtch6303.h

C

```
SYS_STATUS DRV_MTCH6303_Status();
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system level operation and cannot start another.

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized.

Description

This routine provides the current status of the MTCH6303 driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_MTCH6303_Initialize should have been called before calling this function.

Example

```
SYS_STATUS mtch6303Status;

mtch6303Status = DRV_MTCH6303_Status();
if(SYS_STATUS_READY == mtch6303Status)
{
    // This means the driver can be opened using the
    // DRV_MTCH6303_Open() function.
}
```

Function

SYS_STATUS DRV_MTCH6303_Status(void)

DRV_MTCH6303_Tasks Function

Maintains the driver's register read/write state machine and implements its ISR.

File

drv_mtch6303.h

C

```
void DRV_MTCH6303_Tasks();
```

Returns

None.

Description

This routine is used to maintain the driver's register read/write state machine and implement its ISR for interrupt-driven implementations. In interrupt mode, this function is called in I2C Driver event Handler routine. The I2C Driver event Handler routine is registered by MTCH6303 event Handler register routine.

Remarks

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

Function DRV_MTCH6303_Initialize should have been called before calling this function. It also needs registration of the MTCH6303 Driver event handler routine.

Function

void DRV_MTCH6303_Tasks(void)

b) Client Setup Functions

DRV_MTCH6303_Close Function

Closes an opened-instance of the MTCH6303 driver.

File

drv_mtch6303.h

C

```
DRV_MTCH6303_CLIENT_STATUS DRV_MTCH6303_Close();
```

Returns

DRV_MTCH6303_CLIENT_STATUS_ERROR - if driver fails to remove buffer objects from queue.

DRV_MTCHC6303_CLIENT_STATUS_CLOSED - client is successfully closed

Description

This routine closes an opened-instance of the MTCH6303 driver. Any buffers in the driver queue that were submitted by this client will be removed. DRV_MTCH6303_Open must be called to before using the driver again.

Remarks

The driver will abort any ongoing operations when this routine is called.

Preconditions

The DRV_MTCH6303_Initialize routine must have been called. DRV_MTCH6303_Open must have been called.

Example

```
DRV_MTH6303_CLIENT_STATUS mtch6303Status;

mtch6303Status = DRV_MTCH6303_Close()
if( DRV_MTCH6303_CLIENT_STATUS_ERROR == mtch6303Status )
{
    //retry closing the driver client
}
```

Function

DRV_MTCH6303_CLIENT_STATUS DRV_MTCH6303_Close (void)

DRV_MTCH6303_ErrorGet Function

This function returns the error associated with the last client request.

File

drv_mtch6303.h

C

```
DRV_MTCH6303_ERROR DRV_MTCH6303_ErrorGet();
```

Returns

DRV_MTCH6303_ERROR_NONE - no error

Description

This function returns the error associated with the last client request.

Remarks

This routine always return DRV_MTCH6303_ERROR_NONE the client error is currently not updated by any of the MTCH6303 operations API's.

Preconditions

The DRV_MTCH6303_Initialize routine must have been called. DRV_MTCH6303_Open must have been called to open a device client.

Function

DRV_MTCH6303_ERROR DRV_MTCH6303_ErrorGet (void)

DRV_MTCH6303_Open Function

Opens the MTCH6303 driver instance and returns a handle to it.

File

drv_mtch6303.h

C

```
DRV_HANDLE DRV_MTCH6303_Open();
```

Returns

If successful, the routine returns a valid open-instance handle. If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the driver is not ready to be opened, typically when the initialize routine has not completed execution.
- if the bus driver fails to open
- if the client is trying to open the driver but driver has been opened exclusively by another client.

Description

This routine opens the specified MTCH6303 driver instance and provides a handle.

Remarks

The handle returned is valid until the DRV_MTCH6303_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application.

Preconditions

Function DRV_MTCH6303_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_MTCH6303_Open( );
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Function

DRV_HANDLE DRV_MTCH6303_Open { void }

c) Read and Write Functions

DRV_MTCH6303_AddRegisterRead Function

Schedules a non-blocking register read request to read I2C accessible MTCH6303 registers.

File

drv_mtch6303.h

C

```
void DRV_MTCH6303_AddRegisterRead(DRV_MTCH6303_BUFFER_HANDLE * bufferHandle, uint8_t source, size_t nBytes,
uint8_t * destination);
```

Returns

None.

Description

This function schedules a non-blocking register read request to read I2C accessible MTCH6303 registers. The function returns with a valid buffer handle in the bufferHandle argument if the register read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_MTCH6303_BUFFER_HANDLE_INVALID in the bufferHandle argument:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient.
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_MTCH6303_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully or DRV_MTCH6303_BUFFER_EVENT_ERROR event if the buffer was not processed successfully. The register data is collected into destination and can be read once a buffer event complete is reported. A event handler is called on buffer event complete where the register data must be read from destination.

Preconditions

The DRV_MTCH6303_Initialize routine must have been called and the DRV_MTCH6303_Status must have returned SYS_STATUS_READY.

DRV_MTCH6303_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t registerData[NUM_REGISTERS];
DRV_MTCH6303_BUFFER_HANDLE bufferHandle;

// Client registers an event handler with driver
```

```
DRV_MTCH6303_BufferEventHandlerSet( APP_MTCH6303BufferEventHandler,
                                    (uintptr_t)&myAppObj);

DRV_MTCH6303_AddRegisterRead( &bufferHandle,
                              DRV_MTCH6303_REG_TOUCH_STATUS,
                              NUM_REGISTERS,
                              &registerData );

if(DRV_MTCH6303_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_MTCH6303BufferEventHandler( DRV_MTCH6303_BUFFER_EVENT event,
                                     DRV_MTCH6303_BUFFER_HANDLE bufferHandle,
                                     uintptr_t contextHandle )
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_MTCH6303_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_MTCH6303_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
Remarks:   None.
```

Parameters

Parameters	Description
bufferHandle	Handle to the buffer scheduled.
source	Register index.
nBytes	Number of registers to be read, starting from source.
destination	buffer collecting register data.

Function

void DRV_MTCH6303_AddRegisterRead(DRV_MTCH6303_BUFFER_HANDLE * bufferHandle,

uint8_t source,

size_t nBytes,

uint8_t * destination)

DRV_MTCH6303_AddRegisterWrite Function

Schedule a non-blocking driver register write operation to write I2C accessible MTCH6303 registers.

File

drv_mtch6303.h

C

```
void DRV_MTCH6303_AddRegisterWrite(DRV_MTCH6303_BUFFER_HANDLE * bufferHandle, uint8_t destination, size_t
nBytes, uint8_t * source);
```

Returns

None.

Description

This function schedules a non-blocking register write request to write I2C accessible MTCH6303 registers. The function returns with a valid buffer handle in the bufferHandle argument if the register write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_MTCH6303_BUFFER_HANDLE_INVALID in the bufferHandle argument:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0
- if the write queue size is full or queue depth is insufficient.
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_MTCH6303_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully or DRV_MTCH6303_BUFFER_EVENT_ERROR event if the buffer was not processed successfully. A event handler is called on buffer event complete where the application data is written to the I2C accessible MTCH6303 Register.

Remarks

None.

Preconditions

The DRV_MTCH6303_Initialize routine must have been called and the DRV_MTCH6303_Status must have returned SYS_STATUS_READY.

DRV_MTCH6303_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
uint8_t registerData[NUM_REGISTERS];
DRV_MTCH6303_BUFFER_HANDLE bufferHandle;

// Client registers an event handler with driver

DRV_MTCH6303_BufferEventHandlerSet( APP_MTCH6303BufferEventHandler,
                                    (uintptr_t)&myAppObj);

DRV_MTCH6303_AddRegisterWrite( &bufferHandle,
                               DRV_MTCH6303_REG_TOUCH_STATUS,
                               NUM_REGISTERS,
                               &registerData );

if(DRV_MTCH6303_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_MTCH6303BufferEventHandler( DRV_MTCH6303_BUFFER_EVENT event,
                                     DRV_MTCH6303_BUFFER_HANDLE bufferHandle,
                                     uintptr_t contextHandle )
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_MTCH6303_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_MTCH6303_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;
```

```
            default:
                break;
        }
    }
```

Parameters

Parameters	Description
bufferHandle	Pointer to an argument that will contain the return buffer handle.
destination	Index to the start of destination register list.
nBytes	number of registers.
source	pointer to the data to be written to the register.

Function

void DRV_MTCH6303_AddRegisterWrite(DRV_MTCH6303_BUFFER_HANDLE * bufferHandle,

uint8_t destination,

size_t nBytes,

uint8_t * source)

DRV_MTCH6303_TOUCH_AddMessageCommandWrite Function

Schedule a non-blocking driver command message write operation to write command message to MTCH6303 registers.

File

drv_mtch6303.h

C

```
void DRV_MTCH6303_TOUCH_AddMessageCommandWrite(DRV_MTCH6303_TOUCH_BUFFER_HANDLE * bufferHandle,
DRV_MTCH6303_TOUCH_MESSAGE * messageCmd, size_t messageSize);
```

Returns

None.

Description

This function schedules a non-blocking command message write request to write command message to MTCH6303. The function returns with a valid buffer handle in the bufferHandle argument if the register command message write request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. While the request is in the queue, the application message buffer is owned by the driver and should not be modified. The function returns DRV_MTCH6303_TOUCH_BUFFER_HANDLE_INVALID in the bufferHandle argument:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0
- if the message write queue size is full or queue depth is insufficient.
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_MTCH6303_TOUCH_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully or DRV_MTCH6303_TOUCH_BUFFER_EVENT_ERROR event if the buffer was not processed successfully. A event handler is called on buffer event complete where the application command message is written to MTCH6303.

Remarks

None.

Preconditions

The DRV_MTCH6303_Initialize routine must have been called and the DRV_MTCH6303_Status must have returned SYS_STATUS_READY.

DRV_MTCH6303_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
DRV_MTCH6303_TOUCH_MESSAGE messageCommand;
DRV_MTCH6303_TOUCH_BUFFER_HANDLE bufferHandle;

// Client registers an event handler with driver
```

```
DRV_MTCH6303_TOUCH_BufferEventHandlerSet( APP_MTCH6303BufferEventHandler,
                                          (uintptr_t)&myAppObj);

DRV_MTCH6303_TOUCH_AddMessageCommandWrite( &bufferHandle,
                                           &messageCommand,
                                           MY_MESSAGE_SIZE );

if(DRV_MTCH6303_TOUCH_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_MTCH6303BufferEventHandler( DRV_MTCH6303_TOUCH_BUFFER_EVENT event,
                                     DRV_MTCH6303_TOUCH_BUFFER_HANDLE bufferHandle,
                                     uintptr_t contextHandle )
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_MTCH6303_TOUCH_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_MTCH6303_TOUCH_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
bufferHandle	Pointer to an argument that will contain the return buffer handle.
messageCmd	command message to write to MTCH6303.
messageSize	command message size. It includes message header and payload size.

Function

void DRV_MTCH6303_TOUCH_AddMessageCommandWrite
(DRV_MTCH6303_TOUCH_BUFFER_HANDLE * bufferHandle,
   DRV_MTCH6303_TOUCH_MESSAGE * messageCmd,
size_t messageSize)

DRV_MTCH6303_TOUCH_AddMessageReportRead Function

Schedules a non-blocking report message read request to read the report message from MTCH6303 device.

File

drv_mtch6303.h

C

```
void DRV_MTCH6303_TOUCH_AddMessageReportRead(DRV_MTCH6303_TOUCH_BUFFER_HANDLE * bufferHandle,
DRV_MTCH6303_TOUCH_MESSAGE * messageRep, size_t messageSize);
```

Returns

None.

Description

This function schedules a non-blocking report message read request to read the report message from MTCH6303 device. The function returns with a valid buffer handle in the bufferHandle argument if the register read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_MTCH6303_TOUCH_BUFFER_HANDLE_INVALID in the bufferHandle argument:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient.
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_MTCH6303_TOUCH_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully or DRV_MTCH6303_TOUCH_BUFFER_EVENT_ERROR event if the buffer was not processed successfully. The register data is collected into destination and can be read once a buffer event complete is reported. A event handler is called on buffer event complete where the register data must be read from destination.

Remarks

None.

Preconditions

The DRV_MTCH6303_Initialize routine must have been called and the DRV_MTCH6303_Status must have returned SYS_STATUS_READY.

DRV_MTCH6303_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
DRV_MTCH6303_TOUCH_MESSAGE messageReport;
DRV_MTCH6303_TOUCH_BUFFER_HANDLE bufferHandle;

// Client registers an event handler with driver

DRV_MTCH6303_TOUCH_BufferEventHandlerSet( APP_MTCH6303BufferEventHandler,
                                          (uintptr_t)&myAppObj);

DRV_MTCH6303_TOUCH_AddMessageReportRead( &bufferHandle,
                                         &messageReport,
                                         MY_MESSAGE_SIZE );

if(DRV_MTCH6303_TOUCH_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_MTCH6303BufferEventHandler( DRV_MTCH6303_TOUCH_BUFFER_EVENT event,
                                     DRV_MTCH6303_TOUCH_BUFFER_HANDLE bufferHandle,
                                     uintptr_t contextHandle )
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_MTCH6303_TOUCH_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_MTCH6303_TOUCH_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
```

```
            break;
        }
    }
```

Parameters

Parameters	Description
bufferHandle	Handle to the buffer scheduled.
messageRep	report message buffer.
messageSize	report message size. It includes message header and payload size.

Function

void DRV_MTCH6303_TOUCH_AddMessageReportRead
(DRV_MTCH6303_TOUCH_BUFFER_HANDLE * bufferHandle,
DRV_MTCH6303_TOUCH_MESSAGE * messageRep,
size_t messageSize)

DRV_MTCH6303_TOUCH_AddTouchInputRead Function

Schedules a non-blocking read buffer request to read touch input from MTCH6303.

File

drv_mtch6303.h

C

```
void DRV_MTCH6303_TOUCH_AddTouchInputRead(DRV_MTCH6303_TOUCH_BUFFER_HANDLE * bufferHandle,
DRV_MTCH6303_TOUCH_DATA * touchData);
```

Returns

None.

Description

This function schedules a non-blocking read buffer request to read touch input from MTCH6303. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_MTCH6303_TOUCH_BUFFER_HANDLE_INVALID in the bufferHandle argument:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient.
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_MTCH6303_TOUCH_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully or DRV_MTCH6303_TOUCH_BUFFER_EVENT_ERROR event if the buffer was not processed successfully. The touch data is collected into touchData and can be read once a buffer event complete is reported. A event handler is called on buffer event complete where the touch data must be read from touchData.

Remarks

None.

Preconditions

The DRV_MTCH6303_Initialize routine must have been called and the DRV_MTCH6303_Status must have returned SYS_STATUS_READY.

DRV_MTCH6303_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
DRV_MTCH6303_TOUCH_DATA touchData;
DRV_MTCH6303_BUFFER_HANDLE bufferHandle;

// Client registers an event handler with driver

DRV_MTCH6303_TOUCH_BufferEventHandlerSet( APP_MTCH6303BufferEventHandler,
                                          (uintptr_t)&myAppObj);
```

```
DRV_MTCH6303_TOUCH_AddTouchInputRead( &bufferHandle, &touchData );

if(DRV_MTCH6303_TOUCH_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_MTCH6303BufferEventHandler( DRV_MTCH6303_TOUCH_BUFFER_EVENT event,
                                     DRV_MTCH6303_TOUCH_BUFFER_HANDLE bufferHandle,
                                     uintptr_t contextHandle )
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_MTCH6303_TOUCH_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_MTCH6303_TOUCH_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
bufferHandle	Handle to the buffer scheduled.
touchData	Buffer collecting touch data.

Function

void DRV_MTCH6303_TOUCH_AddTouchInputRead
(DRV_MTCH6303_TOUCH_BUFFER_HANDLE * bufferHandle,
DRV_MTCH6303_TOUCH_DATA * touchData)

DRV_MTCH6303_TOUCH_BufferEventHandlerSet Function

Allows a client to identify a buffer event handling function for the driver to call back when queued message transfers have finished.

File

drv_mtch6303.h

C

```
void DRV_MTCH6303_TOUCH_BufferEventHandlerSet(const DRV_MTCH6303_TOUCH_BUFFER_EVENT_HANDLER eventHandler,
const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify a message event handling function for the driver to call back when queued message transfers have finished. When a client calls either the DRV_MTCH6303_TOUCH_AddTouchInputRead, DRV_MTCH6303_TOUCH_AddMessageReportRead or DRV_MTCH6303_TOUCH_AddMessageCommandWrite function, it is provided with a handle identifying the message that was added to the driver's message queue. The driver will pass this handle back to the client by calling "eventHandler" function when the message transfer has completed.

The event handler should be set before the client performs any "message add" operations that could generate events. The event handler once set,

persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

None.

Preconditions

The DRV_MTCH6303_Initialize routine must have been called and the DRV_MTCH6303_Status must have returned SYS_STATUS_READY.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;
DRV_MTCH6303_TOUCH_MESSAGE messageReport;
DRV_MTCH6303_TOUCH_BUFFER_HANDLE bufferHandle;

// myMTCH6303Handle is the handle returned
// by the DRV_MTCH6303_Open function.

// Client registers an event handler with driver. This is done once

DRV_MTCH6303_TOUCH_BufferEventHandlerSet( APP_MTCH6303BufferEventHandler,
                                (uintptr_t)&myAppObj );

DRV_MTCH6303_TOUCH_AddMessageReportRead( &bufferHandle, &messageReport );

if(DRV_MTCH6303_TOUCH_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_MTCH6303BufferEventHandler( DRV_MTCH6303_TOUCH_BUFFER_EVENT event,
                                     DRV_MTCH6303_TOUCH_BUFFER_HANDLE bufferHandle,
                                     uintptr_t contextHandle )
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_MTCH6303_TOUCH_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_MTCH6303_TOUCH_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_MTCH6303_TOUCH_BufferEventHandlerSet

(

const DRV_MTCH6303_TOUCH_BUFFER_EVENT_HANDLER eventHandler,
const uintptr_t context
)

DRV_MTCH6303_TOUCH_Tasks Function

Maintains the driver's message state machine and implements its ISR.

File

drv_mtch6303.h

C

```
void DRV_MTCH6303_TOUCH_Tasks();
```

Returns

None.

Description

This routine is used to maintain the driver's message state machine and implement its ISR for interrupt-driven implementations. In interrupt mode, this function is called in I2C Driver event Handler routine. The I2C Driver event Handler routine is registered by MTCH6303 Touch event Handler register routine.

Remarks

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

Function DRV_MTCH6303_Initialize should have been called before calling this function. It also needs registration of the MTCH6303 Driver Touch event handler routine.

Function

void DRV_MTCH6303_TOUCH_Tasks(void)

DRV_MTCH6303_TouchInputMap Function

Maps the raw touch input to display resolution.

File

drv_mtch6303.h

C

```
inline uint16_t DRV_MTCH6303_TouchInputMap(uint16_t touchValue, uint16_t dispResolution);
```

Returns

This function returns the raw touch input mapped to display resolution in form of number of pixels.

Description

This function maps the raw touch input to display resolution. Raw touch input touchValue is obtained from the individual x or y value of DRV_MTCH6303_TOUCH_DATA. Raw touch value varies from 0 to 0x7FFF. The displayResolution is either horizontal or vertical resolution of the display in pixels. The function returns the raw touch input mapped to display resolution in form of number of pixels.

Remarks

None.

Preconditions

None.

Example

```
// Display with resolution 800 x 480
#define DISP_HOR_RESOUTION  800
#define DISP_VER_RESOLUTION 480

DRV_MTCH6303_TOUCH_DATA touchData;
uint16_t rawTouchX;
```

```
uint16_t rawTouchY;
uint16_t touchX;
uint16_t touchY;

// map 0th touch packet to display resolution
rawTouchX = touchData.touch[0].x;
rawTouchY = touchData.touch[0].y;

// map raw touch input in x direction to display horizontal resolution
touchX = DRV_MTCH6303_TouchInputMap( rawTouchX, DISP_HOR_RESOLUTION );

// map raw touch input in y direction to display vertical resolution
touchY = DRV_MTCH6303_TouchInputMap( rawTouchY, DISP_VER_RESOLUTION );

// use touchX and touchY as input to graphics objects.
```

Parameters

Parameters	Description
touchValue	raw touch input either in x or y direction (0 - 0x7FFF).
dispResolution	display resolution specifying either width or height of the display in pixels.

Function

uint16_t DRV_MTCH6303_TouchInputMap(uint16_t touchValue, uint16_t dispResolution)

DRV_MTCH6303_TouchInputRead Function

Schedules a non-blocking read buffer request to read touch input from MTCH6303.

File

drv_mtch6303.h

C

```
void DRV_MTCH6303_TouchInputRead(DRV_MTCH6303_BUFFER_HANDLE * bufferHandle, DRV_MTCH6303_TOUCH_DATA *
touchData);
```

Returns

None.

Description

This function schedules a non-blocking read buffer request to read touch input from MTCH6303. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance queue and returns immediately. The function returns DRV_MTCH6303_BUFFER_HANDLE_INVALID in the bufferHandle argument:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient.
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_MTCH6303_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully or DRV_MTCH6303_BUFFER_EVENT_ERROR event if the buffer was not processed successfully. The touch data is collected into touchData and can be read once a buffer event complete is reported. A event handler is called on buffer event complete where the touch data must be read from touchData.

Remarks

None.

Preconditions

The DRV_MTCH6303_Initialize routine must have been called and the DRV_MTCH6303_Status must have returned SYS_STATUS_READY.

DRV_MTCH6303_Open must have been called to obtain a valid opened device handle.

Example

```
MY_APP_OBJ myAppObj;
DRV_MTCH6303_TOUCH_DATA touchData;
DRV_MTCH6303_BUFFER_HANDLE bufferHandle;
```

```
// Client registers an event handler with driver

DRV_MTCH6303_BufferEventHandlerSet( APP_MTCH6303BufferEventHandler,
                                    (uintptr_t)&myAppObj);

DRV_MTCH6303_TouchInputRead( &bufferHandle, &touchData );

if(DRV_MTCH6303_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_MTCH6303BufferEventHandler( DRV_MTCH6303_BUFFER_EVENT event,
                                     DRV_MTCH6303_BUFFER_HANDLE bufferHandle,
                                     uintptr_t contextHandle )
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_MTCH6303_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_MTCH6303_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
bufferHandle	Handle to the buffer scheduled.
touchData	Buffer collecting touch data.

Function

void DRV_MTCH6303_TouchInputRead(DRV_MTCH6303_BUFFER_HANDLE * bufferHandle,
DRV_MTCH6303_TOUCH_DATA * touchData)

DRV_MTCH6303_BufferEventHandlerSet Function

Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

File

drv_mtch6303.h

C

```
void DRV_MTCH6303_BufferEventHandlerSet(const DRV_MTCH6303_BUFFER_EVENT_HANDLER eventHandler, const
uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls either the DRV_MTCH6303_TouchInputRead, DRV_MTCH6303_AddRegisterRead or DRV_MTCH6303_AddRegisterWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by

calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

None.

Preconditions

The DRV_MTCH6303_Initialize routine must have been called and the DRV_MTCH6303_Status must have returned SYS_STATUS_READY.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];

// myMTCH6303Handle is the handle returned
// by the DRV_MTCH6303_Open function.

// Client registers an event handler with driver. This is done once

DRV_MTCH6303_BufferEventHandlerSet( APP_MTCH6303BufferEventHandle,
                                    (uintptr_t)&myAppObj );

DRV_MTCH6303_AddRegisterRead( &bufferHandle
                              DRV_MTCH6303_REG_TOUCH_STATUS,
                              MY_BUFFER_SIZE,
                              &mybuffer);

if(DRV_MTCH6303_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_MTCH6303BufferEventHandle( DRV_MTCH6303_BUFFER_EVENT event,
                                    DRV_MTCH6303_BUFFER_HANDLE handle,
                                    uintptr_t context)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_MTCH6303_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_MTCH6303_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
eventHandler	Pointer to the event handler function.

context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_MTCH6303_BufferEventHandlerSet
(
const DRV_MTCH6303_BUFFER_EVENT_HANDLER eventHandler,
const uintptr_t context
)

d) Data Types and Constants

DRV_MTCH6303_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_mtch6303.h

C

```
#define DRV_MTCH6303_BUFFER_HANDLE_INVALID
```

Description

MTCH6303 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_MTCH6303_AddRegisterRead, DRV_MTCH6303_AddRegisterWrite or DRV_MTCH6303_TouchInputRead functions if the request was not successful.

Remarks

None

DRV_MTCH6303_TOUCH_BUFFER_HANDLE_INVALID Macro

Definition of an invalid buffer handle.

File

drv_mtch6303.h

C

```
#define DRV_MTCH6303_TOUCH_BUFFER_HANDLE_INVALID
```

Description

MTCH6303 Driver Invalid Buffer Handle

This is the definition of an invalid buffer handle. An invalid buffer handle is returned by DRV_MTCH6303_TOUCH_AddMessageReportRead, DRV_MTCH6303_TOUCH_AddMessageCommandWrite or DRV_MTCH6303_TOUCH_AddTouchInputRead functions if the request was not successful.

Remarks

None

DRV_MTCH6303_TOUCH_NUM_INPUTS Macro

Definition of number of touch input packets can be identified by MTCH6303.

File

drv_mtch6303.h

C

```
#define DRV_MTCH6303_TOUCH_NUM_INPUTS 0xA
```

Description

MTCH6303 Number of touch input packets

MTCH6303 supports multi-touch and can identify upto 10 different touch input packets.

Remarks

None.

DRV_MTCH6303_BUFFER_EVENT Enumeration

Lists the different conditions that happens during a buffer transfer.

File

drv_mtch6303.h

C

```
typedef enum {
  DRV_MTCH6303_BUFFER_EVENT_COMPLETE,
  DRV_MTCH6303_BUFFER_EVENT_ERROR,
  DRV_MTCH6303_BUFFER_EVENT_ABORT
} DRV_MTCH6303_BUFFER_EVENT;
```

Members

Members	Description
DRV_MTCH6303_BUFFER_EVENT_COMPLETE	Event buffer transfer complete
DRV_MTCH6303_BUFFER_EVENT_ERROR	Event buffer transfer error
DRV_MTCH6303_BUFFER_EVENT_ABORT	Event buffer transfer abort

Description

MTCH6303 Buffer Events

This enumeration identifies the different conditions that can happen during a buffer transaction. Callbacks can be made with the appropriate buffer condition passed as a parameter to execute the desired action.

The values act like flags and multiple flags can be set.

Remarks

None.

DRV_MTCH6303_BUFFER_EVENT_HANDLER Type

Points to a callback after completion of an register read -write or message stream read - write.

File

drv_mtch6303.h

C

```
typedef void (* DRV_MTCH6303_BUFFER_EVENT_HANDLER)(DRV_MTCH6303_BUFFER_EVENT event,
DRV_MTCH6303_BUFFER_HANDLE bufferHandle, uintptr_t context);
```

Description

MTCH6303 Buffer Event Callback

This type identifies the MTCH6303 Buffer Event. It allows the client driver to register a callback using DRV_MTCH6303_BUFFER_EVENT_HANDLER. By using this mechanism, the driver client will be notified at the completion of the corresponding transfer.

Remarks

A transfer can be composed of various transfer segments. Once a transfer is completed the driver will call the client registered transfer callback.

The callback could be called from ISR context and should be kept as short as possible. It is meant for signaling and it should not be blocking.

Parameters

Parameters	Description
DRV_MTCH6303_BUFFER_EVENT	Status of MTCH6303 transfer

bufferHandle	Handle that identifies the particular Buffer Object
context	pointer to the object to be processed.

Function

void (*DRV_MTCH6303_BUFFER_EVENT_HANDLER) (DRV_MTCH6303_BUFFER_EVENT event,
DRV_MTCH6303_BUFFER_HANDLE bufferHandle,
uintptr_t context)

DRV_MTCH6303_BUFFER_HANDLE Type

Handle identifying a read or write buffer passed to the driver.

File

drv_mtch6303.h

C

```
typedef uintptr_t DRV_MTCH6303_BUFFER_HANDLE;
```

Description

MTCH6303 Driver Buffer Handle

A buffer handle value is returned by a call to the DRV_MTCH6303_AddRegisterRead, DRV_MTCH6303_AddRegisterWrite or DRV_MTCH6303_TouchInputRead functions. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from these functions is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None

DRV_MTCH6303_CLIENT_STATUS Enumeration

Defines the client-specific status of the MTCH6303 driver.

File

drv_mtch6303.h

C

```
typedef enum {
  DRV_MTCH6303_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR,
  DRV_MTCH6303_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED,
  DRV_MTCH6303_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY,
  DRV_MTCH6303_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY
} DRV_MTCH6303_CLIENT_STATUS;
```

Members

Members	Description
DRV_MTCH6303_CLIENT_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR	An error has occurred.
DRV_MTCH6303_CLIENT_STATUS_CLOSED = DRV_CLIENT_STATUS_CLOSED	The driver is closed, no operations for this client are ongoing, and/or the given handle is invalid.
DRV_MTCH6303_CLIENT_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY	The driver is currently busy and cannot start additional operations.
DRV_MTCH6303_CLIENT_STATUS_READY = DRV_CLIENT_STATUS_READY	The module is running and ready for additional operations

Description

MTCH6303 Client-Specific Driver Status

This enumeration defines the client-specific status codes of the MTCH6303 driver.

Remarks

Returned by the DRV_MTCH6303_ClientStatus function.

DRV_MTCH6303_ERROR Enumeration

Defines the possible errors that can occur during driver operation.

File

drv_mtch6303.h

C

```
typedef enum {
} DRV_MTCH6303_ERROR;
```

Description

MTCH6303 Driver Errors.

This data type defines the possible errors that can occur when occur during MTCH6303 driver operation. These values are returned by DRV_MTCH6303_ErrorGet function.

Remarks

None

DRV_MTCH6303_TOUCH_BUFFER_EVENT Enumeration

Lists the different conditions that happens during a touch message buffer transfer.

File

drv_mtch6303.h

C

```
typedef enum {
  DRV_MTCH6303_TOUCH_BUFFER_EVENT_COMPLETE,
  DRV_MTCH6303_TOUCH_BUFFER_EVENT_ERROR,
  DRV_MTCH6303_TOUCH_BUFFER_EVENT_ABORT
} DRV_MTCH6303_TOUCH_BUFFER_EVENT;
```

Members

Members	Description
DRV_MTCH6303_TOUCH_BUFFER_EVENT_COMPLETE	Event touch message buffer transfer complete
DRV_MTCH6303_TOUCH_BUFFER_EVENT_ERROR	Event touch message buffer transfer error
DRV_MTCH6303_TOUCH_BUFFER_EVENT_ABORT	Event touch message buffer transfer abort

Description

MTCH6303 Touch Message Buffer Events

This enumeration identifies the different conditions that can happen during a touch message buffer transaction. Callbacks can be made with the appropriate touch message buffer condition passed as a parameter to execute the desired action.

The values act like flags and multiple flags can be set.

Remarks

None.

DRV_MTCH6303_TOUCH_BUFFER_EVENT_HANDLER Type

Points to a callback after completion of an message report read or message command write.

File

drv_mtch6303.h

C

```
typedef void (* DRV_MTCH6303_TOUCH_BUFFER_EVENT_HANDLER)(DRV_MTCH6303_TOUCH_BUFFER_EVENT event,
DRV_MTCH6303_TOUCH_BUFFER_HANDLE bufferHandle, uintptr_t context);
```

Description

MTCH6303 Touch Buffer Event Callback

This type identifies the MTCH6303 Touch Buffer Event. It allows the client driver to register a callback using DRV_MTCH6303_TOUCH_BUFFER_EVENT_HANDLER. By using this mechanism, the driver client will be notified at the completion of the corresponding transfer.

Remarks

A transfer can be composed of various transfer segments. Once a transfer is completed the driver will call the client registered transfer callback.

The callback could be called from ISR context and should be kept as short as possible. It is meant for signaling and it should not be blocking.

Parameters

Parameters	Description
DRV_MTCH6303_TOUCH_BUFFER_EVENT	Status of MTCH6303 touch message transfer
bufferHandle	Handle that identifies the particular Buffer Object
context	pointer to the object to be processed.

Function

void (*DRV_MTCH6303_TOUCH_BUFFER_EVENT_HANDLER) (DRV_MTCH6303_TOUCH_BUFFER_EVENT event,
DRV_MTCH6303_TOUCH_BUFFER_HANDLE bufferHandle,
uintptr_t context)

DRV_MTCH6303_TOUCH_BUFFER_HANDLE Type

Handle identifying a read or write touch message buffer passed to the driver.

File

drv_mtch6303.h

C

```
typedef uintptr_t DRV_MTCH6303_TOUCH_BUFFER_HANDLE;
```

Description

MTCH6303 Driver Touch Message Queue Buffer Handle

A touch message buffer handle value is returned by a call to the DRV_MTCH6303_TOUCH_AddMessageReportRead, DRV_MTCH6303_TOUCH_AddMessageCommandWrite or DRV_MTCH6303_TOUCH_AddTouchInputRead. This handle is associated with the buffer passed into the function and it allows the application to track the completion of the data from (or into) that buffer. The buffer handle value returned from these functions is returned back to the client by the "event handler callback" function registered with the driver.

The buffer handle assigned to a client request expires when the client has been notified of the completion of the buffer transfer (after event handler function that notifies the client returns) or after the buffer has been retired by the driver if no event handler callback was set.

Remarks

None.

DRV_MTCH6303_TOUCH_DATA Structure

Defines MTCH6303 I2C Touch Data

File

drv_mtch6303.h

C

```
typedef struct {
  uint8_t i2cReadAddr;
  DRV_MTCH6303_TOUCH_STATUS status;
  DRV_MTCH6303_TOUCH_INPUT touch[ DRV_MTCH6303_TOUCH_NUM_INPUTS ];
} DRV_MTCH6303_TOUCH_DATA;
```

Members

Members	Description
uint8_t i2cReadAddr;	Dummy I2C Read Address required for bitbang driver
DRV_MTCH6303_TOUCH_STATUS status;	MTCH6303 Touch Status
DRV_MTCH6303_TOUCH_INPUT touch[DRV_MTCH6303_TOUCH_NUM_INPUTS];	MTCH6303 Touch Input array of size DRV_MTCH6303_TOUCH_NUM_INPUTS

Description

MTCH6303 I2C Touch Data

This structure defines MTCH6303 I2C Touch Data. The structure DRV_MTCH6303_TOUCH_DATA is passed to API's DRV_MTCH6303_AddRegisterRead or DRV_MTCH6303_TOUCH_AddTouchInputRead. The API's will update the structure with touch input.

Remarks

It is packed to form structure of size 62 bytes. The structure member i2cReadAddr is only applicable if the I2C driver is of type bitbang. Otherwise the variable required to be commented out.

DRV_MTCH6303_TOUCH_INPUT Structure

Defines MTCH6303 Touch Input Packet

File

drv_mtch6303.h

C

```
typedef struct {
  DRV_MTCH6303_TOUCH_NIBBLE_0 nibble_0;
  uint8_t touchId;
  uint16_t x;
  uint16_t y;
} DRV_MTCH6303_TOUCH_INPUT;
```

Members

Members	Description
DRV_MTCH6303_TOUCH_NIBBLE_0 nibble_0;	MTCH6303 I2C Touch Input Packet Nibble 0
uint8_t touchId;	MTCH6303 I2C Touch Input Packet ID (0 - 16)
uint16_t x;	MTCH6303 I2C Touch Input Packet position x (0 - 0x7FFF)
uint16_t y;	MTCH6303 I2C Touch Input Packet position y (0 - 0x7FFF)

Description

MTCH6303 Touch Input Packet.

This structure defines the MTCH6303 Touch Input Packet.

Remarks

It is part of DRV_MTCH6303_TOUCH_DATA structure. It is packed to form structure of size 6 bytes.

DRV_MTCH6303_TOUCH_MESSAGE Structure

Defines MTCH6303 Touch Message.

File

drv_mtch6303.h

C

```
typedef struct {
  DRV_MTCH6303_TOUCH_MESSAGE_HEADER header;
  uint8_t payload[0x3E];
} DRV_MTCH6303_TOUCH_MESSAGE;
```

Members

Members	Description
DRV_MTCH6303_TOUCH_MESSAGE_HEADER header;	MTCH6303 Touch Message Header
uint8_t payload[0x3E];	MTCH6303 Touch Message payload. First byte of payload is of type DRV_TOUCH_MTCH6303_MSG_ID in case of first fragment of message. Otherwise the first byte acts as a normal payload.

Description

MTCH6303 Touch Message

This structure defines MTCH6303 Touch Message. The variable pointer of type DRV_MTCH6303_TOUCH_MESSAGE is passed to the API's

DRV_MTCH6303_TOUCH_AddMessageReportRead or DRV_MTCH6303_TOUCH_AddMessageCommandWrite.

Remarks

It is packed to form structure of size 63 bytes.

DRV_MTCH6303_TOUCH_MESSAGE_HEADER Structure

Defines Touch Message Header.

File

drv_mtch6303.h

C

```
typedef struct {
  uint32_t msgFragSize : 6;
  uint32_t continued : 1;
  uint32_t moreMessages : 1;
} DRV_MTCH6303_TOUCH_MESSAGE_HEADER;
```

Members

Members	Description
uint32_t msgFragSize : 6;	MTCH6303 Message Fragment Size. If Message Fragment size is 0x3F the Fragment is incomplete and uses up ALL of the parent transport layer packet.
uint32_t continued : 1;	MTCH6303 Message continued from last fragment if set to 1.
uint32_t moreMessages : 1;	MTCH6303 more messages to follow in this block if set to 1.

Description

MTCH6303 Touch Message Header

This structure defines Touch Message Header.

Remarks

It is part of structure DRV_MTCH6303_TOUCH_MESSAGE. It is packed to form structure of size 1 byte.

DRV_MTCH6303_TOUCH_NIBBLE_0 Structure

Defines the I2C Nibble 0 of MTCH6303 Touch input packet.

File

drv_mtch6303.h

C

```
typedef struct {
  uint32_t touchState : 1;
  uint32_t inRange : 1;
  uint32_t reserved : 6;
} DRV_MTCH6303_TOUCH_NIBBLE_0;
```

Members

Members	Description
uint32_t touchState : 1;	Touch packet available
uint32_t inRange : 1;	Touch packet in range
uint32_t reserved : 6;	Reserved bits

Description

MTCH6303 I2C Touch Input Packet Nibble 0

This structure defines the I2C Nibble 0 of MTCH6303 Touch input packet.

Remarks

It is part of DRV_MTCH6303_TOUCH_INPUT structure. It is packed to form structure of size 1 byte.

DRV_MTCH6303_TOUCH_STATUS Structure

Defines the I2C touch status register bits

File

drv_mtch6303.h

C

```
typedef struct {
  uint32_t nTouch : 4;
  uint32_t streamReady : 1;
  uint32_t gestureReady : 1;
  uint32_t gestICData : 1;
  uint32_t reserved : 1;
} DRV_MTCH6303_TOUCH_STATUS;
```

Members

Members	Description
uint32_t nTouch : 4;	Number of available touch packets
uint32_t streamReady : 1;	stream data ready
uint32_t gestureReady : 1;	gesture data ready
uint32_t gestICData : 1;	GestIC data ready
uint32_t reserved : 1;	reserved bit

Description

MTCH6303 I2C touch status

This structure defines the I2C touch status register bits.

Remarks

It is part of DRV_MTCH6303_TOUCH_DATA structure. It is packed to form structure of size 1 byte.

DRV_TOUCH_MTCH6303_MSG_ID Enumeration

List of report or command message identification.

File

drv_mtch6303.h

C

```
typedef enum {
  DRV_TOUCH_MTCH6303_MSG_CMD_QUERY_VERSION
} DRV_TOUCH_MTCH6303_MSG_ID;
```

Members

Members	Description
DRV_TOUCH_MTCH6303_MSG_CMD_QUERY_VERSION	Message sends firmware version query command. Bytes 124:127 = Rev[2].Minor.Major

Description

MTCH6303 Touch message Identification.

This enumeration identifies the different report or command messages supported by MTCH6303. This identifier identifies the type of the message. The identifier is passed in the message DRV_MTCH6303_TOUCH_MESSAGE as first byte of the payload. It is applicable only for first fragment of message. If message involves multiple fragments, the payload of message fragments other than first fragment should start with normal payload byte. The touch message is read or send to MTCH6303 by using DRV_MTCH6303_TOUCH_AddMessageReportRead or DRV_MTCH6303_TOUCH_AddMessageCommandWrite.

Remarks

To be passed as first byte of message payload. Applicable only for first fragment of message.

DRV_TOUCH_MTCH6303_I2C_REGISTER_MAP Enumeration

List of MTCH6303 I2C Accessible Register Identification.

File

drv_mtch6303.h

C

```
typedef enum {
} DRV_TOUCH_MTCH6303_I2C_REGISTER_MAP;
```

Description

MTCH6303 I2C Accessible Register Identification.

This enumeration identifies the different I2C accessible MTCH6303 Registers. The identifier is passed as source to the register read routine or as destination to the register write routine. The MTCH6303 driver routine to read the I2C accessible MTCH6303 registers is DRV_MTCH6303_AddRegisterRead. The MTCH6303 driver routine to write the I2C accessible MTCH6303 registers is DRV_MTCH6303_AddRegisterWrite.

Remarks

To read or write multiple registers, identifier of only first register is sufficient as source or destination respectively.

Files

Files

Name	Description
drv_mtch6303.h	MTCH6303 driver interface declarations for the static single instance driver.

Description

This section lists the source and header files used by the MTCH6303 Touch Driver Library.

drv_mtch6303.h

MTCH6303 driver interface declarations for the static single instance driver.

Enumerations

	Name	Description
	DRV_MTCH6303_BUFFER_EVENT	Lists the different conditions that happens during a buffer transfer.
	DRV_MTCH6303_CLIENT_STATUS	Defines the client-specific status of the MTCH6303 driver.
	DRV_MTCH6303_ERROR	Defines the possible errors that can occur during driver operation.
	DRV_MTCH6303_TOUCH_BUFFER_EVENT	Lists the different conditions that happens during a touch message buffer transfer.
	DRV_TOUCH_MTCH6303_I2C_REGISTER_MAP	List of MTCH6303 I2C Accessible Register Identification.
	DRV_TOUCH_MTCH6303_MSG_ID	List of report or command message identification.

Functions

	Name	Description
	DRV_MTCH6303_AddRegisterRead	Schedules a non-blocking register read request to read I2C accessible MTCH6303 registers.
	DRV_MTCH6303_AddRegisterWrite	Schedule a non-blocking driver register write operation to write I2C accessible MTCH6303 registers.
	DRV_MTCH6303_BufferEventHandlerSet	Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.
	DRV_MTCH6303_Close	Closes an opened-instance of the MTCH6303 driver.
	DRV_MTCH6303_Deinitialize	Deinitializes the instance of the MTCH6303 driver module.
	DRV_MTCH6303_ErrorGet	This function returns the error associated with the last client request.
	DRV_MTCH6303_Initialize	Initializes the MTCH6303 static single instance.
	DRV_MTCH6303_Open	Opens the MTCH6303 driver instance and returns a handle to it.
	DRV_MTCH6303_Status	Gets the current status of the MTCH6303 driver module.

	DRV_MTCH6303_Tasks	Maintains the driver's register read/write state machine and implements its ISR.
	DRV_MTCH6303_TOUCH_AddMessageCommandWrite	Schedule a non-blocking driver command message write operation to write command message to MTCH6303 registers.
	DRV_MTCH6303_TOUCH_AddMessageReportRead	Schedules a non-blocking report message read request to read the report message from MTCH6303 device.
	DRV_MTCH6303_TOUCH_AddTouchInputRead	Schedules a non-blocking read buffer request to read touch input from MTCH6303.
	DRV_MTCH6303_TOUCH_BufferEventHandlerSet	Allows a client to identify a buffer event handling function for the driver to call back when queued message transfers have finished.
	DRV_MTCH6303_TOUCH_Tasks	Maintains the driver's message state machine and implements its ISR.
	DRV_MTCH6303_TouchInputMap	Maps the raw touch input to display resolution.
	DRV_MTCH6303_TouchInputRead	Schedules a non-blocking read buffer request to read touch input from MTCH6303.

Macros

	Name	Description
	DRV_MTCH6303_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_MTCH6303_TOUCH_BUFFER_HANDLE_INVALID	Definition of an invalid buffer handle.
	DRV_MTCH6303_TOUCH_NUM_INPUTS	Definition of number of touch input packets can be identified by MTCH6303.

Structures

	Name	Description
	DRV_MTCH6303_TOUCH_DATA	Defines MTCH6303 I2C Touch Data
	DRV_MTCH6303_TOUCH_INPUT	Defines MTCH6303 Touch Input Packet
	DRV_MTCH6303_TOUCH_MESSAGE	Defines MTCH6303 Touch Message.
	DRV_MTCH6303_TOUCH_MESSAGE_HEADER	Defines Touch Message Header.
	DRV_MTCH6303_TOUCH_NIBBLE_0	Defines the I2C Nibble 0 of MTCH6303 Touch input packet.
	DRV_MTCH6303_TOUCH_STATUS	Defines the I2C touch status register bits

Types

	Name	Description
	DRV_MTCH6303_BUFFER_EVENT_HANDLER	Points to a callback after completion of an register read -write or message stream read - write.
	DRV_MTCH6303_BUFFER_HANDLE	Handle identifying a read or write buffer passed to the driver.
	DRV_MTCH6303_TOUCH_BUFFER_EVENT_HANDLER	Points to a callback after completion of an message report read or message command write.
	DRV_MTCH6303_TOUCH_BUFFER_HANDLE	Handle identifying a read or write touch message buffer passed to the driver.

Description

MTCH6303 Driver Interface Declarations for Static Single Instance Driver

The MTCH6303 device driver provides a simple interface to manage the MTCH6303 module. This file defines the interface Declarations for the MTCH6303 driver.

Remarks

Static single instance driver interface eliminates the need for an object ID or object handle. Static single-open interfaces also eliminate the need for the open handle.

File Name

drv_mtch6303_static.h

Company

Microchip Technology Inc.

mXT336T Touch Driver Library

This topic describes the mXT336T Touch Driver Library.

Introduction

This library provides an interface to manage the mXT336T Touch Driver module on the Microchip family of microcontrollers in different modes of operation.

Description

The MPLAB Harmony mXT336T Touch Driver provides a high-level interface to the mXT336T touch controller device. This driver provides application routines to read the touch input data from the touch screen. The mXT336T device can notify the availability of touch input data through external interrupt. The mXT336T driver allows the application to map a controller pin as an external interrupt pin.

Currently, the mXT336T Touch Driver only supports non-gestural single-fingered touch input.

Using the Library

This topic describes the basic architecture of the mXT336T Touch Driver Library and provides information and examples on its use.

Description

Interface Header Files: `drv_mxt336t.h`, `drv_mxt.h`

The interface to the mXT336T Touch Driver library is defined in the `drv_mxt336t.h` and `drv_mxt.h` header files. Any C language source (`.c`) file that uses the mXT336T Touch Driver library should include this header.

The mXT336T Touch Driver is based on the Object Protocol for the Atmel® maXTouch® mXT336T Touchscreen Controller.

The functioning of the driver is divided into two file sets:

- `drv_mxt.h` has the system touch interface (API's, Initialization and tasks)
- `drv_mxt336t`.h has the device specific interface for getting the device ready for communication and receiving commands.

The device specific interface is based on the Object Protocol previously mentioned.

The aria_quickstart demonstration interfaces with the mXT336T Touch Driver Library. Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the mXT336T Touch Driver Library on the Microchip family microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

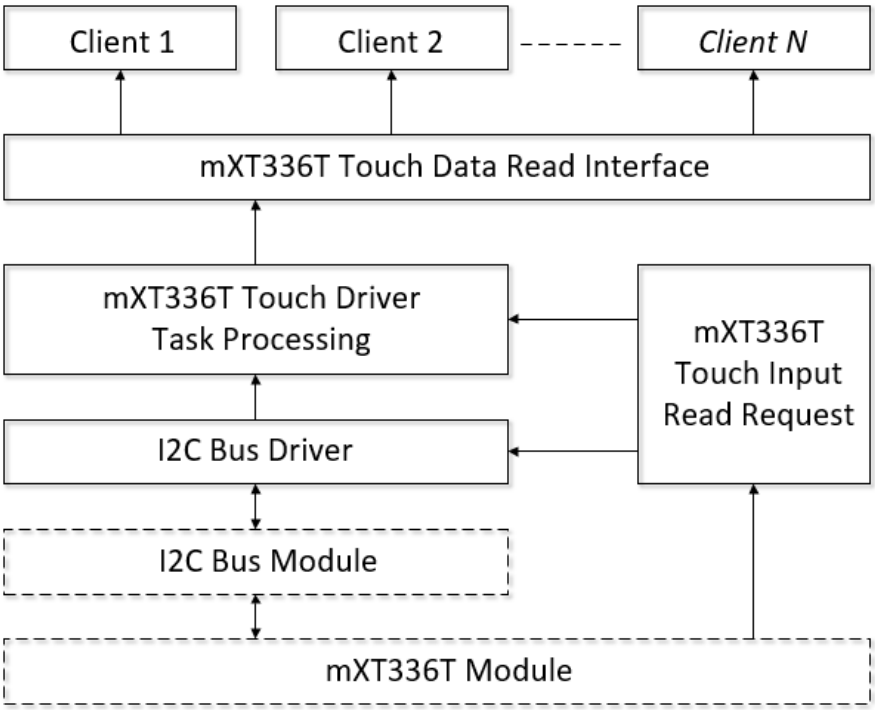
Description

The mXT336T Touch Driver has routines to perform the following operations:

- Sending read request
- Reading the touch input data
- Access to the touch input data

The driver initialization routines allow the application to initialize the driver. The driver must be initialized before it can be used by application. Once the driver is initialized the driver open routine allows retrieving the client handle. Once the touch input is available a touch input read request is sent and input data is retrieved in a buffer. The buffer data is then decoded to get the x and y coordinate of the touch screen in the form of the number of pixels.

mXT336T Driver Abstraction Model



Library Overview

This section contains information about how the Touch Driver operates in a system.

Description

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the mXT336T Touch Driver.

Library Interface Section	Description
Device-specific Functions	Provides mXT336T-specific system module interfaces, device initialization, deinitialization, open, close, task, and status functions.
Generic Functions	Provides generic system module interfaces, device initialization, deinitialization, open, close, task, and status functions.

How the Library Works

This section describes the workings of the Touch Driver Library.

Description

The library provides interfaces to support:

- System functions, which provide system module interfaces, device initialization, deinitialization, open, close, task, and status functions.
- Read Request function, which provides Touch input data read request function
- Read Touch Input function, which provides functions retrieving updated Touch input in the form x and y coordinates.

Initializing the Driver

Before the mXT336T Touch Driver can be opened, it must be configured and initialized. The driver build time configuration is defined by the configuration macros. Refer to the Building the Library section for the location of and more information on the various configuration macros and how these macros should be designed. The driver initialization is configured through the `DRV_TOUCH_INIT` data structure that is passed to the `DRV_MXT336T_Initialize` and the `DRV_MXT_Initialize` functions. The initialization parameters include the interrupt source, interrupt pin remap configuration and touch screen resolution. The following code shows an example of initializing the mXT336T Touch Driver.

Example:

```
/* The following code shows an example of designing the
* DRV_TOUCH_INIT data structure. It also shows how an example
* usage of the DRV_TOUCH_MXT336T_Initialize function.
```

```
*/
DRV_TOUCH_INIT drvTouchInitData;
DRV_MXT_INIT drvMxtInitData;
SYS_MODULE_OBJ objectHandle;

const DRV_MXT336T_INIT drvTouchInitData =
{
 .moduleInit = {0},
 .touchId = DRV_TOUCH_INDEX_0,
 .drvInitialize = NULL,
 .drvOpen = DRV_I2C_Open,
 .orientation = 0,
 .horizontalResolution = 480,
 .verticalResolution = 272,
 .interruptSource = INT_SOURCE_EXTERNAL_1,
 .interruptChannel = PORT_CHANNEL_E,
 .interruptPin = PORTS_BIT_POS_8,
 .resetChannel = PORT_CHANNEL_A,
 .resetPin = PORTS_BIT_POS_2,
};

const DRV_MXT_INIT drvMxtInitData =
{
 .moduleInit = {0},
 .mxtId = DRV_MXT_INDEX_0,
 .drvInitialize = NULL,
 .orientation = 0,
 .horizontalResolution = 480,
 .verticalResolution = 272,
};

/* Driver initialization */
 sysObj.drvMXT336T = DRV_MXT336T_Initialize(DRV_MXT336T_INDEX_0,
 (SYS_MODULE_INIT *)&drvTouchInitData);

 sysObj.drvMxt0 = DRV_MXT_Initialize(DRV_MXT_INDEX_0,
 (SYS_MODULE_INIT *)&drvMxtInitData);
```

Opening the Driver

To use the mXT336T Touch Driver, the application must open the driver. This is done by calling the DRV_MXT_Open function.

If successful, the DRV_MXT_Open function will return a handle to the driver. This handle records the association between the client and the driver instance that was opened. The DRV_MXT_Open function may return DRV_HANDLE_INVALID in the situation where the driver is not ready to be opened. When this occurs, the application can try opening the driver again. Note that the open function may return an invalid handle in other (error) cases as well. The following code shows an example of the driver being opened.

The open function in the driver is called from the system initialization routine by assigning a function pointer from sys_init object.

```
const DRV_TOUCH_INIT sysTouchInit0 =
{
 .drvInitialize = DRV_MXT_Initialize,
 .drvOpen = DRV_MXT_Open,
.
.

};

SYS_TOUCH_HANDLE SYS_TOUCH_Open
(
 const SYS_MODULE_INDEX moduleIndex
)
{
 SYS_TOUCH_CLIENT_OBJ *clientObj;
 SYS_TOUCH_OBJ *dObj;
 .
 .
 .

 /* open touch driver */
 dObj->driverInitData->drvOpen(moduleIndex, DRV_IO_INTENT_READWRITE);
```

```
}
```

Touch Input Read Request

To read the touch input from the mXT336T touch controller device, a read request must be registered. This is done by calling DRV_MXT336T_ReadRequest. If successful, it registers a buffer read request to the I2C command queue. It also adds an input decode command to the mXT336T command queue once the I2C returns with touch input data. It can return error if the driver instance object is invalid or the mXT336T command queue is full. The read request is to be called from the mXT336T ISR. This ISR is triggered once the touch input is available. The following code shows an example of a mXT336T read request registration:

```
SYS_MODULE_OBJ object; // Returned from DRV_TOUCH_MXT336T_Initialize

void ISR(_EXTERNAL_INT_VECTOR, ipl5) _IntHandlerDrvMxt336t(void)
{
DRV__MXT336T_ReadRequest ( object );

// Do other tasks
```

Tasks Routine

This routine processes the mXT336T commands from the command queue. If the state of the command is initialize or done it returns. If the read request registration is successful the state of command is to decode input. The tasks routine decodes the input and updates the global variables storing the touch input data in form of x and y coordinates. The mXT336T Touch Driver task routine is to be called from SYS_Tasks. The following code shows an example:

```
 SYS_MODULE_OBJ drvMXT336T;
 SYS_MODULE_OBJ drvMxt0;; // Returned from DRV_TOUCH_MXT336T_Initialize

void SYS_Tasks( void )
{
 DRV_MXT336T_Tasks(sysObj.drvMXT336T);
 DRV_MXT_Tasks(sysObj.drvMxt0);

// Do other tasks
}
```

Configuring the Library

Macros

	Name	Description
	DRV_MXT336T_CALIBRATION_DELAY	Defines the calibration delay.
	DRV_MXT336T_CALIBRATION_INSET	Defines the calibration inset.
	DRV_MXT336T_CLIENTS_NUMBER	Selects the maximum number of clients.
	DRV_MXT336T_INDEX	MXT336T static index selection.
	DRV_MXT336T_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_MXT336T_INTERRUPT_MODE	Controls operation of the driver in the interrupt or polled mode.
	DRV_MXT336T_SAMPLE_POINTS	Define the sample points.
	DRV_MXT336T_TOUCH_DIAMETER	Defines the touch diameter.

Description

The configuration of the mXT336T Touch Driver is based on the file system_config.h.

This header file contains the configuration selection for the mXT336T Touch Driver. Based on the selections made, the driver may support the selected features. These configuration settings will apply to all instances of the mXT336T Touch Driver.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_MXT336T_CALIBRATION_DELAY Macro

Defines the calibration delay.

File

drv_mxt336t_config_template.h

C

```
#define DRV_MXT336T_CALIBRATION_DELAY 300
```

Description

MXT336T Calibration Delay

This macro enables the delay between calibration touch points.

Remarks

None.

DRV_MXT336T_CALIBRATION_INSET Macro

Defines the calibration inset.

File

drv_mxt336t_config_template.h

C

```
#define DRV_MXT336T_CALIBRATION_INSET 25
```

Description

MXT336T Calibration Inset

This macro defines the calibration inset.

Remarks

None.

DRV_MXT336T_CLIENTS_NUMBER Macro

Selects the maximum number of clients.

File

drv_mxt336t_config_template.h

C

```
#define DRV_MXT336T_CLIENTS_NUMBER 5
```

Description

MXT336T maximum number of clients

This macro selects the maximum number of clients.

This definition selected the maximum number of clients that the MXT336T driver can support at run time.

Remarks

None.

DRV_MXT336T_INDEX Macro

MXT336T static index selection.

File

drv_mxt336t_config_template.h

C

```
#define DRV_MXT336T_INDEX DRV_MXT336T_INDEX_0
```

Description

MXT336T Static Index Selection

This macro specifies the static index selection for the driver object reference.

Remarks

This index is required to make a reference to the driver object.

DRV_MXT336T_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported.

File

drv_mxt336t_config_template.h

C

```
#define DRV_MXT336T_INSTANCES_NUMBER 1
```

Description

MXT336T hardware instance configuration

This macro sets up the maximum number of hardware instances that can be supported.

Remarks

None.

DRV_MXT336T_INTERRUPT_MODE Macro

Controls operation of the driver in the interrupt or polled mode.

File

drv_mxt336t_config_template.h

C

```
#define DRV_MXT336T_INTERRUPT_MODE false
```

Description

MXT336T Interrupt And Polled Mode Operation Control

This macro controls the operation of the driver in the interrupt mode of operation. The possible values of this macro are:

- true - Select if interrupt mode of MXT336T operation is desired
- false - Select if polling mode of MXT336T operation is desired

Not defining this option to true or false will result in a build error.

Remarks

None.

DRV_MXT336T_SAMPLE_POINTS Macro

Define the sample points.

File

drv_mxt336t_config_template.h

C

```
#define DRV_MXT336T_SAMPLE_POINTS 4
```

Description

MXT336T Sample Points

MXT336T sample points

Remarks

None.

DRV_MXT336T_TOUCH_DIAMETER Macro

Defines the touch diameter.

File

drv_mxt336t_config_template.h

C

```
#define DRV_MXT336T_TOUCH_DIAMETER 10
```

Description

MXT336T Touch Diameter

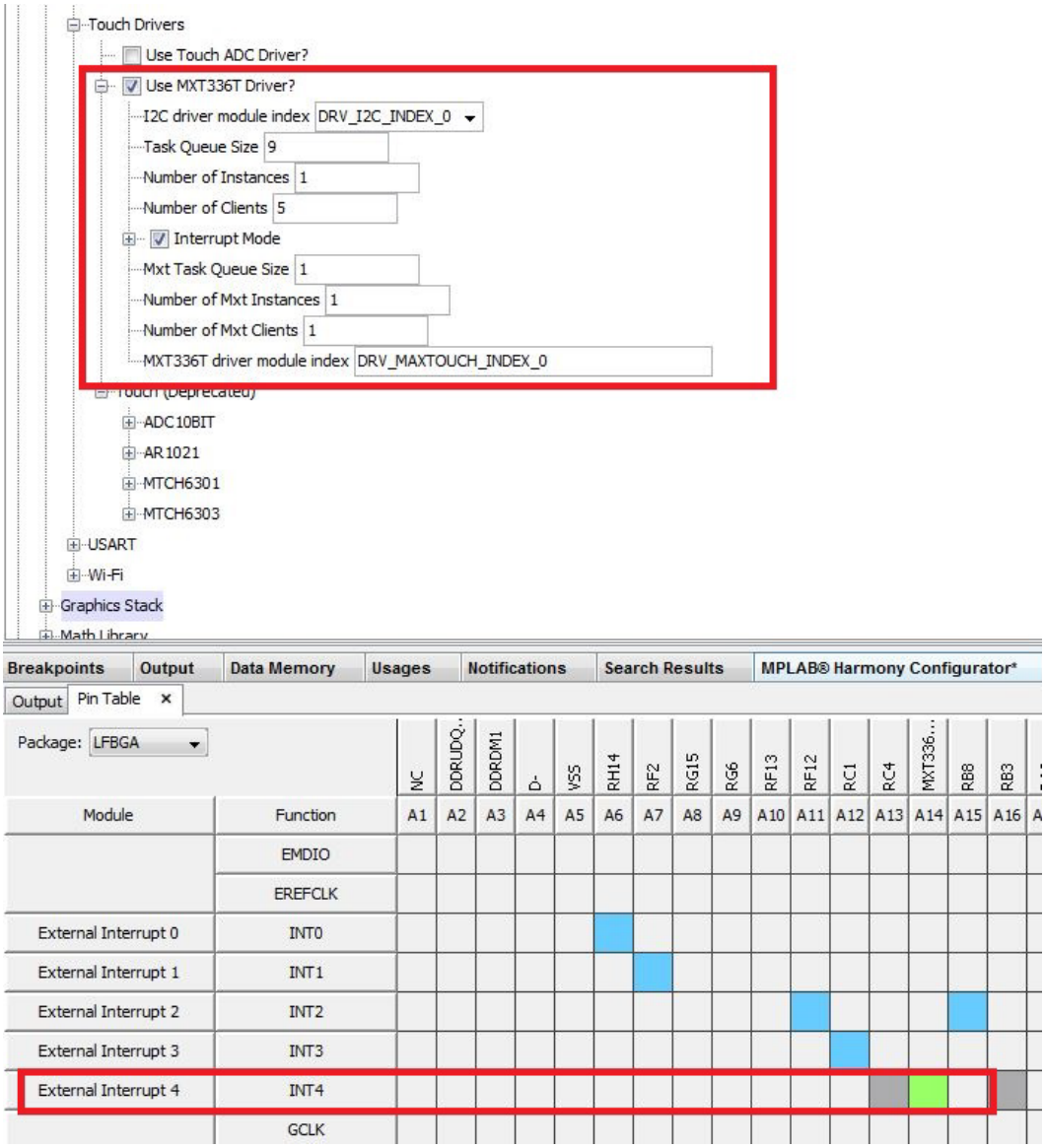
This macro defines the touch diameter

Remarks

None.

Configuring the MHC

The following figure details the settings required to configure the MHC for the mXT336T Touch Driver.



Building the Library

This section lists the files that are available in the mXT336T Touch Driver Library.

Description

This section list the files that are available in the `\src` folder of the mXT336T Touch Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/touch/mxt336t`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_mxt336t.h	Header file that exports the device-specific driver API.
/drv_mxt.h	Header file for generic driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
src/drv_mxt336t.c	Basic mXT336T Touch Driver implementation file.
src/drv_mxt.c	Generic maXTouch touch driver implementation file.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The mXT336T Touch Driver Library depends on the following modules:

- Interrupt System Service Library
- Ports System Service Library
- Touch System Service Library
- I2C Driver Library

Library Interface

a) Device-specific Functions

	Name	Description
	DRV_MXT336T_Close	Closes an opened instance of the MXT336T driver. **Implementation:** Dynamic
	DRV_MXT336T_ReadRequest	Sends a read request to I2C bus driver and adds the read task to queue. **Implementation:** Dynamic
	DRV_MXT336T_Open	Opens the specified MXT336T driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_MXT336T_CloseObject	Closes an opened instance of the MXT336T client object
	DRV_MXT336T_OpenObject	Opens the specified MXT336T object driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_MXT336T_DEVICE_ClientObjectEventHandlerSet	Sets the event handler for a MXT336T client object
	DRV_MXT336T_Deinitialize	Deinitializes the specified instance of the MXT336T driver module. **Implementation:** Dynamic
	DRV_MXT336T_Initialize	Initializes the MXT336T instance for the specified driver index
	DRV_MXT336T_Status	Provides the current status of the MXT336T driver module. **Implementation:** Dynamic

	DRV_MXT336T_Tasks	Maintains the driver's state machine and implements its task queue processing. **Implementation:** Dynamic

b) Generic Functions

	Name	Description
	DRV_MXT_Close	Closes an opened instance of the MXT driver. **Implementation:** Dynamic
	DRV_MXT_MaxtouchEventCallback	
	DRV_MXT_Deinitialize	Deinitializes the specified instance of the MXT driver module. **Implementation:** Dynamic
	DRV_MXT_Open	Opens the specified MXT driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_MXT_TouchDataRead	Notifies the driver that the current touch data has been read
	DRV_MXT_Initialize	Initializes the MXT instance for the specified driver index. **Implementation:** Dynamic
	DRV_MXT_ReadRequest	Sends a read request to I2C bus driver and adds the read task to queue. **Implementation:** Dynamic
	DRV_MXT_TouchGetX	Returns the x coordinate of touch input. **Implementation:** Dynamic
	DRV_MXT_TouchGetY	Returns the y coordinate of touch input. **Implementation:** Dynamic
	DRV_MXT_Status	Provides the current status of the MXT driver module. **Implementation:** Dynamic
	DRV_MXT_Tasks	Maintains the driver's state machine and implements its task queue processing. **Implementation:** Dynamic
	DRV_MXT_TouchStatus	Returns the status of the current touch input.

c) Data Types and Constants

	Name	Description
	_DRV_MXT_CLIENT_OBJECT	MXT Driver client object maintaining client data.
	DRV_MXT_CLIENT_OBJECT	MXT Driver client object maintaining client data.
	DRV_MXT_HANDLE	Touch screen controller MXT driver handle.
	DRV_MXT_INIT	Defines the data required to initialize or reinitialize the MXT driver
	DRV_MXT_MODULE_ID	Number of valid MXT driver indices.
	DRV_MXT_OBJECT	Defines the data structure maintaining MXT driver instance object.
	DRV_MXT_TASK_QUEUE	Defines the MXT Touch Controller driver task data structure.
	DRV_MXT_TASK_STATE	Enumeration defining MXT touch controller driver task state.
	DRV_MXT336T_CLIENT_CALLBACK	Pointer to a MXT336T client callback function data type.
	DRV_MXT336T_HANDLE	Touch screen controller MXT336T driver handle.
	DRV_MXT336T_INIT	Defines the data required to initialize or reinitialize the MXT336T driver
	DRV_MXT336T_OBJECT_CLIENT_EVENT_DATA	This structure maintains the information associated with each msg received or event that occurs
	DRV_MXT336T_OBJECT_TYPE	The enum lists the different objects supported by the maxtouch device.
	DRV_MXT_HANDLE_INVALID	Definition of an invalid handle.
	_DRV_MXT336T_H	This is macro _DRV_MXT336T_H.
	DRV_MXT_I2C_MASTER_READ_ID	MXT input read, I2C address from where master reads touch input data.
	DRV_MXT_I2C_MASTER_WRITE_ID	MXT command register write, I2C address where master sends the commands.
	DRV_MXT_I2C_READ_FRAME_SIZE	I2C Frame size for reading MXT touch input.
	DRV_MXT_INDEX_0	MXT driver index definitions.
	DRV_MXT_INDEX_1	This is macro DRV_MXT_INDEX_1.
	DRV_MXT_INDEX_COUNT	Number of valid Touch controller MXT driver indices.
	DRV_MXT336T_HANDLE_INVALID	Definition of an invalid handle.
	DRV_MXT336T_I2C_FRAME_SIZE	I2C Frame size for reading MXT336T touch input.
	DRV_MXT336T_I2C_MASTER_READ_ID	MXT336T input read, I2C address from where master reads touch input data.

	DRV_MXT336T_I2C_MASTER_WRITE_ID	MXT336T command register write, I2C address where master sends the commands.
	DRV_MXT336T_I2C_READ_ID_FRAME_SIZE	This is macro DRV_MXT336T_I2C_READ_ID_FRAME_SIZE.
	DRV_MXT336T_INDEX_0	MXT336T driver index definitions.
	DRV_MXT336T_INDEX_1	This is macro DRV_MXT336T_INDEX_1.
	DRV_MXT336T_INDEX_COUNT	Number of valid Touch controller MXT336T driver indices.
	t100_event	Types of touch events reported by the Maxtouch Multi touch object
	t100_type	Types of touch types reported by the Maxtouch Multi touch object
	DRV_MXT336T_T100_XRANGE	MXT336T Driver Object Register Adresses for the registers being read in the driver
	DRV_MXT336T_T100_YRANGE	This is macro DRV_MXT336T_T100_YRANGE.

Description

This section describes the functions of the mXT336T Touch Driver Library.

Refer to each section for a detailed description.

a) Device-specific Functions

DRV_MXT336T_Close Function

Closes an opened instance of the MXT336T driver.

Implementation: Dynamic

File

drv_mxt336t.h

C

```
void DRV_MXT336T_Close(DRV_HANDLE handle);
```

Returns

None

Description

This function closes an opened instance of the MXT336T driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_MXT336T_Open before the caller may use the driver again. This function is thread safe in a RTOS application.

Usually, there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_MXT336T_Initialize routine must have been called for the specified MXT336T driver instance.

DRV_MXT336T_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_MXT336T_Open

DRV_MXT336T_Close ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_MXT336T_Close (DRV_HANDLE handle)

DRV_MXT336T_ReadRequest Function

Sends a read request to I2C bus driver and adds the read task to queue.

Implementation: Dynamic

File

drv_mxt336t.h

C

```
void DRV_MXT336T_ReadRequest(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to send a touch input read request to the I2C bus driver and adding the input read decode task to the queue. It is always called from MXT336T interrupt ISR routine.

Remarks

This function is normally not called directly by an application. It is called by the MXT336T ISR routine.

Preconditions

The DRV_MXT336T_Initialize routine must have been called for the specified MXT336T driver instance.

Example

```
SYS_MODULE_OBJ      object;   // Returned from DRV_MXT336T_Initialize

void __ISR(_EXTERNAL_INT_VECTOR, ipl5) _IntHandlerDrvMXT(void)
{
    DRV_MXT336T_ReadRequest ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_MXT336T_Initialize)

Function

void DRV_MXT336T_ReadRequest(SYS_MODULE_OBJ object)

DRV_MXT336T_Open Function

Opens the specified MXT336T driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_mxt336t.h

C

```
DRV_HANDLE DRV_MXT336T_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. An error can occur when the following is true:

- if the number of client objects allocated via DRV_MXT336T_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid

Description

This routine opens the specified MXT336T driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The current version of driver does not support the DRV_IO_INTENT feature. The driver is by default non-blocking. The driver can perform both read and write to the MXT336T device. The driver supports single client only.

Remarks

The handle returned is valid until the DRV_MXT336T_Close routine is called. This routine will NEVER block waiting for hardware. If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

The DRV_MXT336T_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_MXT336T_Open( DRV_MXT336T_INDEX_0,
                                 DRV_IO_INTENT_EXCLUSIVE );

if( DRV_HANDLE_INVALID == handle )
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Index of the driver initialized with DRV_MXT336T_Initialize().
intent	Zero or more of the values from the enumeration DRV_IO_INTENT ORed together to indicate the intended use of the driver. The current version of driver does not support the selective IO intent feature.

Function

DRV_HANDLE DRV_MXT336T_Open (const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT intent)

DRV_MXT336T_CloseObject Function

Closes an opened instance of the MXT336T client object

File

drv_mxt336t.h

C

```
void DRV_MXT336T_CloseObject(DRV_HANDLE handle);
```

Returns

None

Description

This function closes an opened instance of the MXT336T client object, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_MXT336T_OpenObject before the caller may use the driver again. This function is thread safe in a RTOS application.

Usually, there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_MXT336T_Initialize routine must have been called for the specified MXT336T driver instance.

DRV_MXT336T_OpenObject must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_MXT336T_Open

DRV_MXT336T_CloseObject ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_MXT336T_CloseObject (DRV_HANDLE handle)

DRV_MXT336T_OpenObject Function

Opens the specified MXT336T object driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_mxt336t.h

C

```
DRV_HANDLE DRV_MXT336T_OpenObject(const DRV_HANDLE deviceHandle, const uint8_t objType, const uint8_t
objInstance);
```

Returns

If successful, the routine returns a valid object-instance handle (

Description

This routine opens the specified MXT336T object driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver.

Preconditions

The DRV_MXT336T_Initialize function must have been called before calling this function. The driver must have been opened.

Example

```
DRV_HANDLE  handle;

handle = DRV_MXT336T_OpenObject(drvHandle, GEN_PROCESSOR_T5, 1);
```

Parameters

Parameters	Description
deviceHandle	Handle of the MXT336T device
objType	Object type being requested
objInstance	Instance of the object of this type

Function

DRV_HANDLE DRV_MXT336T_OpenObject (const DRV_HANDLE deviceHandle, const uint8_t objType,
const uint8_t objInstance)

DRV_MXT336T_DEVICE_ClientObjectEventHandlerSet Function

Sets the event handler for a MXT336T client object

File

drv_mxt336t.h

C

```
bool DRV_MXT336T_DEVICE_ClientObjectEventHandlerSet(const DRV_HANDLE clientHandle, const
DRV_MXT336T_CLIENT_CALLBACK callback, uintptr_t context);
```

Returns

bool - true if the handler was successfully set

- false if the handler could not be set

Description

This function sets the event handler used to handle report messages from a MXT336T object.

Preconditions

The DRV_MXT336T_OpenObject routine must have been called for the specified MXT336T driver instance.

DRV_MXT336T_OpenObject must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_MXT336T_OpenObject

DRV_MXT336T_DEVICE_ClientObjectEventHandlerSet(handle, objectCallback, NULL);
```

Parameters

Parameters	Description
clientHandle	A valid open-instance handle, returned from the driver's openobject routine
callback	A callback function to handle report messages
context	The context for the call

Function

bool DRV_MXT336T_DEVICE_ClientObjectEventHandlerSet(const DRV_HANDLE clientHandle,

const DRV_MXT336T_CLIENT_CALLBACK callback, uintptr_t context)

DRV_MXT336T_Deinitialize Function

Deinitializes the specified instance of the MXT336T driver module.

Implementation: Dynamic

File

drv_mxt336t.h

C

```
void DRV_MXT336T_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the MXT336T driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again.

This function will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_MXT336T_Status operation. The system has to use DRV_MXT336T_Status to determine when the module is in the ready state.

Preconditions

Function DRV_MXT336T_Initialize must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Parameter: object - Driver object handle, returned from DRV_MXT336T_Initialize

Example

```
SYS_MODULE_OBJ      object;    //Returned from DRV_MXT336T_Initialize
SYS_STATUS          status;

DRV_MXT336T_Deinitialize ( object );

status = DRV_MXT336T_Status( object );
if( SYS_MODULE_UNINITIALIZED == status )
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Function

void DRV_MXT336T_Deinitialize (SYS_MODULE_OBJ object)

DRV_MXT336T_Initialize Function

Initializes the MXT336T instance for the specified driver index

File

drv_mxt336t.h

C

```
SYS_MODULE_OBJ DRV_MXT336T_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the MXT336T driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the 'init' parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the MXT336T module ID. For example, driver instance 0 can be assigned to MXT336T2. If the driver is built statically, then some of the initialization parameters are overridden by configuration macros. Refer to the description of the DRV_MXT336T_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This routine must be called before any other MXT336T routine is called.

This routine should only be called once during system initialization unless DRV_MXT336T_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

None.

Example

```
DRV_MXT336T_INIT        init;
SYS_MODULE_OBJ      objectHandle;

// Populate the MXT336T initialization structure
// Touch Module Id
init.moduleInit                 = {0},
init.touchId                    = DRV_TOUCH_INDEX_0,
init.drvInitialize              = NULL,
init.drvOpen                    = DRV_I2C_Open,
init.interruptSource            = INT_SOURCE_EXTERNAL_1,
init.interruptChannel           = PORT_CHANNEL_D,
init.interruptPin               = PORTS_BIT_POS_1,
init.resetChannel               = PORT_CHANNEL_A,
init.resetPin                   = PORTS_BIT_POS_14,

objectHandle = DRV_MXT336T_Initialize(DRV_TOUCH_INDEX_0,
                                      (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized. Please note this is not the MXT336T ID. The hardware MXT336T ID is set in the initialization structure. This is the index of the driver index to use.
init	Pointer to a data structure containing any data necessary to initialize the driver. If this pointer is NULL, the driver uses the static initialization override macros for each member of the initialization data structure.

Function

SYS_MODULE_OBJ DRV_MXT336T_Initialize(const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init)

DRV_MXT336T_Status Function

Provides the current status of the MXT336T driver module.

Implementation: Dynamic

File

drv_mxt336t.h

C

```
SYS_STATUS DRV_MXT336T_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system-level operation and cannot start another

Description

This function provides the current status of the MXT336T driver module.

Remarks

Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_MODULE_UNINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR.

This function can be used to determine when any of the driver's module level operations has completed.

If the status operation returns SYS_STATUS_BUSY, the previous operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This function will NEVER block waiting for hardware.

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The DRV_MXT336T_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;  // Returned from DRV_MXT336T_Initialize
SYS_STATUS          status;

status = DRV_MXT336T_Status( object );
if( SYS_STATUS_READY != status )
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_MXT336T_Initialize

Function

SYS_STATUS DRV_MXT336T_Status (SYS_MODULE_OBJ object)

DRV_MXT336T_Tasks Function

Maintains the driver's state machine and implements its task queue processing.

Implementation: Dynamic

File

drv_mxt336t.h

C

```
void DRV_MXT336T_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal state machine and implement its command queue processing. It is always called from SYS_Tasks() function. This routine decodes the touch input data available in drvI2CReadFrameData.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks)

Preconditions

The DRV_MXT336T_Initialize routine must have been called for the specified MXT336T driver instance.

Example

```
SYS_MODULE_OBJ      object;   // Returned from DRV_MXT336T_Initialize

void SYS_Tasks( void )
{
    DRV_MXT336T_Tasks ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_MXT336T_Initialize)

Function

void DRV_MXT336T_Tasks (SYS_MODULE_OBJ object);

b) Generic Functions

DRV_MXT_Close Function

Closes an opened instance of the MXT driver.

Implementation: Dynamic

File

drv_mxt.h

C

```
void DRV_MXT_Close(DRV_HANDLE handle);
```

Returns

None

Description

This function closes an opened instance of the MXT driver, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling DRV_MXT_Open before the caller may use the driver again. This function is thread safe in a RTOS application.

Usually, there is no need for the driver client to verify that the Close operation has completed.

Preconditions

The DRV_MXT_Initialize routine must have been called for the specified MXT driver instance.

DRV_MXT_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_MXT_Open
```

```
DRV_MXT_Close ( handle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_MXT_Close (DRV_HANDLE handle)

DRV_MXT_MaxtouchEventCallback Function

File

drv_mxt.h

C

```
void DRV_MXT_MaxtouchEventCallback(DRV_HANDLE clientObject, DRV_MXT336T_OBJECT_CLIENT_EVENT_DATA *
updateObject, uintptr_t context);
```

Remarks

See prototype in app.h.

Function

void DRV_MXT_MaxtouchEventCallback (DRV_HANDLE clientObject,

DRV_MAXTOUCH_OBJECT_CLIENT_EVENT_DATA *updateObject, uintptr_t context);

DRV_MXT_Deinitialize Function

Deinitializes the specified instance of the MXT driver module.

Implementation: Dynamic

File

drv_mxt.h

C

```
void DRV_MXT_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the MXT driver module, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again.

This function will NEVER block waiting for hardware. If the operation requires time to allow the hardware to complete, this will be reported by the DRV_MXT_Status operation. The system has to use DRV_MXT_Status to determine when the module is in the ready state.

Preconditions

Function DRV_MXT_Initialize must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Parameter: object - Driver object handle, returned from DRV_MXT_Initialize

Example

```
SYS_MODULE_OBJ      object;    //Returned from DRV_MXT_Initialize
SYS_STATUS          status;

DRV_MXT_Deinitialize ( object );

status = DRV_MXT_Status( object );
if( SYS_MODULE_UNINITIALIZED == status )
{
    // Check again later if you need to know
```

```
    // when the driver is deinitialized.
}
```

Function

void DRV_MXT_Deinitialize (SYS_MODULE_OBJ object)

DRV_MXT_Open Function

Opens the specified MXT driver instance and returns a handle to it.

Implementation: Dynamic

File

drv_mxt.h

C

```
DRV_HANDLE DRV_MXT_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. An error can occur when the following is true:

- if the number of client objects allocated via DRV_MXT_CLIENTS_NUMBER is insufficient
- if the client is trying to open the driver but driver has been opened exclusively by another client
- if the driver hardware instance being opened is not initialized or is invalid

Description

This routine opens the specified MXT driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The current version of driver does not support the DRV_IO_INTENT feature. The driver is by default non-blocking. The driver can perform both read and write to the MXT device. The driver supports single client only.

Remarks

The handle returned is valid until the DRV_MXT_Close routine is called. This routine will NEVER block waiting for hardware. If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application. It should not be called in an ISR.

Preconditions

The DRV_MXT_Initialize function must have been called before calling this function.

Example

```
DRV_HANDLE  handle;

handle = DRV_MXT_Open( DRV_MXT_INDEX_0,
                                    DRV_IO_INTENT_EXCLUSIVE );

if( DRV_HANDLE_INVALID == handle )
{
    // Unable to open the driver
}
```

Parameters

Parameters	Description
drvIndex	Index of the driver initialized with DRV_MXT_Initialize().
intent	Zero or more of the values from the enumeration DRV_IO_INTENT ORed together to indicate the intended use of the driver. The current version of driver does not support the selective IO intent feature.

Function

DRV_HANDLE DRV_MXT_Open (const SYS_MODULE_INDEX drvIndex,

const DRV_IO_INTENT intent)

DRV_MXT_TouchDataRead Function

Notifies the driver that the current touch data has been read

File

drv_mxt.h

C

```
void DRV_MXT_TouchDataRead(const SYS_MODULE_INDEX index);
```

Returns

None.

Description

Notifies the driver that the current touch data has been read

Function

void DRV_MXT_TouchDataRead(const SYS_MODULE_INDEX index)

DRV_MXT_Initialize Function

Initializes the MXT instance for the specified driver index.

Implementation: Dynamic

File

drv_mxt.h

C

```
SYS_MODULE_OBJ DRV_MXT_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the MXT driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the 'init' parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the MXT module ID. For example, driver instance 0 can be assigned to MXT2. If the driver is built statically, then some of the initialization parameters are overridden by configuration macros. Refer to the description of the DRV_MXT_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This routine must be called before any other MXT routine is called.

This routine should only be called once during system initialization unless DRV_MXT_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

None.

Example

```
DRV_MXT_INIT        init;
SYS_MODULE_OBJ      objectHandle;

// Populate the MXT initialization structure
// Touch Module Id
init.touchId                    = DRV_TOUCH_INDEX_0;

// I2C Bus driver open
init.drvOpen                    = DRV_I2C_Open;

// Interrupt Source for Touch
init.interruptSource            = INT_SOURCE_EXTERNAL_1;

// Interrupt Pin function mapping
```

```
init.interruptPort.inputFunction  = INPUT_FUNC_INT1;

// Pin to be mapped as interrupt pin
init.interruptPort.inputPin       = INPUT_PIN_RPE8;

// Analog pin number
init.interruptPort.analogPin      = PORTS_ANALOG_PIN_25;

// Pin Mode of analog pin
init.interruptPort.pinMode        = PORTS_PIN_MODE_DIGITAL;

// Interrupt pin port
init.interruptPort.channel        = PORT_CHANNEL_E;

// Interrupt pin port maskl
init.interruptPort.dataMask       = 0x8;

// Touch screen orientation
init.orientation                  = DISP_ORIENTATION;

// Touch screen horizontal resolution
init.horizontalResolution         = DISP_HOR_RESOLUTION;

// Touch screen vertical resolution
init.verticalResolution           = DISP_VER_RESOLUTION;


objectHandle = DRV_MXT_Initialize(DRV_TOUCH_INDEX_0,
                                  (SYS_MODULE_INIT*)init);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized. Please note this is not the MXT ID. The hardware MXT ID is set in the initialization structure. This is the index of the driver index to use.
init	Pointer to a data structure containing any data necessary to initialize the driver. If this pointer is NULL, the driver uses the static initialization override macros for each member of the initialization data structure.

Function

SYS_MODULE_OBJ DRV_MXT_Initialize(const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init)

DRV_MXT_ReadRequest Function

Sends a read request to I2C bus driver and adds the read task to queue.

Implementation: Dynamic

File

drv_mxt.h

C

```
void DRV_MXT_ReadRequest(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to send a touch input read request to the I2C bus driver and adding the input read decode task to the queue. It is always called from MXT interrupt ISR routine.

Remarks

This function is normally not called directly by an application. It is called by the MXT ISR routine.

Preconditions

The DRV_MXT_Initialize routine must have been called for the specified MXT driver instance.

Example

```
SYS_MODULE_OBJ      object;   // Returned from DRV_MXT_Initialize

void __ISR(_EXTERNAL_INT_VECTOR, ipl5) _IntHandlerDrvMXT(void)
{
    DRV_MXT_ReadRequest ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_MXT_Initialize)

Function

void DRV_MXT_ReadRequest(SYS_MODULE_OBJ object)

DRV_MXT_TouchGetX Function

Returns the x coordinate of touch input.

Implementation: Dynamic

File

drv_mxt.h

C

```
short DRV_MXT_TouchGetX(uint8_t touchNumber);
```

Returns

It returns the x coordinate of the touch input in terms of number of pixels.

Description

It returns the x coordinate in form of number of pixes for a touch input denoted by touchNumber.

Parameters

Parameters	Description
touchNumber	index to the touch input.

Function

short DRV_MXT_TouchGetX(uint8 touchNumber)

DRV_MXT_TouchGetY Function

Returns the y coordinate of touch input.

Implementation: Dynamic

File

drv_mxt.h

C

```
short DRV_MXT_TouchGetY(uint8_t touchNumber);
```

Returns

It returns the y coordinate of the touch input in terms of number of pixels.

Description

It returns the y coordinate in form of number of pixes for a touch input denoted by touchNumber.

Parameters

Parameters	Description
touchNumber	index to the touch input.

Function

short DRV_MXT_TouchGetY(uint8 touchNumber)

DRV_MXT_Status Function

Provides the current status of the MXT driver module.

Implementation: Dynamic

File

drv_mxt.h

C

```
SYS_STATUS DRV_MXT_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system-level operation and cannot start another

Description

This function provides the current status of the MXT driver module.

Remarks

Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

SYS_MODULE_UNINITIALIZED - Indicates that the driver has been deinitialized

This value is less than SYS_STATUS_ERROR.

This function can be used to determine when any of the driver's module level operations has completed.

If the status operation returns SYS_STATUS_BUSY, the previous operation has not yet completed. Once the status operation returns SYS_STATUS_READY, any previous operations have completed.

The value of SYS_STATUS_ERROR is negative (-1). Any value less than that is also an error state.

This function will NEVER block waiting for hardware.

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The DRV_MXT_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;  // Returned from DRV_MXT_Initialize
SYS_STATUS          status;

status = DRV_MXT_Status( object );
if( SYS_STATUS_READY != status )
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from DRV_MXT_Initialize

Function

SYS_STATUS DRV_MXT_Status (SYS_MODULE_OBJ object)

DRV_MXT_Tasks Function

Maintains the driver's state machine and implements its task queue processing.

Implementation: Dynamic

File

drv_mxt.h

C

```
void DRV_MXT_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal state machine and implement its command queue processing. It is always called from SYS_Tasks() function. This routine decodes the touch input data available in drvI2CReadFrameData.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks)

Preconditions

The DRV_MXT_Initialize routine must have been called for the specified MXT driver instance.

Example

```
SYS_MODULE_OBJ      object;   // Returned from DRV_MXT_Initialize

void SYS_Tasks( void )
{
    DRV_MXT_Tasks ( object );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_MXT_Initialize)

Function

void DRV_MXT_Tasks (SYS_MODULE_OBJ object);

DRV_MXT_TouchStatus Function

Returns the status of the current touch input.

File

drv_mxt.h

C

```
DRV_TOUCH_POSITION_STATUS DRV_MXT_TouchStatus(const SYS_MODULE_INDEX index);
```

Returns

It returns the status of the current touch input.

Description

It returns the status of the current touch input.

Function

DRV_TOUCH_POSITION_SINGLE DRV_MXT_TouchStatus(const SYS_MODULE_INDEX index)

c) Data Types and Constants

DRV_MXT_CLIENT_OBJECT Structure

MXT Driver client object maintaining client data.

File

drv_mxt.h

C

```
typedef struct _DRV_MXT_CLIENT_OBJECT {
  DRV_MXT_OBJECT* driverObject;
  DRV_IO_INTENT intent;
} DRV_MXT_CLIENT_OBJECT;
```

Members

Members	Description
DRV_MXT_OBJECT* driverObject;	Driver Object associated with the client
DRV_IO_INTENT intent;	The intent with which the client was opened

Description

Structure DRV_MXT_CLIENT_OBJECT

This defines the object required for the maintenance of the software clients instance. This object exists once per client instance.

Remarks

None.

DRV_MXT_HANDLE Type

Touch screen controller MXT driver handle.

File

drv_mxt.h

C

```
typedef uintptr_t DRV_MXT_HANDLE;
```

Description

MXT Driver Handle

Touch controller MXT driver handle is a handle for the driver client object. Each driver with succesful open call will return a new handle to the client object.

Remarks

None.

DRV_MXT_INIT Structure

Defines the data required to initialize or reinitialize the MXT driver

File

drv_mxt.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  int mxtId;
  SYS_MODULE_OBJ (* drvInitialize)(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
  SYS_MODULE_INDEX maxtouchID;
  uint16_t orientation;
  uint16_t horizontalResolution;
  uint16_t verticalResolution;
} DRV_MXT_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
int mxtId;	ID
SYS_MODULE_OBJ (* drvInitialize)(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);	initialize function for module (normally called statically
SYS_MODULE_INDEX maxtouchID;	index for the maxtouch driver instance used by this driver
uint16_t orientation;	Orientation of the display (given in degrees of 0,90,180,270)
uint16_t horizontalResolution;	Horizontal Resolution of the displayed orientation in Pixels

Description

Structure DRV_MXT_INIT

This data type defines the data required to initialize or reinitialize the MXT driver. If the driver is built statically, the members of this data structure are statically over-ridden by static override definitions in the system_config.h file.

Remarks

None.

DRV_MXT_MODULE_ID Enumeration

Number of valid MXT driver indices.

File

drv_mxt.h

C

```
typedef enum {
  MXT_ID_1 = 0,
  MXT_NUMBER_OF_MODULES
} DRV_MXT_MODULE_ID;
```

Description

Enumeration: DRV_MXT_MODULE_ID

This constant identifies the number of valid MXT driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific header files defined as part of the peripheral libraries.

DRV_MXT_OBJECT Structure

Defines the data structure maintaining MXT driver instance object.

File

drv_mxt.h

C

```
typedef struct {
  SYS_STATUS status;
  int mxtId;
  SYS_MODULE_INDEX drvIndex;
  bool inUse;
  bool isExclusive;
  uint8_t numClients;
  uint16_t orientation;
  uint16_t horizontalResolution;
  uint16_t verticalResolution;
  int32_t readRequest;
  SYS_MODULE_INDEX maxtouchID;
  DRV_HANDLE hMaxtouch;
  DRV_HANDLE hMaxtouchGestureClient;
  DRV_TOUCH_POSITION_STATUS touchStatus;
```

```
  bool maxtouchDataAvailable;
  uint8_t maxtouchData[32];
  uint16_t xRange;
  uint16_t yRange;
} DRV_MXT_OBJECT;
```

Members

Members	Description
SYS_STATUS status;	The status of the driver
int mxtId;	The peripheral Id associated with the object
SYS_MODULE_INDEX drvIndex;	Save the index of the driver. Important to know this as we are using reference based accessing
bool inUse;	Flag to indicate instance in use
bool isExclusive;	Flag to indicate module used in exclusive access mode
uint8_t numClients;	Number of clients possible with the hardware instance
uint16_t orientation;	Orientation of the display (given in degrees of 0,90,180,270)
uint16_t horizontalResolution;	Horizontal Resolution of the displayed orientation in Pixels
uint16_t verticalResolution;	Vertical Resolution of the displayed orientaion in Pixels
int32_t readRequest;	Touch Input read request counter
SYS_MODULE_INDEX maxtouchID;	index of the maxtouch driver being used
DRV_HANDLE hMaxtouch;	handle for the maxtouch driver being used
DRV_HANDLE hMaxtouchGestureClient;	handle for the maxtouch driver object we are listening to
DRV_TOUCH_POSITION_STATUS touchStatus;	Touch status
bool maxtouchDataAvailable;	flag to indicate new maxtouch data is available
uint8_t maxtouchData[32];	data from the maxtouch device

Description

Structure DRV_MXT_OBJECT

This data structure maintains the MXT driver instance object. The object exists once per hardware instance.

Remarks

None.

DRV_MXT_TASK_QUEUE Structure

Defines the MXT Touch Controller driver task data structure.

File

drv_mxt.h

C

```
typedef struct {
  bool inUse;
  DRV_MXT_TASK_STATE taskState;
  DRV_I2C_BUFFER_HANDLE drvI2CReadBufferHandle;
  uint8_t drvI2CReadFrameData[DRV_MXT_I2C_READ_FRAME_SIZE];
} DRV_MXT_TASK_QUEUE;
```

Members

Members	Description
bool inUse;	Flag denoting the allocation of task
DRV_MXT_TASK_STATE taskState;	Enum maintaining the task state
DRV_I2C_BUFFER_HANDLE drvI2CReadBufferHandle;	I2C Buffer handle

uint8_t drvI2CReadFrameData[DRV_MXT_I2C_READ_FRAME_SIZE];	Response to Read Touch Input Command • Response = { MXT Read Address, • Input Data Size, • Touch Id, Pen status, • Touch X coordinate (0 to 6), • Touch X coordinate (7 to 11), • Touch Y coordinate (0 to 6), • Touch Y coordinate (7 to 11) }

Description

Structure DRV_MXT_TASK_QUEUE

This data type defines the data structure maintaing task context in the task queue. The inUse flag denotes the task context allocation for a task. The enum variable taskState maintains the current task state. The I2C buffer handle drvI2CReadBufferHandle maintains the I2C driver buffer handle returned by the I2C driver read request. The byte array variable drvI2CReadFrameData maintains the I2C frame data sent by MXT after a successful read request.

Remarks

None.

DRV_MXT_TASK_STATE Enumeration

Enumeration defining MXT touch controller driver task state.

File

drv_mxt.h

C

```
typedef enum {
  DRV_MXT_TASK_STATE_INIT = 0,
  DRV_MXT_TASK_STATE_READ_INPUT,
  DRV_MXT_TASK_STATE_DECODE_INPUT,
  DRV_MXT_TASK_STATE_DONE
} DRV_MXT_TASK_STATE;
```

Members

Members	Description
DRV_MXT_TASK_STATE_INIT = 0	Task initialize state
DRV_MXT_TASK_STATE_READ_INPUT	Task read touch input request state
DRV_MXT_TASK_STATE_DECODE_INPUT	Task touch input decode state
DRV_MXT_TASK_STATE_DONE	Task complete state

Description

Enumeration DRV_MXT_TASK_STATE

This enumeration defines the MXT touch controller driver task state. The task state helps to synchronize the operations of initialization the the task, adding the read input task to the task queue once the touch controller notifies the available touch input and a decoding the touch input received.

Remarks

None.

DRV_MXT336T_CLIENT_CALLBACK Type

Pointer to a MXT336T client callback function data type.

File

drv_mxt336t.h

C

```
typedef void (* DRV_MXT336T_CLIENT_CALLBACK)(DRV_HANDLE clientObject, DRV_MXT336T_OBJECT_CLIENT_EVENT_DATA
*updateObject, uintptr_t context);
```

Description

MXT336T Driver Callback Function Pointer

This data type defines a pointer to a MXT336T client callback function.

DRV_MXT336T_HANDLE Type

Touch screen controller MXT336T driver handle.

File

drv_mxt336t.h

C

```
typedef uintptr_t DRV_MXT336T_HANDLE;
```

Description

MXT336T Driver Handle

Touch controller MXT336T driver handle is a handle for the driver client object. Each driver with successful open call will return a new handle to the client object.

Remarks

None.

DRV_MXT336T_INIT Type

Defines the data required to initialize or reinitialize the MXT336T driver

File

drv_mxt336t.h

C

```
typedef struct DRV_MXT336T_INIT@2 DRV_MXT336T_INIT;
```

Description

Structure DRV_MXT336T_INIT

This data type defines the data required to initialize or reinitialize the MXT336T driver. If the driver is built statically, the members of this data structure are statically over-ridden by static override definitions in the system_config.h file.

Remarks

None.

DRV_MXT336T_OBJECT_CLIENT_EVENT_DATA Structure

This structure maintains the information associated with each msg received or event that occurs

File

drv_mxt336t.h

C

```
typedef struct {
  uint8_t reportID;
  uint8_t dataSize;
  uint8_t * pData;
  uint16_t xRange;
  uint16_t yRange;
} DRV_MXT336T_OBJECT_CLIENT_EVENT_DATA;
```

Members

Members	Description
uint8_t reportID;	report ID within the object
uint8_t dataSize;	max size of data
uint8_t * pData;	data associated with the report

Description

Structure DRV_MXT336T_OBJECT_CLIENT_EVENT_DATA

This structure maintains the information associated with each msg received or event that occurs. Each msg gets a reportID that identifies the object reporting the change in status. For touch messages the xRange and yRange for the touch device gets reported and the data pointer contains the status msg information which has the touch type, touch event and touch coordinates.

Remarks

None.

DRV_MXT336T_OBJECT_TYPE Enumeration

The enum lists the different objects supported by the maxtouch device.

File

drv_mxt336t.h

C

```
typedef enum {
  DRV_MXT336T_OBJECT_RESERVED_T0 = 0,
  DRV_MXT336T_OBJECT_RESERVED_T1 = 1,
  DRV_MXT336T_OBJECT_DEBUG_DELTAS_T2 = 2,
  DRV_MXT336T_OBJECT_DEBUG_REFERENCES_T3 = 3,
  DRV_MXT336T_OBJECT_DEBUG_SIGNALS_T4 = 4,
  DRV_MXT336T_OBJECT_GEN_MESSAGEPROCESSOR_T5 = 5,
  DRV_MXT336T_OBJECT_GEN_COMMANDPROCESSOR_T6 = 6,
  DRV_MXT336T_OBJECT_GEN_POWERCONFIG_T7 = 7,
  DRV_MXT336T_OBJECT_GEN_ACQUISITIONCONFIG_T8 = 8,
  DRV_MXT336T_OBJECT_TOUCH_MULTITOUCHSCREEN_T9 = 9,
  DRV_MXT336T_OBJECT_TOUCH_SINGLETOUCHSCREEN_T10 = 10,
  DRV_MXT336T_OBJECT_TOUCH_XSLIDER_T11 = 11,
  DRV_MXT336T_OBJECT_TOUCH_YSLIDER_T12 = 12,
  DRV_MXT336T_OBJECT_TOUCH_XWHEEL_T13 = 13,
  DRV_MXT336T_OBJECT_TOUCH_YWHEEL_T14 = 14,
  DRV_MXT336T_OBJECT_TOUCH_KEYARRAY_T15 = 15,
  DRV_MXT336T_OBJECT_PROCG_SIGNALFILTER_T16 = 16,
  DRV_MXT336T_OBJECT_PROCI_LINEARIZATIONTABLE_T17 = 17,
  DRV_MXT336T_OBJECT_SPT_COMMSCONFIG_T18 = 18,
  DRV_MXT336T_OBJECT_SPT_GPIOPWM_T19 = 19,
  DRV_MXT336T_OBJECT_PROCI_GRIPFACESUPPRESSION_T20 = 20,
  DRV_MXT336T_OBJECT_RESERVED_T21 = 21,
  DRV_MXT336T_OBJECT_PROCG_NOISESUPPRESSION_T22 = 22,
  DRV_MXT336T_OBJECT_TOUCH_PROXIMITY_T23 = 23,
  DRV_MXT336T_OBJECT_PROCI_ONETOUCHGESTUREPROCESSOR_T24 = 24,
  DRV_MXT336T_OBJECT_SPT_SELFTEST_T25 = 25,
  DRV_MXT336T_OBJECT_DEBUG_CTERANGE_T26 = 26,
  DRV_MXT336T_OBJECT_PROCI_TWOTOUCHGESTUREPROCESSOR_T27 = 27,
  DRV_MXT336T_OBJECT_SPT_CTECONFIG_T28 = 28,
  DRV_MXT336T_OBJECT_SPT_GPI_T29 = 29,
  DRV_MXT336T_OBJECT_SPT_GATE_T30 = 30,
  DRV_MXT336T_OBJECT_TOUCH_KEYSET_T31 = 31,
  DRV_MXT336T_OBJECT_TOUCH_XSLIDERSET_T32 = 32,
  DRV_MXT336T_OBJECT_RESERVED_T33 = 33,
  DRV_MXT336T_OBJECT_GEN_MESSAGEBLOCK_T34 = 34,
  DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T35 = 35,
  DRV_MXT336T_OBJECT_RESERVED_T36 = 36,
  DRV_MXT336T_OBJECT_DEBUG_DIAGNOSTIC_T37 = 37,
  DRV_MXT336T_OBJECT_SPT_USERDATA_T38 = 38,
  DRV_MXT336T_OBJECT_SPARE_T39 = 39,
  DRV_MXT336T_OBJECT_PROCI_GRIPSUPPRESSION_T40 = 40,
  DRV_MXT336T_OBJECT_PROCI_PALMSUPPRESSION_T41 = 41,
  DRV_MXT336T_OBJECT_PROCI_TOUCHSUPPRESSION_T42 = 42,
  DRV_MXT336T_OBJECT_SPT_DIGITIZER_T43 = 43,
  DRV_MXT336T_OBJECT_SPT_MESSAGECOUNT_T44 = 44,
  DRV_MXT336T_OBJECT_PROCI_VIRTUALKEY_T45 = 45,
  DRV_MXT336T_OBJECT_SPT_CTECONFIG_T46 = 46,
  DRV_MXT336T_OBJECT_PROCI_STYLUS_T47 = 47,
  DRV_MXT336T_OBJECT_PROCG_NOISESUPPRESSION_T48 = 48,
  DRV_MXT336T_OBJECT_GEN_DUALPULSE_T49 = 49,
  DRV_MXT336T_OBJECT_SPARE_T50 = 50,
  DRV_MXT336T_OBJECT_SPT_SONY_CUSTOM_T51 = 51,
  DRV_MXT336T_OBJECT_TOUCH_PROXKEY_T52 = 52,
  DRV_MXT336T_OBJECT_GEN_DATASOURCE_T53 = 53,
  DRV_MXT336T_OBJECT_PROCG_NOISESUPPRESSION_T54 = 54,
  DRV_MXT336T_OBJECT_PROCI_ADAPTIVETHRESHOLD_T55 = 55,
```

```
    DRV_MXT336T_OBJECT_PROCI_SHIELDLESS_T56 = 56,
    DRV_MXT336T_OBJECT_PROCI_EXTRATOUCHSCREENDATA_T57 = 57,
    DRV_MXT336T_OBJECT_SPT_EXTRANOISESUPCTRLS_T58 = 58,
    DRV_MXT336T_OBJECT_SPT_FASTDRIFT_T59 = 59,
    DRV_MXT336T_OBJECT_SPT_TIMER_T61 = 61,
    DRV_MXT336T_OBJECT_PROCG_NOISESUPPRESSION_T62 = 62,
    DRV_MXT336T_OBJECT_PROCI_ACTIVESTYLUS_T63 = 63,
    DRV_MXT336T_OBJECT_SPT_REFERENCERELOAD_T64 = 64,
    DRV_MXT336T_OBJECT_PROCI_LENSBENDING_T65 = 65,
    DRV_MXT336T_OBJECT_SPT_GOLDENREFERENCES_T66 = 66,
    DRV_MXT336T_OBJECT_PROCI_CUSTOMGESTUREPROCESSOR_T67 = 67,
    DRV_MXT336T_OBJECT_SERIAL_DATA_COMMAND_T68 = 68,
    DRV_MXT336T_OBJECT_PROCI_PALMGESTUREPROCESSOR_T69 = 69,
    DRV_MXT336T_OBJECT_SPT_DYNAMICCONFIGURATIONCONTROLLER_T70 = 70,
    DRV_MXT336T_OBJECT_SPT_DYNAMICCONFIGURATIONCONTAINER_T71 = 71,
    DRV_MXT336T_OBJECT_PROCG_NOISESUPPRESSION_T72 = 72,
    DRV_MXT336T_OBJECT_PROCI_ZONEINDICATION_T73 = 73,
    DRV_MXT336T_OBJECT_PROCG_SIMPLEGESTUREPROCESSOR_T74 = 74,
    DRV_MXT336T_OBJECT_MOTION_SENSING_OBJECT_T75 = 75,
    DRV_MXT336T_OBJECT_PROCI_MOTION_GESTURES_T76 = 76,
    DRV_MXT336T_OBJECT_SPT_CTESCANCONFIG_T77 = 77,
    DRV_MXT336T_OBJECT_PROCI_GLOVEDETECTION_T78 = 78,
    DRV_MXT336T_OBJECT_SPT_TOUCHEVENTTRIGGER_T79 = 79,
    DRV_MXT336T_OBJECT_PROCI_RETRANSMISSIONCOMPENSATION_T80 = 80,
    DRV_MXT336T_OBJECT_PROCI_UNLOCKGESTURE_T81 = 81,
    DRV_MXT336T_OBJECT_SPT_NOISESUPEXTENSION_T82 = 82,
    DRV_MXT336T_OBJECT_ENVIRO_LIGHTSENSING_T83 = 83,
    DRV_MXT336T_OBJECT_PROCI_GESTUREPROCESSOR_T84 = 84,
    DRV_MXT336T_OBJECT_PEN_ACTIVESTYLUSPOWER_T85 = 85,
    DRV_MXT336T_OBJECT_PROCG_NOISESUPACTIVESTYLUS_T86 = 86,
    DRV_MXT336T_OBJECT_PEN_ACTIVESTYLUSDATA_T87 = 87,
    DRV_MXT336T_OBJECT_PEN_ACTIVESTYLUSRECEIVE_T88 = 88,
    DRV_MXT336T_OBJECT_PEN_ACTIVESTYLUSTRANSMIT_T89 = 89,
    DRV_MXT336T_OBJECT_PEN_ACTIVESTYLUSWINDOW_T90 = 90,
    DRV_MXT336T_OBJECT_DEBUG_CUSTOMDATACONFIG_T91 = 91,
    DRV_MXT336T_OBJECT_PROCI_SYMBOLGESTUREPROCESSOR_T92 = 92,
    DRV_MXT336T_OBJECT_PROCI_TOUCHSEQUENCELOGGER_T93 = 93,
    DRV_MXT336T_OBJECT_SPT_PTCCONFIG_T95 = 95,
    DRV_MXT336T_OBJECT_SPT_PTCTUNINGPARAMS_T96 = 96,
    DRV_MXT336T_OBJECT_TOUCH_PTCKEYS_T97 = 97,
    DRV_MXT336T_OBJECT_PROCG_PTCNOISESUPPRESSION_T98 = 98,
    DRV_MXT336T_OBJECT_PROCI_KEYGESTUREPROCESSOR_T99 = 99,
    DRV_MXT336T_OBJECT_TOUCH_MULTITOUCHSCREEN_T100 = 100,
    DRV_MXT336T_OBJECT_SPT_TOUCHSCREENHOVER_T101 = 101,
    DRV_MXT336T_OBJECT_SPT_SELFCAPHOVERCTECONFIG_T102 = 102,
    DRV_MXT336T_OBJECT_PROCI_SCHNOISESUPPRESSION_T103 = 103,
    DRV_MXT336T_OBJECT_SPT_AUXTOUCHCONFIG_T104 = 104,
    DRV_MXT336T_OBJECT_SPT_DRIVENPLATEHOVERCONFIG_T105 = 105,
    DRV_MXT336T_OBJECT_SPT_ACTIVESTYLUSMMBCONFIG_T106 = 106,
    DRV_MXT336T_OBJECT_PROCI_ACTIVESTYLUS_T107 = 107,
    DRV_MXT336T_OBJECT_PROCG_NOISESUPSELFCAP_T108 = 108,
    DRV_MXT336T_OBJECT_SPT_SELFCAPGLOBALCONFIG_T109 = 109,
    DRV_MXT336T_OBJECT_SPT_SELFCAPTUNINGPARAMS_T110 = 110,
    DRV_MXT336T_OBJECT_SPT_SELFCAPCONFIG_T111 = 111,
    DRV_MXT336T_OBJECT_PROCI_SELFCAPGRIPSUPPRESSION_T112 = 112,
    DRV_MXT336T_OBJECT_SPT_PROXMEASURECONFIG_T113 = 113,
    DRV_MXT336T_OBJECT_SPT_ACTIVESTYLUSMEASCONFIG_T114 = 114,
    DRV_MXT336T_OBJECT_PROCI_SYMBOLGESTURE_T115 = 115,
    DRV_MXT336T_OBJECT_SPT_SYMBOLGESTURECONFIG_T116 = 116,
    DRV_MXT336T_OBJECT_GEN_INFOBLOCK16BIT_T254 = 254,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T220 = 220,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T221 = 221,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T222 = 222,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T223 = 223,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T224 = 224,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T225 = 225,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T226 = 226,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T227 = 227,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T228 = 228,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T229 = 229,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T230 = 230,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T231 = 231,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T232 = 232,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T233 = 233,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T234 = 234,
```

```
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T235 = 235,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T236 = 236,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T237 = 237,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T238 = 238,
    DRV_MXT336T_OBJECT_SPT_PROTOTYPE_T239 = 239,
    DRV_MXT336T_OBJECT_RESERVED_T255 = 255
} DRV_MXT336T_OBJECT_TYPE;
```

Description

Enumeration DRV_MXT336T_OBJECT_TYPE

The MAxtouch devices follow a Object protocol for their driver implementation. This makes it possible to implement a generic driver for many maxtouch devices. The device communicates the different properties or status like touch messages etc with the driver through an Object table. The different types of objects associated with the maxtouch device are listed in the enum below.

Remarks

None.

DRV_MXT_HANDLE_INVALID Macro

Definition of an invalid handle.

File

drv_mxt.h

C

```
#define DRV_MXT_HANDLE_INVALID ((DRV_MXT_HANDLE)(-1))
```

Description

MXT Driver Invalid Handle

This is the definition of an invalid handle. An invalid handle is is returned by DRV_MXT_Open() and DRV_MXT_Close() functions if the request was not successful.

Remarks

None.

_DRV_MXT336T_H Macro

File

drv_mxt336t.h

C

```
#define _DRV_MXT336T_H
```

Description

This is macro _DRV_MXT336T_H.

DRV_MXT_I2C_MASTER_READ_ID Macro

MXT input read, I2C address from where master reads touch input data.

File

drv_mxt.h

C

```
#define DRV_MXT_I2C_MASTER_READ_ID 0x4B
```

Description

MXT Driver Module Master Input Read I2C address

This constant defines the MXT touch input read I2C address. This address is used as I2C address to read Touch input from MXT Touch controller.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific data sheets.

DRV_MXT_I2C_MASTER_WRITE_ID Macro

MXT command register write, I2C address where master sends the commands.

File

drv_mxt.h

C

```
#define DRV_MXT_I2C_MASTER_WRITE_ID 0x4A
```

Description

MXT Driver Module Master Command Write I2C Address

This constant defines the MXT command register I2C write address. This address is used as I2C address to write commands into MXT Touch controller register.

Remarks

This constant should be used in place of hard-coded numeric literals. This value is derived from device-specific data sheets.

DRV_MXT_I2C_READ_FRAME_SIZE Macro

I2C Frame size for reading MXT touch input.

File

drv_mxt.h

C

```
#define DRV_MXT_I2C_READ_FRAME_SIZE 7
```

Description

MXT Driver Module I2C Frame Size

This constant identifies the size of I2C frame required to read from MXT touch controller. MXT notifies the availability of input data through interrupt pin.

Remarks

This constant should be used in place of hard-coded numeric literals. This value is derived from device-specific data sheets.

DRV_MXT_INDEX_0 Macro

MXT driver index definitions.

File

drv_mxt.h

C

```
#define DRV_MXT_INDEX_0 0
```

Description

MXT Driver Module Index Numbers

These constants provide the MXT driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_MXT_Initialize and DRV_MXT_Open functions to identify the driver instance in use.

DRV_MXT_INDEX_1 Macro

File

drv_mxt.h

C

```
#define DRV_MXT_INDEX_1 1
```

Description

This is macro DRV_MXT_INDEX_1.

DRV_MXT_INDEX_COUNT Macro

Number of valid Touch controller MXT driver indices.

File

drv_mxt.h

C

```
#define DRV_MXT_INDEX_COUNT 2
```

Description

MXT Driver Module Index Count

This constant identifies the number of valid Touch Controller MXT driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals. This value is derived from device-specific header files defined as part of the peripheral libraries.

DRV_MXT336T_HANDLE_INVALID Macro

Definition of an invalid handle.

File

drv_mxt336t.h

C

```
#define DRV_MXT336T_HANDLE_INVALID ((DRV_MXT336T_HANDLE)(-1))
```

Description

MXT336T Driver Invalid Handle

This is the definition of an invalid handle. An invalid handle is is returned by DRV_MXT336T_Open() and DRV_MXT336T_Close() functions if the request was not successful.

Remarks

None.

DRV_MXT336T_I2C_FRAME_SIZE Macro

I2C Frame size for reading MXT336T touch input.

File

drv_mxt336t.h

C

```
#define DRV_MXT336T_I2C_FRAME_SIZE 32
```

Description

MXT336T Driver Module I2C Frame Size

This constant identifies the size of I2C frame required to read from MXT336T touch controller. MXT336T notifies the availability of input data through interrupt pin.

Remarks

This constant should be used in place of hard-coded numeric literals. This value is derived from device-specific data sheets.

DRV_MXT336T_I2C_MASTER_READ_ID Macro

MXT336T input read, I2C address from where master reads touch input data.

File

drv_mxt336t.h

C

```
#define DRV_MXT336T_I2C_MASTER_READ_ID 0x95
```

Description

MXT336T Driver Module Master Input Read I2C address

This constant defines the MXT336T touch input read I2C address. This address is used as I2C address to read Touch input from MXT336T Touch controller.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific data sheets.

DRV_MXT336T_I2C_MASTER_WRITE_ID Macro

MXT336T command register write, I2C address where master sends the commands.

File

drv_mxt336t.h

C

```
#define DRV_MXT336T_I2C_MASTER_WRITE_ID 0x94
```

Description

MXT336T Driver Module Master Command Write I2C Address

This constant defines the MXT336T command register I2C write address. This address is used as I2C address to write commands into MXT336T Touch controller register.

Remarks

This constant should be used in place of hard-coded numeric literals. This value is derived from device-specific data sheets.

DRV_MXT336T_I2C_READ_ID_FRAME_SIZE Macro

File

drv_mxt336t.h

C

```
#define DRV_MXT336T_I2C_READ_ID_FRAME_SIZE 8
```

Description

This is macro DRV_MXT336T_I2C_READ_ID_FRAME_SIZE.

DRV_MXT336T_INDEX_0 Macro

MXT336T driver index definitions.

File

drv_mxt336t.h

C

```
#define DRV_MXT336T_INDEX_0 0
```

Description

MXT336T Driver Module Index Numbers

These constants provide the MXT336T driver index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the DRV_MXT336T_Initialize and DRV_MXT336T_Open functions to identify the driver instance in use.

DRV_MXT336T_INDEX_1 Macro

File

drv_mxt336t.h

C

```
#define DRV_MXT336T_INDEX_1 1
```

Description

This is macro DRV_MXT336T_INDEX_1.

DRV_MXT336T_INDEX_COUNT Macro

Number of valid Touch controller MXT336T driver indices.

File

drv_mxt336t.h

C

```
#define DRV_MXT336T_INDEX_COUNT 2
```

Description

MXT336T Driver Module Index Count

This constant identifies the number of valid Touch Controller MXT336T driver indices.

Remarks

This constant should be used in place of hard-coded numeric literals. This value is derived from device-specific header files defined as part of the peripheral libraries.

t100_event Enumeration

Types of touch events reported by the Maxtouch Multi touch object

File

drv_mxt.h

C

```
enum t100_event {
  MXT_T100_EVENT_NO_EVENT = 0,
  MXT_T100_EVENT_MOVE = 1,
  MXT_T100_EVENT_UNSUP = 2,
  MXT_T100_EVENT_SUP = 3,
  MXT_T100_EVENT_DOWN = 4,
  MXT_T100_EVENT_UP = 5,
  MXT_T100_EVENT_UNSUPSUP = 6,
  MXT_T100_EVENT_UNSUPUP = 7,
  MXT_T100_EVENT_DOWNSUP = 8,
  MXT_T100_EVENT_DOWNUP = 9
};
```

Description

Enumeration: t100_event

The maxtouch multi touch object DRV_MXT336T_OBJECT_TOUCH_MULTITOUCHSCREEN_T100 return a number of different types of touch events. Each touch event has a return type associated with it. These are listed in this enum. These events are returned in the touch status message associated with the multi touch object.

Remarks

None.

t100_type Enumeration

Types of touch types reported by the Maxtouch Multi touch object

File

drv_mxt.h

C

```
enum t100_type {
  MXT_T100_TYPE_FINGER = 1,
  MXT_T100_TYPE_PASSIVE_STYLUS = 2,
  MXT_T100_TYPE_ACTIVE_STYLUS = 3,
  MXT_T100_TYPE_HOVERING_FINGER = 4,
  MXT_T100_TYPE_GLOVE = 5,
  MXT_T100_TYPE_LARGE_TOUCH = 6
};
```

Description

Enumeration: t100_type

The maxtouch multi touch object DRV_MXT336T_OBJECT_TOUCH_MULTITOUCHSCREEN_T100 return a number of different types of touch types. These are listed in this enum. The touch type is returned in the touch status message associated with the multi touch object.

Remarks

None.

DRV_MXT336T_T100_XRANGE Macro

MXT336T Driver Object Register Adresses for the registers being read in the driver

File

drv_mxt336t.h

C

```
#define DRV_MXT336T_T100_XRANGE 13
```

Description

MXT336T Driver Object Register Adresses for the registers being read in the driver

MXT336T Objects have different registers that contain certain values regarding display resoltuion etc. These register addresses are used to read the values from object tables.

Remarks

This constant should be used in place of hard-coded numeric literals.

This value is derived from device-specific protocol guides.

DRV_MXT336T_T100_YRANGE Macro

File

drv_mxt336t.h

C

```
#define DRV_MXT336T_T100_YRANGE 24
```

Description

This is macro DRV_MXT336T_T100_YRANGE.

Files

Files

Name	Description
drv_mxt.h	Touch controller MXT Driver interface header file.

drv_mxt336t.h	Touch controller MXT336T Driver interface header file.

Description

drv_mxt.h

Touch controller MXT Driver interface header file.

Enumerations

	Name	Description
	t100_event	Types of touch events reported by the Maxtouch Multi touch object
	t100_type	Types of touch types reported by the Maxtouch Multi touch object
	DRV_MXT_MODULE_ID	Number of valid MXT driver indices.
	DRV_MXT_TASK_STATE	Enumeration defining MXT touch controller driver task state.

Functions

	Name	Description
	DRV_MXT_Close	Closes an opened instance of the MXT driver. **Implementation:** Dynamic
	DRV_MXT_Deinitialize	Deinitializes the specified instance of the MXT driver module. **Implementation:** Dynamic
	DRV_MXT_Initialize	Initializes the MXT instance for the specified driver index. **Implementation:** Dynamic
	DRV_MXT_MaxtouchEventCallback	
	DRV_MXT_Open	Opens the specified MXT driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_MXT_ReadRequest	Sends a read request to I2C bus driver and adds the read task to queue. **Implementation:** Dynamic
	DRV_MXT_Status	Provides the current status of the MXT driver module. **Implementation:** Dynamic
	DRV_MXT_Tasks	Maintains the driver's state machine and implements its task queue processing. **Implementation:** Dynamic
	DRV_MXT_TouchDataRead	Notifies the driver that the current touch data has been read
	DRV_MXT_TouchGetX	Returns the x coordinate of touch input. **Implementation:** Dynamic
	DRV_MXT_TouchGetY	Returns the y coordinate of touch input. **Implementation:** Dynamic
	DRV_MXT_TouchStatus	Returns the status of the current touch input.

Macros

	Name	Description
	DRV_MXT_HANDLE_INVALID	Definition of an invalid handle.
	DRV_MXT_I2C_MASTER_READ_ID	MXT input read, I2C address from where master reads touch input data.
	DRV_MXT_I2C_MASTER_WRITE_ID	MXT command register write, I2C address where master sends the commands.
	DRV_MXT_I2C_READ_FRAME_SIZE	I2C Frame size for reading MXT touch input.
	DRV_MXT_INDEX_0	MXT driver index definitions.
	DRV_MXT_INDEX_1	This is macro DRV_MXT_INDEX_1.
	DRV_MXT_INDEX_COUNT	Number of valid Touch controller MXT driver indices.

Structures

	Name	Description
	_DRV_MXT_CLIENT_OBJECT	MXT Driver client object maintaining client data.
	DRV_MXT_CLIENT_OBJECT	MXT Driver client object maintaining client data.
	DRV_MXT_INIT	Defines the data required to initialize or reinitialize the MXT driver
	DRV_MXT_OBJECT	Defines the data structure maintaining MXT driver instance object.
	DRV_MXT_TASK_QUEUE	Defines the MXT Touch Controller driver task data structure.

Types

	Name	Description
	DRV_MXT_HANDLE	Touch screen controller MXT driver handle.

Description

Touch Controller MXT Driver Interface File

This header file describes the macros, data structure and prototypes of the touch controller MXT driver interface.

File Name

drv_MXT.c

drv_mxt336t.h

Touch controller MXT336T Driver interface header file.

Enumerations

	Name	Description
	DRV_MXT336T_OBJECT_TYPE	The enum lists the different objects supported by the maxtouch device.

Functions

	Name	Description
	DRV_MXT336T_Close	Closes an opened instance of the MXT336T driver. **Implementation:** Dynamic
	DRV_MXT336T_CloseObject	Closes an opened instance of the MXT336T client object
	DRV_MXT336T_Deinitialize	Deinitializes the specified instance of the MXT336T driver module. **Implementation:** Dynamic
	DRV_MXT336T_DEVICE_ClientObjectEventHandlerSet	Sets the event handler for a MXT336T client object
	DRV_MXT336T_Initialize	Initializes the MXT336T instance for the specified driver index
	DRV_MXT336T_Open	Opens the specified MXT336T driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_MXT336T_OpenObject	Opens the specified MXT336T object driver instance and returns a handle to it. **Implementation:** Dynamic
	DRV_MXT336T_ReadRequest	Sends a read request to I2C bus driver and adds the read task to queue. **Implementation:** Dynamic
	DRV_MXT336T_Status	Provides the current status of the MXT336T driver module. **Implementation:** Dynamic
	DRV_MXT336T_Tasks	Maintains the driver's state machine and implements its task queue processing. **Implementation:** Dynamic

Macros

	Name	Description
	_DRV_MXT336T_H	This is macro _DRV_MXT336T_H.
	DRV_MXT336T_HANDLE_INVALID	Definition of an invalid handle.
	DRV_MXT336T_I2C_FRAME_SIZE	I2C Frame size for reading MXT336T touch input.
	DRV_MXT336T_I2C_MASTER_READ_ID	MXT336T input read, I2C address from where master reads touch input data.
	DRV_MXT336T_I2C_MASTER_WRITE_ID	MXT336T command register write, I2C address where master sends the commands.
	DRV_MXT336T_I2C_READ_ID_FRAME_SIZE	This is macro DRV_MXT336T_I2C_READ_ID_FRAME_SIZE.
	DRV_MXT336T_INDEX_0	MXT336T driver index definitions.
	DRV_MXT336T_INDEX_1	This is macro DRV_MXT336T_INDEX_1.
	DRV_MXT336T_INDEX_COUNT	Number of valid Touch controller MXT336T driver indices.
	DRV_MXT336T_T100_XRANGE	MXT336T Driver Object Register Adresses for the registers being read in the driver
	DRV_MXT336T_T100_YRANGE	This is macro DRV_MXT336T_T100_YRANGE.

Structures

	Name	Description
	DRV_MXT336T_OBJECT_CLIENT_EVENT_DATA	This structure maintains the information associated with each msg received or event that occurs

Types

	Name	Description
	DRV_MXT336T_CLIENT_CALLBACK	Pointer to a MXT336T client callback function data type.
	DRV_MXT336T_HANDLE	Touch screen controller MXT336T driver handle.
	DRV_MXT336T_INIT	Defines the data required to initialize or reinitialize the MXT336T driver

Description

Touch Controller MXT336T Driver Interface File

This header file describes the macros, data structure and prototypes of the touch controller MXT336T driver interface.

File Name

drv_MXT336T.c

USB Driver Libraries

Common Interface

Provides information on the USB Driver interface that is common to all PIC32 devices.

Description

The USB Driver Common Interface definition specifies the functions and their behavior that a USB Driver must implement so that the driver can be used by the MPLAB Harmony USB Host and Device Stack.

Note: The MPLAB Harmony USB Driver for PIC32MX and PIC32MZ devices implements the USB Driver Common Interface.

The USB Driver Common Interface contains functions that are grouped as follows:

- *Driver System Functions* - These functions are called by MPLAB Harmony to initialize and maintain the operational state of the USB Driver. The system functions can vary between different PIC32 device USB Drivers. As such, the USB Driver Common Interface does not require these functions to be of the same type. These functions are not called by the USB Host or Device Stack and therefore are allowed to (and can) vary across different PIC32 device USB Drivers. A description of these functions, along with a description of how to initialize the USB Driver for Host, Device or Dual Role operation, is provided in the specific PIC32 device USB Driver help section (see PIC32MX USB Driver and PIC32MZ USB Driver).
- *Driver General Client Functions* -These functions are called by the USB Host or Device Stack to gain access to the driver
- *Driver Host Mode Client Functions* - These functions are called exclusively by the USB Host Stack to operate and access the USB as a Host
- *Driver Device Mode Client Functions* - These functions are called exclusively by the USB Device Stack to operate and access the USB as a Device

The USB Driver Common Interface is defined in the `<install-dir>\framework\driver\usb\drv_usb.h` file. This file contains the data types and structures that define the interface. Specifically, the DRV_USB_HOST_INTERFACE structure, contained in this file, is the common interface for USB Driver Host mode functions. It is a structure of function pointers, pointing to functions that define the Driver Host mode Client functions. The following code example shows this structure and the function pointer it contains.

```
// *****************************************************************************
/* USB Driver Client Functions Interface (For Host mode)

  Summary:
    Group of function pointers to the USB Driver Host mode Client Functions.

  Description:
    This structure is a group of function pointers pointing to the USB Driver
    Host mode Client routines. The USB Driver should export this group of
    functions so that the Host layer can access the driver functionality.

  Remarks:
    None.
*/
```

```
typedef struct
{
    /* This is a pointer to the driver Open function. This function may be
     * called twice in a Dual Role application, once by the Host Stack and then
     * by the Device Stack */
    DRV_HANDLE (*open)(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);

    /* This is pointer to the driver Close function */
    void (*close)(DRV_HANDLE handle);

    /* This is a pointer to the event call back set function */
    void (*eventHandlerSet)(DRV_HANDLE handle, uintptr_t hReferenceData,
           DRV_USB_EVENT_CALLBACK eventHandler);

    /* This is a pointer to the Host IRP submit function */
    USB_ERROR (*hostIRPSubmit)(DRV_USB_HOST_PIPE_HANDLE pipeHandle, USB_HOST_IRP * irp);

    /* This is a pointer to the Host IRP Cancel all function */
    void (*hostIRPCancel)(USB_HOST_IRP * irp);

    /* This is pointer to the Host event disable function */
    bool (*hostEventsDisable)(DRV_HANDLE handle);

    /* This is a pointer to the Host event enable function */
    void (*hostEventsEnable)(DRV_HANDLE handle, bool eventContext);

    /* This is a pointer to the Host pipe setup function */
    DRV_USB_HOST_PIPE_HANDLE (*hostPipeSetup)
    (
        DRV_HANDLE client,
        uint8_t deviceAddress,
        USB_ENDPOINT endpointAndDirection,
        uint8_t hubAddress,
        uint8_t hubPort,
        USB_TRANSFER_TYPE pipeType,
        uint8_t bInterval,
        uint16_t wMaxPacketSize,
        USB_SPEED speed
    );

    /* This is a pointer to the Host Pipe Close function */
    void (*hostPipeClose)(DRV_USB_HOST_PIPE_HANDLE pipeHandle);

    /* This is a pointer to the Host Root Hub functions */
    DRV_USB_ROOT_HUB_INTERFACE rootHubInterface;

} DRV_USB_HOST_INTERFACE;
```

The DRV_USB_DEVICE_INTERFACE structure, contained in this file, is the common interface for USB Driver Device mode functions. It is a structure of function pointers, pointer to functions that define the Driver Device mode Client functions. The following code example shows this structure and the function pointer it contains.

```
// *****************************************************************************
/* USB Driver Client Functions Interface (For Device Mode)

  Summary:
    Group of function pointers to the USB Driver Device Mode Client Functions.

  Description:
    This structure is a group of function pointers pointing to the USB Driver
    Device Mode Client routines. The USB Driver should export this group of
    functions so that the Device Layer can access the driver functionality.

  Remarks:
    None.
*/

typedef struct
{
    /* This is a pointer to the driver Open function */
    DRV_HANDLE (*open)(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

```
    /* This is pointer to the driver Close function */
    void (*close)(DRV_HANDLE handle);

    /* This is a pointer to the event call back set function */
    void (*eventHandlerSet)(DRV_HANDLE handle, uintptr_t hReferenceData,
           DRV_USB_EVENT_CALLBACK eventHandler);

    /* This is a pointer to the device address set function */
    void (*deviceAddressSet)(DRV_HANDLE handle, uint8_t address);

    /* This is a pointer to the device current speed get function */
    USB_SPEED (*deviceCurrentSpeedGet)(DRV_HANDLE handle);

    /* This is a pointer to the SOF Number get function */
    uint16_t (*deviceSOFNumberGet)(DRV_HANDLE handle);

    /* This is a pointer to the device attach function */
    void (*deviceAttach)(DRV_HANDLE handle);

    /* This is a pointer to the device detach function */
    void (*deviceDetach)(DRV_HANDLE handle);

    /* This is a pointer to the device endpoint enable function */
    USB_ERROR (*deviceEndpointEnable)(DRV_HANDLE handle, USB_ENDPOINT endpoint,
                USB_TRANSFER_TYPE transferType, uint16_t endpointSize);

    /* This is a pointer to the device endpoint disable function */
    USB_ERROR (*deviceEndpointDisable)(DRV_HANDLE handle, USB_ENDPOINT endpoint);

    /* This is a pointer to the device endpoint stall function */
    USB_ERROR (*deviceEndpointStall)(DRV_HANDLE handle, USB_ENDPOINT endpoint);

    /* This is a pointer to the device endpoint stall clear function */
    USB_ERROR (*deviceEndpointStallClear)(DRV_HANDLE handle, USB_ENDPOINT endpoint);

    /* This is pointer to the device endpoint enable status query function */
    bool (*deviceEndpointIsEnabled)(DRV_HANDLE handle, USB_ENDPOINT endpoint);

    /* This is pointer to the device endpoint stall status query function */
    bool (*deviceEndpointIsStalled)(DRV_HANDLE handle, USB_ENDPOINT endpoint);

    /* This is a pointer to the device IRP submit function */
    USB_ERROR (*deviceIRPSubmit)(DRV_HANDLE handle, USB_ENDPOINT endpoint,
                USB_DEVICE_IRP * irp);

    /* This is a pointer to the device IRP Cancel all function */
    USB_ERROR (*deviceIRPCancelAll)(DRV_HANDLE handle, USB_ENDPOINT endpoint);

    /* This is a pointer to the device remote wakeup start function */
    void (*deviceRemoteWakeupStart)(DRV_HANDLE handle);

    /* This is a pointer to the device remote wakeup stop function */
    void (*deviceRemoteWakeupStop)(DRV_HANDLE handle);

    /* This is a pointer to the device Test mode enter function */
    USB_ERROR (*deviceTestModeEnter)(DRV_HANDLE handle, USB_TEST_MODE_SELECTORS testMode);

} DRV_USB_DEVICE_INTERFACE;
```

Both of these structures also contain pointers to General Client functions. The specific PIC32 device USB Driver allocates and initializes such a structure. The following code example shows how the PIC32MX USB Host mode Driver allocates and initializes the DRV_USB_HOST_INTERFACE structure. This code is contained in the <install-dir>\framework\driver\usb\usbhs\src\dynamic\drv_usbfs_host.c file.

```
/*****************************************************
 * This structure is a set of pointer to the USBFS driver
 * functions. It is provided to the Host layer as the
 * interface to the driver.
 * ****************************************************/
```

```
DRV_USB_HOST_INTERFACE gDrvUSBFSHostInterface =
{
    .open = DRV_USBFS_Open,
    .close = DRV_USBFS_Close,
    .eventHandlerSet = DRV_USBFS_ClientEventCallBackSet,
    .hostIRPSubmit = DRV_USBFS_HOST_IRPSubmit,
    .hostIRPCancel = DRV_USBFS_HOST_IRPCancel,
    .hostPipeSetup = DRV_USBFS_HOST_PipeSetup,
    .hostPipeClose = DRV_USBFS_HOST_PipeClose,
    .hostEventsDisable = DRV_USBFS_HOST_EventsDisable,
    .hostEventsEnable = DRV_USBFS_HOST_EventsEnable,
    .rootHubInterface.rootHubPortInterface.hubPortReset = DRV_USBFS_HOST_ROOT_HUB_PortReset,
    .rootHubInterface.rootHubPortInterface.hubPortSpeedGet =
                                                    DRV_USBFS_HOST_ROOT_HUB_PortSpeedGet,
    .rootHubInterface.rootHubPortInterface.hubPortResetIsComplete =
                                                    DRV_USBFS_HOST_ROOT_HUB_PortResetIsComplete,
    .rootHubInterface.rootHubPortInterface.hubPortSuspend = DRV_USBFS_HOST_ROOT_HUB_PortSuspend,
    .rootHubInterface.rootHubPortInterface.hubPortResume = DRV_USBFS_HOST_ROOT_HUB_PortResume,
    .rootHubInterface.rootHubMaxCurrentGet = DRV_USBFS_HOST_ROOT_HUB_MaximumCurrentGet,
    .rootHubInterface.rootHubPortNumbersGet = DRV_USBFS_HOST_ROOT_HUB_PortNumbersGet,
    .rootHubInterface.rootHubSpeedGet = DRV_USBFS_HOST_ROOT_HUB_BusSpeedGet,
    .rootHubInterface.rootHubInitialize = DRV_USBFS_HOST_ROOT_HUB_Initialize,
    .rootHubInterface.rootHubOperationEnable = DRV_USBFS_HOST_ROOT_HUB_OperationEnable,
    .rootHubInterface.rootHubOperationIsEnabled = DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled,
};
```

Similarly, the PIC32MX USB Device mode Driver allocates and initializes the DRV_USB_DEVICE_INTERFACE structure. This can be reviewed in the <install-dir>\framework\driver\usb\usbhs\src\dynamic\drv_usbfs_device.c file.

```
/*************************************************
 * This structure is a pointer to a set of USB Driver
 * Device mode functions. This set is exported to the
 * Device Layer when the Device Layer must use the
 * PIC32MX USB Controller.
 *************************************************/

DRV_USB_DEVICE_INTERFACE gDrvUSBFSDeviceInterface =
{
    .open = DRV_USBFS_Open,
    .close = DRV_USBFS_Close,
    .eventHandlerSet = DRV_USBFS_ClientEventCallBackSet,
    .deviceAddressSet = DRV_USBFS_DEVICE_AddressSet,
    .deviceCurrentSpeedGet = DRV_USBFS_DEVICE_CurrentSpeedGet,
    .deviceSOFNumberGet = DRV_USBFS_DEVICE_SOFNumberGet,
    .deviceAttach = DRV_USBFS_DEVICE_Attach,
    .deviceDetach = DRV_USBFS_DEVICE_Detach,
    .deviceEndpointEnable = DRV_USBFS_DEVICE_EndpointEnable,
    .deviceEndpointDisable = DRV_USBFS_DEVICE_EndpointDisable,
    .deviceEndpointStall = DRV_USBFS_DEVICE_EndpointStall,
    .deviceEndpointStallClear = DRV_USBFS_DEVICE_EndpointStallClear,
    .deviceEndpointIsEnabled = DRV_USBFS_DEVICE_EndpointIsEnabled,
    .deviceEndpointIsStalled = DRV_USBFS_DEVICE_EndpointIsStalled,
    .deviceIRPSubmit = DRV_USBFS_DEVICE_IRPSubmit,
    .deviceIRPCancelAll = DRV_USBFS_DEVICE_IRPCancelAll,
    .deviceRemoteWakeupStop = DRV_USBFS_DEVICE_RemoteWakeupStop,
    .deviceRemoteWakeupStart = DRV_USBFS_DEVICE_RemoteWakeupStart,
    .deviceTestModeEnter = NULL


};
```

A pointer to the DRV_USB_HOST_INTERFACE structure is passed to the USB Host Stack as part of USB Host Stack initialization. The following code example shows how this is done.

```
/*********************************************************************************
 * This is a table of the USB Host mode drivers that this application will
 * support. Also contained in the driver index. In this example, the
 * application will want to use instance 0 of the PIC32MX USB Full-Speed driver.
 * ********************************************************************************/
const USB_HOST_HCD hcdTable =
{
    .drvIndex = DRV_USBFS_INDEX_0,
```

```
    .hcdInterface = DRV_USBFS_HOST_INTERFACE
};

/* Here the pointer to the USB Driver Common Interface is provided to the USB
 * Host Layer via the hostControllerDrivers member of the Host Layer
 * Initialization data structure. */
const USB_HOST_INIT usbHostInitData =
{
    .nTPLEntries = 1 ,
    .tplList = (USB_HOST_TPL_ENTRY *)USBTPList,
    .hostControllerDrivers = (USB_HOST_HCD *)&hcdTable

};
```

A pointer to the DRV_USB_DEVICE_INTERFACE structure is passed to the USB Device Stack as part of the USB Device Stack initialization. The Host Stack and Device Stack then access the driver functions through the function pointers contained in these structures.

The Driver General Client, Host mode and Device mode Client functions are described in this section. Any references to a USB Driver Client in the following sections, implies the client is a USB Host Stack and/or the USB Device Stack.

Driver General Client Functions

Provides information on the General Client functions for the USB Driver.

Description

The DRV_USB_HOST_INTERFACE and the DRV_USB_DEVICE_INTERFACE structures contain pointers to the USB Driver's General Client functions. These functions are not specific to the operation mode (Host, Device, or Dual Role) of the driver. A USB Driver must implement these functions and ensure that the Host or Device Stack can access these functions through the driver's common interface structures. The common interface contains three general client functions:

- Driver Open Function
- Driver Close Function
- Driver Event Handler Set Function

Driver Open Function

The `open` member of the DRV_USB_HOST_INTERFACE and the DRV_USB_DEVICE_INTERFACE structures should point to the USB Driver Open function. The signature of the Open function is as follows:

```
DRV_HANDLE (*open)(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

The USB Driver Open function must match this signature. The Driver Client uses the USB Driver index (drvIndex) to specify the instance of the USB module that Host Stack or the Device Stack should open. The USB Driver should ignore the `intent` parameter. The function should return a driver handle. If the driver is not ready to be opened, it should return an invalid handle (DRV_HANDLE_INVALID). In such a case, the client will continue trying to open the driver by calling the Open function again. The driver may also fail to open for an invalid `index` parameter or if USB module is in an error condition.

When supporting Dual Role operation, both the Host Stack and Device Stack will call the Driver Open function in one application. The USB Driver must support multiple calls to the Open function in the same application. The Open function should be thread-safe.

Driver Close Function

The `close` member of the DRV_USB_HOST_INTERFACE and the DRV_USB_DEVICE_INTERFACE structures should point to the USB Driver Close function. The signature of the Close function is as follows:

```
void (*close)(DRV_HANDLE handle);
```

The USB Driver Close function must match this signature. The Driver Client passes the handle obtained from the Driver Open function as a parameter to the close. The USB Host Stack or USB Device Stack will close the driver only when the stack is deinitialized (which is typically a rare case). The USB Driver should deallocate any client-related resources in the Close function. If the specified driver handle is not valid, the Close function should not have any side effects. The USB Driver expects the Close function to be called from the context of the thread in which the driver was opened; therefore, this function *is not* expected to be thread-safe.

Driver Event Handler Set Function

The `eventHandlerSet` member of the DRV_USB_HOST_INTERFACE and the DRV_USB_DEVICE_INTERFACE structures should point to the USB Driver Event Handler Set function. The signature of the Event Handler Set function is as follows:

```
void (*eventHandlerSet)(DRV_HANDLE handle, uintptr_t hReferenceData, DRV_USB_EVENT_CALLBACK eventHandler);
```

The USB Driver Event Handler Set function must match this signature. The signature of the Client Event Handling function should match DRV_USB_EVENT_CALLBACK. The USB Driver calls this function when it must communicate USB events to the client. The client can set the `eventHandler` parameter to NULL if it does not want to receive USB Driver events. The client will receive Host mode events if the USB Driver is operating in Host mode. It will receive Device mode events if the USB Driver is operating in Device mode. The DRV_USB_EVENT type enumeration contains all the possible events that the USB Driver would generate. The following code example shows the enumeration.

```
// *****************************************************************************
/* USB Driver Events Enumeration
```

```
  Summary:
    Identifies the different events that the USB Driver provides.

  Description:
    Identifies the different events that the USB Driver provides. The USB Driver
    should be able to provide these events.

  Remarks:
    None.
*/

typedef enum
{
    /* Bus error occurred and was reported. This event can be generated in both
     * Host and Device mode. */
    DRV_USB_EVENT_ERROR = 1,

    /* Host has issued a device Reset. This event occurs only in Device mode */
    DRV_USB_EVENT_RESET_DETECT,

    /* Resume detected while USB in suspend mode. This event can be generated in
     * both Host and Device mode. In Host mode, the events occurs when a remote
     * wakeup capable device has generated resume signaling. In Device mode,
     * this event will occur when the Host has issued resume signaling. */
    DRV_USB_EVENT_RESUME_DETECT,

    /* This event is generated in Device mode only. It occurs when the Host
     * suspends the bus and the bus goes idle. */
    DRV_USB_EVENT_IDLE_DETECT,

    /* This event is generated in Host mode and Device mode. In Host mode, this
     * event occurs when the device has stalled the Host. In Device mode, this
     * event occurs when the Host has accessed a stalled endpoint thus
     * triggering the device to send a STALL to the Host. */
    DRV_USB_EVENT_STALL,

    /* This event is generated in Host mode and Device mode. In Device mode,
     * this event occurs when a SOF has been generated by the Host. In Host
     * mode, this event occurs when controller is about to generate an SOF.
     * */
    DRV_USB_EVENT_SOF_DETECT,

    /* This event is generated in Device mode when the VBUS voltage is above
     * VBUS session valid. */
    DRV_USB_EVENT_DEVICE_SESSION_VALID,

    /* This event is generated in Device mode when the VBUS voltage falls
     * below VBUS session valid. */
    DRV_USB_EVENT_DEVICE_SESSION_INVALID,

} DRV_USB_EVENT;
```

This completes the discussion on the Driver General Client Functions.

Driver Host Mode Client Functions

Provides information on the Host mode Client functions for the USB Driver.

Description

The DRV_USB_HOST_INTERFACE structure contains pointers to the USB Driver's Host mode Client functions. These functions are only applicable when the USB module is operating as a USB Host. Along with the function pointers to the driver's Host mode specific functions, the DRV_USB_HOST_INTERFACE structure also contains another structure of function pointers of the type DRV_USB_ROOT_HUB_INTERFACE. This structure contains function pointers to the USB Driver's Root Hub functions. A USB Driver must implement these functions and ensure that the Host Stack can access these functions through the driver's DRV_USB_HOST_INTERFACE structure. The Driver Host mode Client functions in the DRV_USB_HOST_INTERFACE structure are:

- Driver Host Pipe Setup Function
- Driver Host Pipe Close Function

- Driver Host Events Disable Function
- Driver Host Events Enable Function
- Driver Host IRP Submit Function
- Driver Host IRP Cancel Function

Driver Host Pipe Setup Function

The `hostPipeSetup` member of the DRV_USB_HOST_INTERFACE structure should point to the USB Driver Host Pipe Setup function. The signature of the Host Pipe Setup function is as follows:

```
DRV_USB_HOST_PIPE_HANDLE (*hostPipeSetup)  ( DRV_HANDLE client,  uint8_t deviceAddress,
                           USB_ENDPOINT endpointAndDirection, uint8_t hubAddress,
                           uint8_t hubPort, USB_TRANSFER_TYPE pipeType, uint8_t bInterval,
                           uint16_t wMaxPacketSize, USB_SPEED speed);
```

The USB Driver Host mode Pipe Setup function must match this signature. The USB Host Stack calls this function to create a communication pipe to the attached device. The function parameters define the property of this communication pipe. The `driverHandle` parameter is the handle to the driver obtained through the driver Open function. The `deviceAddress` and the `endpointAddress` parameters specify the address of the USB device and the endpoint on this device to which this pipe must connect.

If the device is connected to the Host though a hub, `hubAddress` and `hubPort` must specify the address of the hub and port to which the device is connected. The USB Driver will use these parameters to schedule split transactions if the target device is a Low-Speed or Full-Speed device and is connected to the Host through a high-speed hub. If the device is connected directly to the Host, these parameters should be set to zero ('0').

The `pipeType` parameter specifies the type of USB transfers that this pipe would support. The `bInterval` parameter is interpreted as per the USB 2.0 Specification based on the transfer type and the speed of the pipe. The `wMaxPacketSize` parameter defines the maximum size of a transaction that the driver should use while transporting a transfer on the pipe. The Host layer will use the information obtained from the USB device descriptors of the attached device to decide the `wMaxPacketSize` parameter.

The Driver Host Pipe Setup function should be thread-safe, but does not have to be event safe. The Host layer (or the Host Client Drivers) will not, and should not attempt to create a pipe in an interrupt, and therefore, an event context. The function should return DRV_USB_PIPE_HANDLE_INVALID if the driver could not open the pipe. The driver may not be able to open a pipe due to incorrect function parameters or due to lack of resources.

Driver Host Pipe Close Function

The `hostPipeClose` member of the DRV_USB_HOST_INTERFACE structure should point to the USB Driver Host Pipe Close function. The signature of the Host Pipe Close function is as follows:

```
void (*hostPipeClose)(DRV_USB_HOST_PIPE_HANDLE pipeHandle);
```

The USB Driver Host mode Pipe Close function must match this signature. The USB Host Stack calls this function to close communication pipes. The `pipeHandle` parameter is the pipe handle obtained from the Pipe Setup function. The Host Client Driver typically closes pipes when a device detach was detected. The Client Driver may also close pipes when a device configuration needs to change or when the Client Driver is being unloaded by the Host. The Pipe Close function has no side effect if the pipe handle is invalid. Closing the pipe will abort all I/O Request Packets (IRP) that are scheduled on the pipe. Any transaction in progress will complete. The IRP callback functions for each IRP scheduled in the pipe will be called with a USB_HOST_IRP_STATUS_ABORTED status.

The USB Driver Pipe Close function must be thread-safe and event-safe. The latter requirement allows the Pipe Close function to be called in the context of the device detach Interrupt Service Routine.

Driver Host Event Disable Function

The `hostEventsDisable` member of the DRV_USB_HOST_INTERFACE structure should point to the USB Driver Host mode Driver Events Disable function. The signature of the Events Disable function is as follows:

```
bool (*hostEventsDisable)(DRV_HANDLE handle);
```

The USB Driver Host mode Driver Events Disable function must match this signature. The Host Stack will call this function when it wants to execute a section of code that should not be interrupted by the USB Driver. Calling this function should disable USB Driver event generation. The `handle` parameter is set to the driver handle obtained via the driver Open function. The function will return the present state of the event generation, whether it is enabled or disabled. The Host Stack will pass this value to the USB Driver Host mode Driver Events Enable function when it needs to enable the driver events.

Driver Host Events Enable Function

The `hostEventsEnable` member of the DRV_USB_HOST_INTERFACE structure should point to the USB Driver Host mode Driver Events Enable function. The signature of the events enable function is as follows:

```
void (*hostEventsEnable)(DRV_HANDLE handle, bool eventContext);
```

The USB Driver Host mode Driver Events Enable function must match this signature. The USB Host Stack calls this function to re-enable the USB Driver Host mode Events (if they were enabled) after it called the USB Driver Host mode Events Disable function to disable driver events. The `handle` parameter is set to the driver handle obtained via the driver Open function. The `eventContext` parameter is set to the value returned by the Host mode Driver Events Disable function. The USB Driver will use the `eventContext` parameter to restore the event generation status (enabled or disabled) to what it was when the USB Driver Host mode Driver Events Disable function was called.

Driver Host IRP Submit Function

The `hostIRPSubmit` member of the DRV_USB_HOST_INTERFACE structure should point to the USB Driver Host IRP Submit function. The

signature of the IRP Submit function is as follows:
```
USB_ERROR (*hostIRPSubmit)(DRV_USB_HOST_PIPE_HANDLE pipeHandle, USB_HOST_IRP * irp);
```

The USB Driver Host IRP Submit function must match this signature. The Host Stack calls this function to submit an IRP to the USB Driver. The USB Driver provides this mechanism to transfer data between the Host Stack and the attached device. The `pipeHandle` parameter should be set to the pipe handle obtained by the Pipe Setup function. The pipe handle specifies the pipe, and therefore, the target device, endpoint, speed and transfer type, on which the I/O must be processed. The `irp` parameter should point to the IRP data structure. The IRP data structure will transport an entire transfer over the pipe. The USB Driver will split up the transfer into transactions based on the parameters specified at the time of pipe creation. This process does not require Host Stack intervention.

The function will return USB_ERROR_HOST_PIPE_INVALID if the pipe handle is not valid. It will return USB_ERROR_OSAL_FUNCTION if an error occurred while performing a RTOS-related operation. It will return USB_ERROR_NONE if the IRP was submitted successfully.

The USB Driver will queue the IRP if there is already an IRP being processed on the pipe. The completion of the IRP processing is indicated by the USB Driver calling the IRP Callback function specified within the IRP. The Host IRP Submit function must be thread-safe and IRP callback-safe. The Host Stack may resubmit the IRP within the IRP Callback function. The IRP Callback function itself executes within an interrupt context. The completion status of the IRP will be available in the `status` member of the IRP when the IRP callback function is invoked.

Driver Host IRP Cancel Function

The `hostIRPCancel` member of the DRV_USB_HOST_INTERFACE structure should point to the USB Driver Host IRP Cancel function. The signature of the IRP Cancel function is as follows
```
void (*hostIRPCancel)(USB_HOST_IRP * irp);
```

The USB Driver Host IRP Cancel function must match this signature. The Host Stack and Host Client Drivers will call this function to cancel an IRP that was submitted. The IRP will be aborted successfully if it is not in progress. If the IRP processing has begun, the on-going transaction will complete and pending transactions in the transfer will be aborted. In either case, the IRP Callback function will be called with the IRP status as USB_HOST_IRP_STATUS_ABORTED.

Driver Host USB Root Hub Port Interface

Provides information on the Root Hub Port interface of the USB Host Driver.

Description

The `rootHubPortInterface` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub Port functions. The data type of this member is USB_HUB_INTERFACE. This data type is a structure containing function pointers pointing to the port control functions of the root hub. The USB Driver must assign the function pointers in this structure to the root hub port control functions. These same functions are also exported by a Hub Driver to the USB Host Stack, which allow the Host Stack to control a device regardless of whether it is connected to a root hub or an external hub. The port functions are valid only when a device is attached to the port. The behavior of these functions on a port to which no device is connected is not defined. Descriptions of the port control functions are provided, which include:

- Driver Host Hub Port Reset Function
- Driver Host Hub Port Reset Completion Status Function
- Driver Host Hub Port Suspend Function
- Driver Host Hub Port Resume Function
- Driver Host Hub Port Speed Get Function

Driver Host Hub Port Reset Function

The `hubPortReset` member of the USB_HUB_INTERFACE structure should point to the USB Driver Root Hub Port Reset function. The signature of this function is as follows:
```
USB_ERROR (*hubPortReset)(uintptr_t hubAddress, uint8_t port);
```

The USB Driver Root Hub Port Reset function must follow this signature. This function starts reset signaling on the port. If the device is connected to the root hub, the USB Host Stack will set the `hubAddress` parameter to the driver handle obtained through the driver Open function. The USB Host Stack uses the parent identifier provided by the root hub driver when the USB_HOST_DeviceEnumerate function was called to query the driver handle that is linked to this root hub. If the device is connected to an external hub, the `hubAddress` parameter is directly set to the parent identifier.

For the PIC32MX and PIC32MZ USB Drivers, the `port` parameter is ignored. For an external hub, this must be set to the port to which the device is connected. The function returns USB_ERROR_NONE if the function was successful. If the reset signaling is already in progress on the port, calling this function has no effect. The USB Driver will itself time duration of the reset signal. This does not require USB Host Stack intervention. The USB Host Stack will call the port reset completion status function to check if the reset signaling has completed. Calling this function on a port which exists on an external hub will cause the hub driver to issue a control transfer to start the port reset procedure.

Driver Host Hub Port Reset Completion Status Function

The `hubPortResetIsComplete` member of the USB_HUB_INTERFACE structure should point to the USB Driver Root Hub Port Reset Completion Status function. The signature of this function is as follows:
```
bool (*hubPortResetIsComplete)(uintptr_t hubAddress, uint8_t port);
```

The USB Driver Root Hub Port Reset Completion Status function must follow this signature. The USB Host Stack calls this function to check if the port reset sequence that was started on a port has completed. The function returns true if the reset signaling has completed. If the device is connected to the root hub, the USB Host Stack will set the `hubAddress` parameter to the driver handle obtained through the driver Open function.

If the device is connected to an external hub, the `hubAddress` parameter is directly set to the parent identifier.

For the PIC32MX and PIC32MZ USB Drivers, the `port` parameter is ignored. For an external hub, this parameter must be set to the port to which the device is connected.

Driver Host Hub Port Suspend Function

The `hubPortSuspend` member of the USB_HUB_INTERFACE structure should point to the USB Driver Root Hub Port Suspend function. The signature of this function is as follows:

```
USB_ERROR(*hubPortSuspend)(uintptr_t hubAddress, uint8_t port);
```

The USB Driver Root Hub Port Suspend function must follow this signature. The USB Host Stack calls this function to suspend the port. If the device is connected to the root hub, the USB Host Stack will set the `hubAddress` parameter to the driver handle obtained through the driver Open function. If the device is connected to an external hub, the `hubAddress` parameter is directly set to the parent identifier.

For the PIC32MX and PIC32MZ USB Drivers, the `port` parameter is ignored. For an external hub, this parameter must be set to the port to which the device is connected. The function returns USB_ERROR_NONE if the request was successful. Calling this function on a suspended port will not have any effect.

Driver Host Hub Port Resume Function

The `hubPortResume` member of the USB_HUB_INTERFACE structure should point to the USB Driver Root Hub Port Resume function. The signature of this function is as follows:

```
USB_ERROR(*hubPortResume)(uintptr_t hubAddress, uint8_t port);
```

The USB Driver Root Hub Port Resume function must follow this signature. The USB Host Stack calls this function to resume a suspended port. If the device is connected to the root hub, the USB Host Stack will set the `hubAddress` parameter to the driver handle obtained through the driver Open function. If the device is connected to an external hub, the `hubAddress` parameter is directly set to the parent identifier.

For the PIC32MX and PIC32MZ USB Drivers, the `port` parameter is ignored. For an external hub, this parameter must be set to the port to which the device is connected. The function returns USB_ERROR_NONE if the request was successful. Calling this function on a port that is not suspended will not have any effect.

Driver Host Hub Port Speed Get Function

The `hubPortSpeedGet` member of the USB_HUB_INTERFACE structure should point to the USB Driver Root Hub Port Speed Get function. The signature of this function is as follows:

```
USB_SPEED(*hubPortSpeedGet)(uintptr_t hubAddress, uint8_t port);
```

The USB Driver Root Hub Port Speed Get function must follow this signature. The USB Host Stack calls this function to obtain the USB speed of the device that is attached to the port. The Host Stack calls this function only after it has completed reset of the port. If the device is connected to the root hub, the USB Host Stack will set the `hubAddress` parameter to the driver handle obtained through the driver Open function. If the device is connected to an external hub, the `hubAddress` parameter is directly set to the parent identifier.

For the PIC32MX and PIC32MZ USB Drivers, the `port` parameter is ignored. For an external hub, this parameter must be set to the port to which the device is connected. The function returns USB_SPEED_ERROR if the request was not successful. It will return the functional USB speed otherwise.

This concludes the section describing the USB Driver Host mode Client Functions. The USB Driver Device Mode Client Functions are discussed in the next section.

Driver Host Root Hub Interface

Provides information on the Root Hub interface for the USB Host Driver.

Description

The USB Driver Common Interface requires the USB Driver to be operating in Host mode to provide root hub control functions. If the USB peripheral does not contain root hub features in hardware, these features must be emulated in software by the driver. The USB peripheral on PIC32MX and PIC32MZ devices does not contain root hub features; therefore, the USB Driver for these peripherals emulates the root hub functionality in software. The `rootHubInterface` member of the DRV_USB_HOST_INTERFACE structure is a structure of type DRV_USB_ROOT_HUB_INTERFACE. The members of this structure are function pointers to the root hub control functions of the USB Driver.

Along with other Host mode functions, the USB Driver while operating in Host mode must also ensure that the `rootHubInterface` member of DRV_USB_HOST_INTERFACE is set up correctly so that the USB Host Stack can access the root hub functions. Descriptions of the function pointer types in the DRV_USB_ROOT_HUB_INTERFACE include:

- Driver Host Root Hub Speed Get Function
- Driver Host Root Hub Port Numbers Get Function
- Driver Host Root Hub Maximum Current Get Function
- Driver Host Root Hub Operation Enable Function
- Driver Host Root Hub Operation Enable Status Function
- Driver Host Root Hub Initialize Function

Driver Host Root Hub Speed Get Function

The `rootHubSpeedGet` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub Speed Get

function. The signature of this function is as follows:
```
USB_SPEED (*rootHubSpeedGet)(DRV_HANDLE handle);
```

The USB Driver Root Hub Speed Get function must match this signature. The USB Host Stack calls this function to identify the speed at which the root hub is operating. The `handle` parameter is the handle obtained by calling the USB Driver Open function. The operation speed is configured by the USB Driver initialization and depends on the capability of the USB peripheral. For example, the USB peripheral on PIC32MZ devices supports both Hi-Speed and Full-Speed Host mode operation. It can be configured through initialization to only operate at Full-Speed. The Root Hub Speed Get function must return the USB speed at which the USB peripheral is operating. This should not be confused with the speed of the attached device.

Driver Host Root Hub Port Numbers Get Function

The `rootHubPortNumbersGet` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub Port Numbers Get function. The signature of this function is as follows:
```
USB_SPEED (*rootHubSpeedGet)(DRV_HANDLE handle);
```

The USB Driver Root Hub Speed Get function must match this signature. This function should return the number of ports that the root hub contains. On the USB peripheral for both PIC32MZ and PIC32MX devices, this value is always '1'.

Driver Host Root Hub Maximum Current Get Function

The `rootHubMaxCurrentGet` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub Maximum Current Get function. The signature of this function is as follows:
```
uint32_t (*rootHubMaxCurrentGet)(DRV_HANDLE handle);
```

The USB Driver Root Hub Maximum Current Get function must match this signature. This function returns the maximum VBUS current that the root hub can provide. The USB Host Stack calls this function to know the maximum current that the root hub VBUS power supply can provide. This value is then used to determine if the Host can support the current requirements of the attached device. The `handle` parameter is the driver handle obtained by calling the driver Open function.

The PIC32MX and the PIC32MZ USB peripherals cannot supply VBUS. The root hub driver only switches the VBUS supply. The current rating of the VBUS is specified through the USB Driver initialization. The root hub maximum current get function implementation in these drivers returns this value to the Host Stack.

Driver Host Root Hub Operation Enable Function

The `rootHubOperationEnable` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub Operation Enable function. The signature of this function is as follows"
```
void (*rootHubOperationEnable)(DRV_HANDLE handle, bool enable);
```

The USB Driver Root Hub Operation Enable function must match this signature. The USB Host Stack calls this function when it ready to receive device attach events from the root hub. Calling this function will cause the USB Driver root hub functionality to enable detection of device attach and detach. The USB Driver will then raise events to the USB Host Stack. The `handle` parameter is the driver handle obtained by calling the driver Open function. Setting the `enable` parameter to true enables the root hub operation. Setting the `enable` parameter to false disables the root hub operation.

Driver Host Root Hub Operation Enable Status Function

The `rootHubOperationIsEnabled` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub Operation Enable Status function. The signature of this function is as follows:
```
bool (*rootHubOperationIsEnabled)(DRV_HANDLE handle);
```

The USB Driver Root Hub Operation Enable Status function must match this signature. This USB Host Stack calls this function after calling the operation enable function to check if this has completed. The function returns true if the operation enable function has completed. The USB Host Stack will call this function periodically until it returns true.

Driver Host Root Hub Initialize Function

The `rootHubInitialize` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub Initialize function. The signature of this function is as follows:
```
void (*rootHubInitialize)(DRV_HANDLE handle, USB_HOST_DEVICE_OBJ_HANDLE usbHostDeviceInfo);
```

The USB Driver Root Hub Initialize function must match this signature. The USB Host Stack calls this function to assign a device identifier (usbHostDeviceInfo) to the root hub. This function is called before the Host Stack enables the root hub operation. The USB Driver root hub should use this identifier as the parent identifier when it calls the USB_HOST_DeviceEnumerate function to enumerate the attached device. At the time of enumeration, the USB Host Stack will use this parent identifier to identify the parent hub (whether root hub or external hub) of the attached device. The USB Driver root hub should retain the `usbHostDeviceInfo` parameter for the life time of its operation.

Driver Device Mode Client Functions

Provides information on the USB Driver Device mode Client functions.

Description

The DRV_USB_DEVICE_INTERFACE structure contains pointers to the USB Driver's Device mode Client Functions. These functions are only applicable when the USB module is operating as a USB Device. A USB Driver must implement these functions and ensure that the Device Stack

can access these functions through the driver's DRV_USB_DEVICE_INTERFACE structure. Descriptions of the Driver Device Mode Client functions in the DRV_USB_DEVICE_INTERFACE structure include:

- Driver Device Address Set Function
- Driver Device Current Speed Get Function
- Driver Device SOF Number Get Function
- Driver Device Attach Function
- Driver Device Detach Function
- Driver Device Endpoint Enable Function
- Driver Device Endpoint Disable Function
- Driver Device Endpoint Stall Function
- Driver Device Endpoint Stall Clear Function
- Driver Device Endpoint Enable Status Function
- Driver Device Endpoint Stall Status Function
- Driver Device IRP Submit Function
- Driver Device IRP Cancel All Function
- Driver Device IRP Cancel Function
- Driver Device Remote Wakeup Start Function
- Driver Device Remote Wakeup Stop Function
- Driver Device Test Mode Enter Function

The PIC32MZ and the PIC32MX USB peripheral drivers implement the Device mode functions and export these functions to the Device Stack though their respective DRV_USB_DEVICE_INTERFACE structure.

Driver Device Address Set Function

The `deviceAddressSet` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device Address Set function. The signature of this function is as follows:

```
void (*deviceAddressSet)(DRV_HANDLE handle, uint8_t address);
```

The USB Driver Device Address Set Function should match this signature. The USB Device Stack will call this function to set the Device USB Address. The function will be called in an interrupt context and hence the function implementation must be interrupt-safe. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `address` parameter is the address provided by the USB Host through the Set Device Address Standard request.

Driver Device Current Speed Get Function

The `deviceCurrentSpeedGet` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Current Speed Get function. The signature of this function is as follows:

```
USB_SPEED (*deviceCurrentSpeedGet)(DRV_HANDLE handle);
```

The USB Driver Device Current Speed Get function should match this signature. The USB Device Stack will call this function to obtain the speed at which the device has connected to the USB. It will call this function after reset signaling has completed. The `handle` parameter is driver handle obtained from calling the driver Open function. This function is called in an interrupt context and should be interrupt-safe.

Driver Device SOF Number Get Function

The `deviceSOFNumberGet` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Start-Of-Frame Number Get function. The signature of this function is as follows:

```
uint16_t (*deviceSOFNumberGet)(DRV_HANDLE handle);
```

The USB Driver SOF Number Get function should match this signature. The USB Device Stack will call this function to obtain the current SOF number. The USB peripheral uses a 16 bit counter to count the number of SOFs that have occurred since USB reset. This value is returned along with the Device Stack Start of Frame event. This function is called from an interrupt context and should be interrupt-safe. The `handle` parameter is the driver handle obtained from calling the driver Open function.

Driver Device Attach Function

The `deviceAttach` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Attach function. The signature of this function is as follows:

```
uint16_t(*deviceAttach)(DRV_HANDLE handle);
```

The USB Driver Attach function should match this signature. The USB Device Stack will call this function when the Device application calls the USB Device Stack Device Attach function. The USB Driver will enable the required signaling resistors for indicate attach to the Host. The application could call this function in response to a VBUS power available event. This function must be interrupt-safe. The `handle` parameter is the driver handle obtained from calling the driver Open function.

Driver Device Detach Function

The deviceDetach member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Detach function. The signature of this function is as follows:

uint16_t(*deviceDetach)(DRV_HANDLE handle);

The USB Driver Detach function should match this signature. The USB Device Stack will call this function when the Device application calls the USB Device Stack Device Detach function. The USB Driver will disable the required signaling resistors to indicate detach to the Host. The application could call this function in response to a VBUS power not available event. This function should be interrupt-safe. The `handle` parameter is driver handle obtained from calling the driver Open function.

Driver Device Endpoint Enable Function

The `deviceEndpointEnable` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Endpoint Enable function. The signature of this function is as follows:

```
USB_ERROR (*deviceEndpointEnable)(DRV_HANDLE handle, USB_ENDPOINT endpoint,
            USB_TRANSFER_TYPE transferType, uint16_t endpointSize);
```

The USB Driver Endpoint Enable function should match this signature. The USB Device Stack Function Driver will call this function when it is initialized by the USB Device Layer. The Device Layer, on receiving the Set Configuration request from the Host, identifies the function drivers that are required by the configuration and initializes them. The function drivers will call the endpoint enable function to enable the endpoints required for their operation. Enabling the endpoint will cause it reply to transaction requests from the Host and accept transfer requests from the device application.

The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) that should be enabled. The transferType is the type of the USB transfer that this endpoint will handle. The endpointSize is the size of the maximum transaction that the endpoint will handle. This should match the endpoint size communicated to the Host via the device endpoint descriptors.

The function will return USB_ERRROR_NONE if the endpoint was configured successfully. The function will return USB_ERROR_DEVICE_ENDPOINT_INVALID if the specified endpoint is not provisioned in the system configuration. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid.

The endpoint enable function will be called in an interrupt context and should be interrupt-safe. It is not expected to be thread safe. For standard function drivers, the endpoint enable function will be called in the context of the USB Device Layer Client. For vendor USB devices, the vendor application must call the endpoint enable function in response to and within the context of the device configured event. Again this event itself will execute in the context of the Device Layer.

Driver Device Endpoint Disable Function

The `deviceEndpointDisable` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Endpoint Disable function. The signature of this function is as follows:

```
USB_ERROR (*deviceEndpointDisable)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
```

The USB Driver Endpoint Disable function should match this signature. The USB Device Stack Function Driver will call this function when it is deinitialized by the USB Device Layer. The Device Layer will deinitialize function drivers when it receives a USB reset event from the driver or on receiving the Set Configuration request from the Host with configuration parameter 0. Disabling the endpoint will cause it NAK transaction request from the Host and not accept transfer requests from the device application.

The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) that should be disabled.

The function will return USB_ERRROR_NONE if the function executed successfully. The function will return USB_ERROR_DEVICE_ENDPOINT_INVALID if the specified endpoint is not provisioned in the system configuration. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid.

The endpoint disable function will be called in an interrupt context and should be interrupt-safe. It is not expected to be thread safe. For standard function drivers, the endpoint disable function will be called in the context of the USB Device Layer Client. For vendor USB devices, the vendor application must call the endpoint enable function in response to and within the context of the device reset event. Again this event itself will execute in the context of the Device Layer. Disabling the endpoint will not cancel any transfers that have been queued against the endpoint. The function drivers will call the IRP Cancel All function to cancel any pending transfers.

Driver Device Endpoint Stall Function

The `deviceEndpointStall` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Endpoint Stall function. The signature of this function is as follows:

```
USB_ERROR (*deviceEndpointStall)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
```

The USB Driver Endpoint Stall function should match this signature. The USB Device Stack Function Driver will call this function to stall an endpoint. The Device Layer itself will stall endpoint 0 for several reasons including non-support of the Host request or failure while executing the request. A function driver will also stall an endpoint for protocol specific reasons. The driver will stall both, receive and transmit directions when stalling Endpoint 0. The driver will stall the specified direction while stalling a non-zero endpoint.

This function must be thread safe and interrupt safe. Stalling the endpoint will abort all the transfers queued on the endpoint with the completion status set to USB_DEVICE_IRP_STATUS_ABORTED_ENDPOINT_HALT. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) that should be stalled. The function will return USB_ERRROR_NONE if the function executed successfully. The function will return USB_ERROR_DEVICE_ENDPOINT_INVALID if the specified endpoint is not provisioned in the system configuration. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid.

Driver Device Endpoint Stall Clear Function

The `deviceEndpointStallClear` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Endpoint Stall Clear function. The signature of this function is as follows:

```
USB_ERROR (*deviceEndpointStallClear)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
```

The USB Driver Endpoint Stall Clear function should match this signature. The USB Device Stack Function Driver will call this function to clear the stall on a non-zero endpoint. The Device Layer will call this function to clear the stall condition on Endpoint 0. Clearing the stall on a non-zero endpoint will clear all transfers scheduled on the endpoint and transfer completion status will be set to USB_DEVICE_IRP_STATUS_TERMINATED_BY_HOST. When the stall is cleared, the data toggle for non-zero endpoint will be set to DATA0. The data toggle on Endpoint 0 OUT endpoint will be set to DATA1. The USB Driver will clear the Stall condition on an endpoint even if it was not stalled.

This function must be thread safe and interrupt safe. Stalling the endpoint will flush all the transfers queued on the endpoint. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) whose stall condition must be cleared. The function will return USB_ERRROR_NONE if the function executed successfully. The function will return USB_ERROR_DEVICE_ENDPOINT_INVALID if the specified endpoint is not provisioned in the system configuration. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid.

Driver Device Endpoint Enable Status Function

The `deviceEndpointIsEnabled` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Endpoint Enable Status function. The signature of this function is as follows:

```
bool (*deviceEndpointIsEnabled)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
```

The USB Driver Endpoint Enable Status function should match this signature. The USB Device Stack function will call this function to check if an endpoint has been enabled. The function returns true if the endpoint is enabled. The endpoint is enabled through the USB Driver Endpoint Enable function. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) whose enable status needs to be queried.

Driver Device Endpoint Stall Status Function

The `deviceEndpointIsStalled` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Endpoint Stall Status function. The signature of this function is as follows:

```
bool (*deviceEndpointIsStalled)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
```

The USB Driver Endpoint Stall Status function should match this signature. The USB Device Stack function will call this function to check if an endpoint has been stalled. The function returns true if the endpoint is stalled. The endpoint is stalled through the USB Driver Endpoint Stall function. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) whose stall status needs to be queried.

Driver Device IRP Submit Function

The deviceIRPSubmit member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device IRP Submit function. The signature of the IRP submit function is as follows:

```
USB_ERROR (*deviceIRPSubmit)(DRV_HANDLE handle, USB_ENDPOINT endpoint, USB_DEVICE_IRP * irp);
```

The USB Driver Device IRP Submit function must match this signature. The Device Stack (USB Device calls this function to submit an IRP to the USB Driver. The USB Driver provides this mechanism to transfer data between the device and the Host. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter should set to endpoint through which transfer must be processed. The `irp` parameter should point to the Device IRP data structure. The IRP data structure will transport an entire transfer over the endpoint. The USB Driver will split up the transfer into transactions based on the endpoint size specified at the time of enabling the endpoint. This process does not require Device Stack intervention.

The function will return USB_ERRROR_NONE if the function executed successfully. The function will return USB_ERROR_DEVICE_ENDPOINT_INVALID if the specified endpoint is not provisioned in the system configuration. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid. It will return USB_ERROR_DEVICE_IRP_IN_USE if an in progress IRP is resubmitted. It will return USB_ERROR_ENDPOINT_NOT_CONFIGURED if the IRP is submitted to an endpoint that is not enabled.

The USB Driver will queue the IRP if there is already an IRP being processed on the endpoint. The completion of the IRP processing is indicated by the USB Driver calling the IRP callback function specified within the IRP. The Device IRP Submit function must be thread safe and IRP callback safe. The Device Stack may resubmit the IRP within the IRP callback function. The IRP callback function itself executes within an interrupt context. The completion status of the IRP will be available in the status member of the IRP when the IRP callback function is invoked.

Driver Device IRP Cancel All Function

The `deviceIRPCancelAll` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device IRP Cancel All function. The signature of this is as follows:

```
USB_ERROR (*deviceIRPCancelAll)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
```

The USB Driver Device IRP Cancel All function must match this signature. The USB Device Stack will call this function before disabling the endpoint. Calling this function will call all IRPs that are queued on the endpoint to be canceled. The callback of each IRP will be invoked and the IRP completion status will be set to USB_DEVICE_IRP_STATUS_ABORTED. If an IRP is in progress, an ongoing transaction will be allowed to complete and pending transactions will be canceled. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) whose queued IRPs must be canceled.

The function is thread safe and interrupt safe and will return USB_ERRROR_NONE if it executed successfully. The function will return USB_ERROR_DEVICE_ENDPOINT_INVALID if the specified endpoint is not provisioned in the system configuration. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid.

Driver Device IRP Cancel Function

The `deviceIRPCancel` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device IRP Cancel function. The signature of this is as follows:

```
USB_ERROR (*deviceIRPCancel)(DRV_HANDLE handle, USB_DEVICE_IRP * IRP);
```

The USB Driver Device IRP Cancel function must match this signature. This function is called by the USB Device Stack function driver to cancel a scheduled IRP. If the IRP is in the queue but it's processing has not started, the IRP will removed from the queue and the IRP callback function will be called from within the cancel function. The callback will be invoked with the IRP completion status set to USB_DEVICE_IRP_STATUS_ABORTED. If an IRP is in progress, an ongoing transaction will be allowed to complete and pending transactions will be canceled. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `irp` parameter is the IRP to be canceled.

The function is thread safe and will return USB_ERRROR_NONE if it executed successfully. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid or if the IRP has status indicates that this IRP is not queued or not in progress. The application should not release the data memory associated with IRP unless the callback has been received.

Driver Device Remote Wakeup Start Function

The `deviceRemoteWakeupStart` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device Remote Wakeup Start function. The signature of this function is as follows:

```
void (*deviceRemoteWakeupStart)(DRV_HANDLE handle);
```

The USB Driver Device Remote Wakeup Start function must match this signature. The USB Device Stack will call the function when the device application wants to start remote wakeup signaling. This would happen if the device supports remote wake-up capability and this has been enabled by the Host. The `handle` parameter is the driver handle obtained from calling the driver Open function.

Driver Device Remote Wakeup Stop Function

The `deviceRemoteWakeupStop` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device Remote Wakeup Stop function. The signature of this function is as follows:

```
void (*deviceRemoteWakeupStop)(DRV_HANDLE handle);
```

The USB Driver Device Remote Wakeup Stop function must match this signature. The USB Device Stack will call the function when the device application wants to stop remote wakeup signaling. The application would call after calling the remote wakeup start function. The `handle` parameter is the driver handle obtained from calling the driver Open function.

Driver Device Test Mode Enter Function

The `deviceTestModeEnter` parameter of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device Test Mode Enter function. The signature of this function is as follows:

```
USB_ERROR (*deviceTestModeEnter)(DRV_HANDLE handle, USB_TEST_MODE_SELECTORS testMode);
```

The USB Driver Device Test Mode Enter function should match this signature. The USB Device Stack calls this driver function to place the driver into test mode. This is required when the USB Host (operating at Hi-Speed) send the Set Feature request with the feature selector test set to test mode. This request also specifies which of the test mode signals, the driver should enable. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `testMode` parameter should be set to one of the test modes as defined in table 9-7 of the USB 2.0 specification.

The test mode enter function is only supported by the PIC32MZ USB Driver as the USB peripheral on this controller supports Hi-Speed operation. The function will return USB_ERRROR_NONE if it executed successfully. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid.

This concludes the discussion on the DRV_USB_DEVICE_INTERFACE structure. The following sections describe using the USB Common Driver.

Opening the Driver

Provides information and examples for opening the driver.

Description

The USB Host Stack and the USB Device Stack must obtain a handle to the USB Driver to access the functionality of the driver. This handle is obtained through the USB Driver Open function. The DRV_USB_DEVICE_INTERFACE structure and DRV_USB_DEVICE_HOST_INTERFACE structure provide access to the USB Driver Open function through the `open` member of these structures. Calling the Open function may not return a valid driver handle the first time the function is called. In fact, the USB Driver will return an invalid driver handle until the driver is ready to be opened. The Host and the Device Stack call the Open function repetitively in a state machine, until the function returns a valid handle.

The USB Host Stack can open the USB Driver but can call its Host mode functions only if the USB Driver was initialized for Host mode or Dual Role operation. The USB Host Stack accesses the driver functions through the DRV_USB_HOST_INTERFACE pointer that was provided to the Host Layer through the Host Stack initialization. The USB Device Stack can open the USB Driver but can call its Device mode functions only if the USB Driver was initialized for Device mode or Dual Role operation. The USB Device Stack accesses the driver functions through the DRV_USB_HOST_INTERFACE pointer that was provided to the Host Layer through the Host Stack initialization

The following code example shows how the USB Host Layer opens the USB Driver.

```
/* This code example shows how the Host Layer open the HCD via the hcdInterface.
 * The driver handle is stored in hcdHandle member of the busObj data structure.
```

```
 * The busObj data structure Host Layer local data structure. The Host Layer
 * opens the HCD when the bus is enabled. This operation takes place in the
 * USB_HOST_BUS_STATE_ENABLING state. */

/* Note the Host Layer calls the Open function by accessing the open member of
 * the hcdInterface which is of the type DRV_USB_HOST_INTERFACE. Also note how
 * the function is called repetitively until the Open function returns a valid
 * handle. */

case USB_HOST_BUS_STATE_ENABLING:

    /* The bus is being enabled. Try opening the HCD */
    busObj->hcdHandle = busObj->hcdInterface->open(busObj->hcdIndex, DRV_IO_INTENT_EXCLUSIVE |
            DRV_IO_INTENT_NONBLOCKING | DRV_IO_INTENT_READWRITE );

    /* Validate the Open function status */
    if (DRV_HANDLE_INVALID == busObj->hcdHandle )
    {
        /* The driver may not open the first time. This is okay. We
         * should try opening it again. The state of bus is not
         * changed. */
    }
```

The following code example shows how the USB Device Layer opens the USB Driver.

```
/* This code example shows how the USB Device Layer calls the USBCD open
 * function to open the USBCD. The Device Layer accesses the USBCD Open function
 * through the driverInterface member of the usbDeviceInstanceState object. The
 * driverInterface member is a DRV_USB_DEVICE_INTERFACE type. The
 * usbDeviceInstanceState is a USB Device Layer local object. */

/* The Device Layer attempts to open the USBCD when it is initializing. Note how
 * the Device Layer advances to the next state only when the USBCD returns a
 * valid handle.   */

switch(usbDeviceThisInstance->taskState)
{
    case USB_DEVICE_TASK_STATE_OPENING_USBCD:

        /* Try to open the driver handle. This could fail if the driver is
         * not ready to be opened. */
        usbDeviceThisInstance->usbCDHandle =
           usbDeviceThisInstance->driverInterface->open( usbDeviceThisInstance->driverIndex,
           DRV_IO_INTENT_EXCLUSIVE|DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE);

        /* Check if the driver was opened */
        if(usbDeviceThisInstance->usbCDHandle != DRV_HANDLE_INVALID)
        {
            /* Yes the driver could be opened. */

            /* Advance the state to the next state */
            usbDeviceThisInstance->taskState = USB_DEVICE_TASK_STATE_RUNNING;

            /* Update the USB Device Layer state to indicate that it can be
             * opened */
            usbDeviceThisInstance->usbDeviceInstanceState = SYS_STATUS_READY;
        }

        break;
```

USB Driver Host Mode Operation

Provides information on Host mode operation.

Description

The USB Driver operates or can operate in the Host mode when it is initialized for Host mode or Dual Role operation. When operating in Host mode, the USB Driver is also referred to as the Host Controller Driver (HCD). In Dual Role mode, the USB Driver will switch to Host mode when the USB Driver Host Root Hub Operation Enable function is called.

The USB Driver Client must perform these steps to operate the USB Driver in Host mode.

1. Open the USB Driver to obtain the driver handle.
2. Set the event handler.
3. Call the Root Hub Control function to obtain the speed of the root hub, the number of ports that the root hub supports, and the maximum current that the root hub VBUS can supply.
4. Calls the Root Hub Initialize function with an identifier parameter. This `identifier` parameter allows the Host Stack to uniquely identify the root hub when where there are multiple root hubs.
5. The Driver Client will then enable the root hub operation and will wait until the root hub operation is enabled.
6. The Driver Client can now call the USB Driver Host mode functions.

The following sections explain Steps 2 through 6 in more detail.

Handling Host Mode Driver Events

Currently, the HCD does not provide any events to the client. The client can optionally register an event handler through the eventHandlerSet function pointer in the DRV_USB_HOST_INTERFACE structure. Future releases of the USB Driver may contain features that provide events to the Driver Client. Please refer to the following **Root Hub Operation** section for details on how the driver indicates device attach and detach to the client.

Root Hub Operation

A key feature of the HCD is the Root Hub Driver. The Root Hub Driver emulates hub operation in USB Driver software and provides a hub like interface to the USB Host Layer. The USB Host Layer treats the root hub like an external hub. This simplifies the implementation of USB Host Layer while supporting multiple devices through a hub. In that, the USB Host layer does not have to treat a device connected directly to the USB peripheral differently than a device connected to an external hub. The following code example shows how the USB Host Layer calls the root hub function to obtain information about the root hub.

```
/* This code example shows how the USB Host Layer calls the root hub functions to
 * obtain information about the root. The USB Host Layer first opens the HCD and
 * then accesses the root hub functions through the rootHubInterface member of
 * hcdInterface. rootHubInterface is of the type DRV_USB_ROOT_HUB_INTERFACE and
 * the hcdInterface is of the type of DRV_USB_HOST_INTERFACE. */

/* The code example shows how the Host Layer gets to know the root hub operation
 * speed, number of root hub ports and the maximum amount of current that the
 * root can supply. These function can be called only after HCD was opened and a
 * valid driver handle obtained. */

case USB_HOST_BUS_STATE_ENABLING:

    /* The bus is being enabled. Try opening the HCD */
    busObj->hcdHandle = busObj->hcdInterface->open(busObj->hcdIndex, DRV_IO_INTENT_EXCLUSIVE |
            DRV_IO_INTENT_NONBLOCKING | DRV_IO_INTENT_READWRITE );

    /* Validate the Open function status */
    if (DRV_HANDLE_INVALID == busObj->hcdHandle )
    {
        /* The driver may not open the first time. This is okay. We
         * should try opening it again. The state of bus is not
         * changed. */
    }
    else
    {
        /* Update the bus root hub information with the
         * details of the controller. Get the bus speed, number of
         * ports, the maximum current that the HCD can supply,
         * pointer to the root hub port functions. */

        SYS_DEBUG_PRINT(SYS_ERROR_INFO,
             "\r\nUSB Host Layer: Bus %d Root Hub Driver Opened.",hcCount);

        busObj->rootHubInfo.speed =
             busObj->hcdInterface->rootHubInterface.rootHubSpeedGet(busObj->hcdHandle);

        busObj->rootHubInfo.ports =
             busObj->hcdInterface->rootHubInterface.rootHubPortNumbersGet(busObj->hcdHandle);

        busObj->rootHubInfo.power =
             busObj->hcdInterface->rootHubInterface.rootHubMaxCurrentGet(busObj->hcdHandle);

        busObj->rootHubInfo.rootHubPortInterface =
             busObj->hcdInterface->rootHubInterface.rootHubPortInterface;
```

The USB Host Layer must initialize and enable the operation of the root hub. While initializing the Root Hub Driver, the Host layer will assign a unique identifier to the root hub. The root hub will return this value as the parent identifier while calling the USB_HOST_DeviceEnumerate function. The USB Host Layer must then enable the operation of the root hub driver. This will cause the root hub driver to detect device attach and detach. The following code example shows how the USB Host Layer initializes and enables the root hub driver

```
/* The following code example show how the USB Host Layer initializes the root
 * hub and then enables the root hub operation. The
 * rootHubDevice->deviceIdentifier is a unique identifier that allows the USB
 * Host layer to identify this root hub. It is returned by the root hub driver
 * in the USB_HOST_DeviceEnumerate() function as the parent identifier when the
 * device is connected to the root hub. */

/* The hcdHandle is the driver handle. The hcdInterface pointer is of the type
 * DRV_USB_HOST_INTERFACE and points to the HCD interface. */

busObj->hcdInterface->rootHubInterface.rootHubInitialize( busObj->hcdHandle ,
                                                          rootHubDevice->deviceIdentifier );
busObj->hcdInterface->rootHubInterface.rootHubOperationEnable( busObj->hcdHandle , true );
```

When a device is attached, the Root Hub Driver will implement the required settling attach settling delay and will then call the USB Host Layer's USB_HOST_DeviceEnumerate function to enumerate the device. While calling this function, the root hub driver will provide the identifier that was provided to it in its initialize function. The USB_HOST_DeviceEnumerate function will return an identifier which uniquely identifies the attached device. The root hub driver uses this value to identify the device to the Host when the USB_HOST_DeviceDenumerate function is called on device detach. The following code example shows how the Root Hub driver calls the USB_HOST_DeviceEnumerate and the USB_HOST_DeviceDenumerate functions.

```
/* The following code shows how the root hub driver calls the
 * USB_HOST_DeviceEnumerate() function in the device attach interrupt. As seen
 * here, the root hub returns the identifier that the USB Host Layer assigned to
 * it the rootHubInitialize function call. The pUSBDrvObj->usbHostDeviceInfo
 * variable contains this identifier. */

if(PLIB_USB_InterruptFlagGet(usbID, USB_INT_ATTACH))
{
    /* We can treat this as a valid attach. We then clear the
     * detach flag and enable the detach interrupt. We enable
     * the Transaction interrupt */

    PLIB_USB_InterruptFlagClear(usbID, USB_INT_HOST_DETACH);
    PLIB_USB_InterruptEnable(usbID, USB_INT_HOST_DETACH);
    PLIB_USB_InterruptEnable(usbID, USB_INT_TOKEN_DONE);

    /* Ask the Host layer to enumerate this device. While calling
     * this function, the UHD of the parent device which is the
     * root hub in this case.
     * */
    pUSBDrvObj->attachedDeviceObjHandle = USB_HOST_DeviceEnumerate
                                          (pUSBDrvObj->usbHostDeviceInfo, 0);
}

/* The following code example shows how the root hub driver calls the
 * USB_HOST_DeviceDenumerate() function in the device detach interrupt. Note how
 * the attachedDeviceObjHandle that was assigned at the time of device
 * enumeration is returned to the Host Layer to let the Host know which device
 * is being detached. */

if((usbInterrupts & USB_INT_HOST_DETACH) && (enabledUSBInterrupts & USB_INT_HOST_DETACH))
{
    /* Perform other detach related handling */

    /* Ask the Host Layer to de-enumerate this device. */
    USB_HOST_DeviceDenumerate (pUSBDrvObj->attachedDeviceObjHandle);

    /* Disable the LS Direct Connect. It may have been enabled if the last
     attach was for a Low-Speed device. */
    PLIB_USB_EP0LSDirectConnectDisable(pUSBDrvObj->usbID);

    /* Continue to perform detach handling */
}
```

Root Hub Port Operation

The HCD Root Hub Driver exposes a set of port related functions that allow the USB Host Layer to control the port. The most commonly used functions are the function to reset the port and get the port speed. In this case, this is the speed of the attached device. The following code example shows how the USB Host Layer calls the hubPortReset, hubPortResetIsComplete and hubPortSpeedGet port functions.

```
/* The following code shows an example of how the Host Layer called the
 * hubPortReset function to reset the port to which the device is connected.
 * The code proceeds with the port reset if no device on the bus is in an
 * enumeration state. It will then call the hubPortReset function of the parent
 * hub of the device. The parent hub, hubInterface member of deviceObj points to
 * this driver, can be the root hub or an external hub */

if(!busObj->deviceIsEnumerating)
{
    /* Remember which device is enumerating */
    busObj->enumeratingDeviceIdentifier = deviceObj->deviceIdentifier;

    /* Grab the flag */
    busObj->deviceIsEnumerating = true;

    /* Reset the device */
    deviceObj->hubInterface->hubPortReset( deviceObj->hubHandle, deviceObj->devicePort );
}

/* The following code example shows how the Host checks if the port reset
 * operation has completed. If the reset operation has completed, the speed of
 * the attached device can be obtained. The reset settling delay can then be
 * started. */

 case USB_HOST_DEVICE_STATE_WAITING_FOR_RESET_COMPLETE:

    /* Check if the reset has completed */
    if(deviceObj->hubInterface->hubPortResetIsComplete
                ( deviceObj->hubHandle ,deviceObj->devicePort ))
    {
        /* The reset has completed. We can also obtain the speed of the
         * device. We give a reset recovery delay to the device */

        deviceObj->speed = deviceObj->hubInterface->hubPortSpeedGet
                           (deviceObj->hubHandle, deviceObj->devicePort);

        deviceObj->deviceState = USB_HOST_DEVICE_STATE_START_RESET_SETTLING_DELAY;
    }
```

Opening and Closing a Pipe

The HCD client can open a pipe to the device after resetting the device. The USB Host Layer calls the hostPipeSetup function in the DRV_USB_HOST_INTERFACE structure to open a pipe. The USB Host Layer must open a pipe to communicate to a specific endpoint on a target device. While opening the pipe, the USB Host Layer must specify parameters which specify the address of the target device, the type of the transfer that the pipe must support and the speed of the pipe. If the device is connected to a hub, the address of the hub must be specified. The HCD Pipe Setup function *is not* interrupt-safe. It should not be called in any event handler that executes in an interrupt context.

The Pipe Setup function returns a valid pipe handle if the pipe was opened successfully. Pipe creation may fail if the target device was disconnected or if there are insufficient resources to open the pipe. The pipe handle is then used along with the hostIRPSubmit function to transfer data between the Host and the device. The following code shows example usage of a Pipe Open function.

```
/* The following code example shows how the Host Layer uses the hostPipeSetup
 * function to open a control pipe to the attached device. Most of the
 * parameters that are passed to this function become known when the device is
 * attached. The pipe handle is checked for validity after the hostPipeSetup
 * function call. */

if(busObj->timerExpired)
{
    busObj->busOperationsTimerHandle = SYS_TMR_HANDLE_INVALID;
    /* Settling delay has completed. Now we can open default address
     * pipe and and get the configuration descriptor */

    SYS_DEBUG_PRINT(SYS_ERROR_INFO,
                    "\r\nUSB Host Layer: Bus %d Device Reset Complete.", busIndex);
```

```
    deviceObj->controlPipeHandle =
              deviceObj->hcdInterface->hostPipeSetup( deviceObj->hcdHandle,
              USB_HOST_DEFAULT_ADDRESS , 0 /* Endpoint */,
              deviceObj->hubAddress /* Address of the hub */,
              deviceObj->devicePort /* Address of the port */,
              USB_TRANSFER_TYPE_CONTROL, /* Type of pipe to open */
              0 /* bInterval */, 8 /* Endpoint Size */, deviceObj->speed );

    if(DRV_USB_HOST_PIPE_HANDLE_INVALID == deviceObj->controlPipeHandle)
    {
        /* We need a pipe else we cannot proceed */
        SYS_DEBUG_PRINT(SYS_ERROR_DEBUG,
        "\r\nUSB Host Layer: Bus %d Could not open control pipe. Device not supported.", busIndex);
    }
}
```

An open pipe consumes computational and memory resources and must therefore must be closed if it will not be used. This is especially true of pipes to a device that is detached. The Host Layer calls the hostPipeClose function in the DRV_USB_HOST_INTERFACE structure to close the pipe. The pipe to be closed is specified by the pipe handle. The Pipe Close function can be called from an event handler. It is interrupt safe. Closing a pipe will cancel all pending transfers on that pipe. The IRP callback for such canceled transfers will be called with the status USB_HOST_IRP_STATUS_ABORTED. The following code example shows an example of closing the pipe.

```
/* The following code example shows an example of how the Host Layer calls the
 * hostPipeClose function to close an open pipe. Pipe should be closed if it
 * will not used. An open pipe consumes memory resources. In this example, the
 * Host Layer closes the pipe if it was not able successfully submit an IRP to
 * this pipe. */

/* Submit the IRP */
if(USB_ERROR_NONE != deviceObj->hcdInterface->hostIRPSubmit
                  ( deviceObj->controlPipeHandle, & (deviceObj->controlTransferObj.controlIRP)))
{
    /* We need to be able to send the IRP. We move the device to
     * an error state. Close the pipe and send an event to the
     * application. The assigned address will be released when
     * the device in unplugged. */

    SYS_DEBUG_PRINT(SYS_ERROR_DEBUG,
        "\r\nUSB Host Layer: Bus %d Set Address IRP failed. Device not supported.", busIndex);

    /* Move the device to error state */
    deviceObj->deviceState = USB_HOST_DEVICE_STATE_ERROR;

    /* Close the pipe as we are about mark this device as unsupported. */
    deviceObj->hcdInterface->hostPipeClose(deviceObj->controlPipeHandle);
}
```

Transferring Data to an Attached Device

The USB Host Layer, the HCD client, needs to transfer data to the attached device to understand the device capabilities and to operate the device. The HCD uses a concept of Input Output Request Packet (IRP) to transfer data to and from the attached device. IRPs are transported over pipes which are setup by calling the USB Driver Pipe Setup function.

A Host IRP is a USB_HOST_IRP type data structure. The IRP is created by the Host layer and submitted to the HCD for processing through the hostIRPSubmit function. At the time of submitting the IRP, the pipe over which the IRP must be transported is specified. The data request in the IRP is transported using the attributes of pipe. When an IRP is submitted to the HCD, it is owned by the HCD and cannot be modified by the Host Layer until the HCD issues an IRP callback. The HCD will issue the IRP callback when it has completed or terminated processing of the IRP.

An IRP does not have its own transfer type. It inherits the properties of the pipe to which it is submitted. Hence an IRP becomes a control transfer IRP it was submitted to a control transfer pipe. A pipe allows multiple IRPs to be queued. This allows the Host Layer to submit IRPs to a pipe even while an IRP is being processed on the pipe. The HCD will process an IRP in the order that it was received. The following code example shows the USB_HOST_IRP data structure.

```
/* The following code example shows the USB_HOST_IRP structure. The Host Layer
 * uses this structure to place data transfer requests on a pipe. */

typedef struct _USB_HOST_IRP
{
    /* Points to the 8 byte setup command packet in case this is a IRP is
     * scheduled on a CONTROL pipe. Should be NULL otherwise */
    void * setup;

    /* Pointer to data buffer */
    void * data;
```

```
    /* Size of the data buffer */
    unsigned int size;

    /* Status of the IRP */
    USB_HOST_IRP_STATUS status;

    /* Request specific flags */
    USB_HOST_IRP_FLAG flags;

    /* User data */
    uintptr_t userData;

    /* Pointer to function to be called when IRP is terminated. Can be NULL, in
     * which case the function will not be called. */
    void (*callback)(struct _USB_HOST_IRP * irp);

    /*************************************
     * These members of the IRP should not be
     * modified by client
     *************************************/
   uintptr_t privateData[7];

} USB_HOST_IRP;
```

The `setup` member of the USB_HOST_IRP structure must point to the 8 byte setup packet for control transfers. The driver will send this 8 byte data in the Setup phase of the control transfer. It can be NULL for non-control transfers. This member is only considered if the IRP is submitted to a control transfer pipe. It is ignored for non-control transfer pipes. The structure of the setup command should match that specified in the USB 2.0 specification.

The `data` member of the USB_HOST_IRP structure points to a data buffer. This data buffer will contain the data that needs to be sent to the device for data stage of a OUT transfer, or it will contain the data that was received from the device during an IN transfer. Any hardware specific cache coherency and address alignment requirements must be considered while allocating this data buffer. The Driver Client should not modify or examine the contents of the IRP after the IRP has been submitted and is being processed. It can be examined after the driver has released the IRP.

The `size` member of the USB_HOST_IRP structure contains the size of the transfer. for Bulk transfers, the size of the transfer can exceed the size of the transaction (which is equal to size of the endpoint reported by the device). The HCD in such a case will split up the transfer into transactions. This process does not require external intervention. For control transfers, the size of the transfer is specified in the setup packet (pointed to by the `setup` member of the USB_HOST_IRP structure). The driver will itself process the Setup, Data (if required) and Handshake stages of control transfer. This process again does not require external intervention. For interrupt and isochronous transfers, the size of transfer specified in the IRP cannot exceed the size of the transaction. If size is specified as 0, then the driver will send a zero length packet. The `size` parameter of the IRP is updated by the driver when IRP processing is completed. This will contain the size of the completed transfer.

The `status` member of the IRP provides the completion status of the IRP and should be checked only when the IRP processing has completed. This is indicated by the driver calling the IRP callback function. The IRP status is a USB_HOST_IRP_STATUS type. The following code example shows the different possible values of the `status` member and an example of submit a control transfer IRP.

```
/* The following code shows an example of how the Host Layer populates
 * the IRP object and then submits it. IRP_Callback function is called when an
 * IRP has completed processing. The status of the IRP at completion can be
 * checked in the status flag. The size field of the irp will contain the amount
 * of data transferred. */

void IRP_Callback(USB_HOST_IRP * irp)
{
    /* irp is pointing to the IRP for which the callback has occurred. In most
     * cases this function will execute in an interrupt context. The application
     * should not perform any hardware access or interrupt unsafe operations in
     * this function. */

    switch(irp->status)
    {
        case USB_HOST_IRP_STATUS_ERROR_UNKNOWN:
            /* IRP was terminated due to an unknown error */
            break;

        case USB_HOST_IRP_STATUS_ABORTED:
            /* IRP was terminated by the application */
            break;

        case USB_HOST_IRP_STATUS_ERROR_BUS:
            /* IRP was terminated due to a bus error */
```

```
            break;

        case USB_HOST_IRP_STATUS_ERROR_DATA:
            /* IRP was terminated due to data error */
            break;

        case USB_HOST_IRP_STATUS_ERROR_NAK_TIMEOUT:
            /* IRP was terminated because of a NAK timeout */
            break;

        case USB_HOST_IRP_STATUS_ERROR_STALL:
            /* IRP was terminated because of a device sent a STALL */
            break;

        case USB_HOST_IRP_STATUS_COMPLETED:
            /* IRP has been completed */
            break;

        case USB_HOST_IRP_STATUS_COMPLETED_SHORT:
            /* IRP has been completed but the amount of data processed was less
             * than requested. */
            break;

        default:
            break;
    }
}

/* In the following code example the a control transfer IRP is submitted to a
 * control pipe. The setup parameter of the IRP points to the Setup command of
 * the control transfer. The direction of the data stage is specified by the
 * Setup packet. */

USB_HOST_IRP irp;
USB_ERROR result;
USB_HOST_PIPE_HANDLE controlPipe;
USB_SETUP_PACKET setup;
uint8_t controlTransferData[32];

irp.setup = setup;
irp.data = controlTransferData;
irp.size = 32;
irp.flags = USB_HOST_IRP_FLAG_NONE ;
irp.userData = &someApplicationObject;
irp.callback = IRP_Callback;

result = DRV_USBFS_HOST_IRPSubmit(controlPipeHandle, &irp);

switch(result)
{
    case USB_ERROR_NONE:
        /* The IRP was submitted successfully */
        break;

    case USB_ERROR_HOST_PIPE_INVALID:
        /* The specified pipe handle is not valid */
        break;

    case USB_ERROR_OSAL_FUNCTION:
        /* An error occurred while trying to grab mutex */
        break;

    default:
        break;
}
```

The `flags` member of the USB_HOST_IRP structure specifies flags which affect the behavior of the IRP. The USB_HOST_IRP_FLAG enumeration specifies the available option. The USB_HOST_IRP_FLAG_SEND_ZLP causes the driver to add a Zero Length Packet (ZLP) to the data stage of the transfer when the transfer size is an exact multiple of the endpoint size. The USB_HOST_IRP_WAIT_FOR_ZLP flag will cause the driver to wait for a ZLP from the device in a case where the size of data received thus far in the transfer is an exact multiple of the endpoint

size.

The `callback` member of the USB_HOST_IRP structure points to a function which the driver calls when the IRP processing is completed. The Driver Client must implement this function and assign the pointer to this function to the `callback` member of the IRP. Every IRP can have its own callback function or one common callback function could be used. The callback function will execute in an interrupt context. The Driver Client should not execute interrupt unsafe, blocking, or computationally intensive operations in the callback function. The client can call hostIRPSubmit function in the IRP callback function to submit another IRP or resubmit the same IRP. The client can check the status and size of the IRP in the callback function.

The `userData` member of the USB_HOST_IRP structure can be used by the client to associate a client specific context with the Host. This context can then be used by the client, in the IRP callback function to identify the context in which the IRP was submitted. This member is particularly useful if the client wants to implement one callback function for all IRPs.

The `privateData` member of the IRP is used by the driver and should not be accessed or manipulated by the Driver Client. The following code examples show usage of IRPs to transfer data between the Host and the attached device and along with the different flags.

```
/* The following code shows an example of submitting an IRP to send data
 * to a device. In this example we will request the driver to send a ZLP after
 * sending the last transaction. The driver will send the ZLP only if the size
 * of the transfer is a multiple of the endpoint size. This is not a control
 * transfer IRP. So the setup field of the IRP will be ignored.  */

USB_HOST_IRP irp;
USB_ERROR result;
USB_HOST_PIPE_HANDLE bulkOUTPipeHandle;
uint8_t data[128];

irp.data = data;
irp.size = 128;
irp.flags = USB_HOST_IRP_FLAG_SEND_ZLP ;
irp.userData = &someApplicationObject;
irp.callback = IRP_Callback;

result = DRV_USBFS_HOST_IPRSubmit( bulkOUTPipeHandle, &irp );


/* The following code shows an example of submitting an IRP to receive
 * data to a device. In this example we will request the driver to wait for a
 * ZLP after receiving the last transaction. The driver will wait for the ZLP
 * only if the size of the transfer is a multiple of the endpoint size. This is
 * not a control transfer IRP. So the setup field of the IRP will be ignored.
 * */

USB_HOST_IRP irp;
USB_ERROR result;
USB_HOST_PIPE_HANDLE bulkINPipeHandle;
uint8_t data[128];

irp.data = data;
irp.size = 128;
irp.flags = USB_HOST_IRP_FLAG_WAIT_FOR_ZLP ;
irp.userData = &someApplicationObject;
irp.callback = IRP_Callback;

result = DRV_USBFS_HOST_IPRSubmit( bulkINPipeHandle, &irp );
```

USB Driver Device Mode Operation

Provides information on Device mode operation.

Description

The USB Driver operates can operate in the Device mode when it is initialized for Device mode or Dual Role operation. When operating in Device mode, the USB Driver is also referred to as the USB Controller Driver (USBCD). In Dual-Role mode, the USB Driver will switch to USBCD mode when the USB Driver Device Attach function is called.

The USB Driver Client must perform these steps to operate the USB Driver in Device mode.

1. Open the USB Driver to obtain the driver handle.
2. Set the event handler.
3. Wait for the application to attach the device to the bus.
4. Enable Endpoint 0 and respond to USB Host Enumeration requests.
5. Allow the application and function drivers to enable other endpoints and communicate with the Host.

The following sections discuss these operations in more detail.

General Device Mode Operations

Provides information on general Device mode operations.

Description

This section describes the USBCD operations such as setting event handlers and attaching and detaching the device.

Handling Device Mode Driver Events

The Device Layer will call the USBCD eventHandlerSet function to register the Device mode event handling function. The USBCD generates various events that indicate different states of the USB. These events are defined by the DRV_USB_EVENT enumeration. The following code example shows how the Device Layer registers the driver event handling function.

```
/* This code example shows the implementation of the USB_DEVICE_Attach and the
 * USB_DEVICE_Detach function. These functions are actually macro that map
 * directly deviceAttach and the deviceDetach function of the driverInterface
 * member of the deviceClient Object (which is the macro parameter) */

#define USB_DEVICE_Attach( x )  ((USB_DEVICE_OBJ *)x)->driverInterface->deviceAttach
                                ( ((USB_DEVICE_OBJ *)(x))->usbCDHandle)
#define USB_DEVICE_Detach( x )  ((USB_DEVICE_OBJ *)x)->driverInterface->deviceDetach
                                ( ((USB_DEVICE_OBJ *)x)->usbCDHandle)
```

If the driver is operating in interrupt mode, the client event handling function will execute in an interrupt context. The client should not call interrupt unsafe, computationally intensive or blocking functions in the event handler. The following code shows a small example of the Device Layer USBCD Event Handler:

```
/* This code example shows a partial implementation of the USB Device Layer
 * event handler. Note how the code type casts the referenceHandle parameter to
 * a USB_DEVICE_OBJ type. This referenceHandle is the same value that the Device
 * Layer passed when the event handler was set. This now easily allows one
 * implementation of the event handling code to be used by multiple Device
 * Layer instances. */

    void _USB_DEVICE_EventHandler
(
    uintptr_t referenceHandle,
    DRV_USB_EVENT eventType,
    void * eventData
)
{
    USB_DEVICE_OBJ* usbDeviceThisInstance;
    USB_DEVICE_MASTER_DESCRIPTOR * ptrMasterDescTable;
    USB_DEVICE_EVENT_DATA_SOF SOFFrameNumber;

    usbDeviceThisInstance = (USB_DEVICE_OBJ *)referenceHandle;

    /* Handle events, only if this instance is in initialized state */
    if( usbDeviceThisInstance->usbDeviceInstanceState <= SYS_STATUS_UNINITIALIZED )
    {
        /* The device should anyway not be attached when the Device Layer is
         * not initialized. If we receive driver event when the Device Layer is
         * not initialized, there is nothing we can do but ignore them. */
        return;
    }

    switch(eventType)
    {
        case DRV_USB_EVENT_RESET_DETECT:

            /* Clear the suspended state */
            usbDeviceThisInstance->usbDeviceStatusStruct.isSuspended = false;

            /* Cancel any IRP already submitted in the RX direction. */
            DRV_USB_DEVICE_IRPCancelAll( usbDeviceThisInstance->usbCDHandle,
                    controlEndpointRx );

            /* Code not shown for the sake of brevity. */
```

```
    }
}
```

In the previous code example, the Device Layer (the Driver Client) sets the `hReferenceData` parameter, of the Event Handler Set function, to point to a local object. This pointer is returned to the Device Layer, in the event handler when an event occurs. For multiple instances of USB drivers in one application, this allows the Device Layer to easily associate a Device Layer specific context to the driver instance, thus simplifying implementation of the event handler.

Attaching and Detaching the Device

The USB Device Layer calls the USBCD deviceAttach and deviceDetach functions to attach and detach the device on the USB. The USB Device Layer should be ready to handle events which would occur when the device is attached on the bus. Hence the USB Device Layer should register the USBCD event handler before the attach function is called. The deviceAttach and deviceDetach functions can be called in an interrupt context. These functions are respectively called when the USB Device application detects a valid VBUS voltage and when the VBUS voltage is not valid.

Setting the Device Address

The USB Device Layer will call the USBCD deviceAddressSet function to set the USB address of the device. The Device Layer will do this when it receives the Set Address control request from the Host. The USBCD will reset the device address to '0' when it has received reset signaling from the root hub. The following code example shows how the USB Device Layer calls this function.

```
/* The following code example shows how the USB Device Layer calls the
 * DRV_USB_DEVICE_AddressSet function to set the address. The
 * DRV_USB_DEVICE_AddressSet function is actually a macro that calls the
 * deviceAddressSet function of the driverInterface of usbDeviceThisInstance
 * object. The usbDeviceThisInstance is Device Layer object.
 *
 * As seen in this code, the Device Layer calls the address set function when
 * the it a pending set address control request from the Host has completed. */


void _USB_DEVICE_Ep0TransmitCompleteCallback(USB_DEVICE_IRP * handle)
{
    USB_DEVICE_IRP * irpHandle = (USB_DEVICE_IRP *)handle;
    USB_DEVICE_OBJ * usbDeviceThisInstance;
    USB_DEVICE_CONTROL_TRANSFER_STRUCT * controlTransfer;

    usbDeviceThisInstance = (USB_DEVICE_OBJ *)irpHandle->userData;
    controlTransfer = &(usbDeviceThisInstance->controlTransfer);

    if(irpHandle->status == USB_DEVICE_IRP_STATUS_ABORTED)
    {
        return;
    }

    if(usbDeviceThisInstance->usbDeviceStatusStruct.setAddressPending)
    {
        DRV_USB_DEVICE_AddressSet(usbDeviceThisInstance->usbCDHandle,
                                  usbDeviceThisInstance->deviceAddress);
        usbDeviceThisInstance->usbDeviceStatusStruct.setAddressPending = false;
    }

    /* Code not shown for the sake of brevity */

}
```

Device Current Speed and SOF Number

The USB Device Layer will call the USBCD deviceCurrentSpeedGet function to know the speed at which the device is attached to the USB. This allows the Device Layer to select the correct endpoint settings at the time of processing the Set Configuration request issued by the Host. The USB Device Layer will call the deviceSOFNumberGet function to return the SOF number at the time of the SOF event.

Device Remote Wake-up

The USB Device Layer will call the USBCD deviceRemoteWakeupStop and deviceRemoteWakeupStart functions to stop and start remote signaling. The Device layer application will call the USB Device Layer Stop and Start Remote Wakeup Signaling functions to remotely let the root hub know that the device is ready to be woken up. The timing of the remote signaling is controlled by the Device Layer. The client should call the remote wakeup function only when the device is suspended by the Host.

Device Endpoint Operations

Provides information on Device Endpoint operations.

Description

The UBSCD Endpoint functions allow the Driver Client to enable, disable, stall and clear the stall condition on an endpoint. The client submits requests to transmit and receive data from the USB Host on an endpoint.

Endpoint Enable and Disable functions

The USBCD client must enable an endpoint it must use the endpoint for communicating with the USB Host. The client will call the USBCD deviceEndpointEnable function to enable the endpoint. While calling this function, the client must specify the endpoint address, the transfer type to be processed on this endpoint and the maximum size of a transaction on this endpoint. This function is thread-safe when called in an RTOS application. The USBCD allows an endpoint to be accessed by one thread only. The USB Device Layer and the device function drivers will enable the endpoint when the Host sets the device configuration. The USBCD deviceEndpointIsEnabled function is available to check if an endpoint is enabled. The following code example shows how the USB Device Layer enables the device endpoint.

```
/* The following code example shows the USB Device Layer enables Endpoint 0 to
 * prepare for the enumeration process after it has received reset signaling
 * from the Host. The Device Layer calls the deviceEndpointEnable function to
 * to enable the endpoint. The driverInterface member of the
 * usbDeviceThisInstance structure points to the USB Device Mode Driver Common
 * Interface. */

void _USB_DEVICE_EventHandler
(
    uintptr_t referenceHandle,
    DRV_USB_EVENT eventType,
    void * eventData
)
{
    /* Code not shown due to space constraints */

    switch(eventType)
    {
        case DRV_USB_EVENT_RESET_DETECT:

            /* Clear the suspended state */
            usbDeviceThisInstance->usbDeviceStatusStruct.isSuspended = false;

            /* Cancel any IRP already submitted in the RX direction. */
            DRV_USB_DEVICE_IRPCancelAll( usbDeviceThisInstance->usbCDHandle,
                    controlEndpointRx );

            /* Cancel any IRP already submitted in the TX direction. */
            DRV_USB_DEVICE_IRPCancelAll( usbDeviceThisInstance->usbCDHandle,
                    controlEndpointTx );

            /* Deinitialize all function drivers.*/
            _USB_DEVICE_DeInitializeAllFunctionDrivers ( usbDeviceThisInstance );

            /* Disable all endpoints except for EP0.*/
            DRV_USB_DEVICE_EndpointDisableAll(usbDeviceThisInstance->usbCDHandle);

            /* Enable EP0 endpoint in RX direction */
            (void)usbDeviceThisInstance->driverInterface->deviceEndpointEnable
                  (usbDeviceThisInstance->usbCDHandle,
                    controlEndpointTx, USB_TRANSFER_TYPE_CONTROL, USB_DEVICE_EP0_BUFFER_SIZE);

            /* Code not shown due to space constraints */

            break;
    }
}
```

The USB Device Layer and the Function drivers will disable an endpoint when the Host sets a zero-device configuration or when the Host resets the device. The USBCD deviceEndpointDisable function disables an endpoint. When an endpoint is disabled, it does not accept requests for Host communication. Disabling an endpoint does not cancel any communication requests that that have been submitted on the endpoint. These requests must be canceled explicitly.

Device Endpoint Stall and Stall Clear

The USBCD client can call the deviceEndpointStall and deviceEndpointStallClear functions to stall and cleat the stall on an endpoint respectively. The USB Device Layer and function driver may stall endpoint to indicate error or to indicate a protocol state. The endpoint stall condition may be

cleared in response to a USB Host Clear Feature request. Stalling or clearing the stall on an endpoint will cause all communication requests on the endpoint to be canceled. The function calls are thread safe and interrupt safe. The deviceEndpointIsStalled function is also available to check if an endpoint is in a stalled state. The following code example shows how the USB Device Layer calls these functions to stall and clear the stall on an endpoint.

```
/* The following code example shows how the USB Device Layer calls the driver
 * endpoint stall function (deviceEndpointStall) to stall an endpoint when the a
 * Host send a Set Feature request with feature selector set to endpoint halt.
 * The endpoint to be halted is identified in the setup packet and is identified
 * in this code example as usbEndpoint. Also shown is how the stall clear
 * (deviceEndpointStallClear) and stall status check (deviceEndpointIsStalled)
 * functions are called. */

/* The driverInterface member of the usbDeviceThisInstance structure is a
 * pointer to the USB Driver Common Interface. */

void _USB_DEVICE_ProcessStandardEndpointRequest
(
    USB_DEVICE_OBJ * usbDeviceThisInstance,
    uint8_t interfaceNumber,
    USB_SETUP_PACKET * setupPkt
)
{
    USB_ENDPOINT usbEndpoint;
    usbEndpoint = setupPkt->bEPID;

    if( setupPkt->bRequest == USB_REQUEST_GET_STATUS )
    {
        usbDeviceThisInstance->getStatusResponse.status = 0x00;
        usbDeviceThisInstance->getStatusResponse.endPointHalt
            =  usbDeviceThisInstance->driverInterface->deviceEndpointIsStalled
               (usbDeviceThisInstance->usbCDHandle, usbEndpoint );

        USB_DEVICE_ControlSend( (USB_DEVICE_HANDLE)usbDeviceThisInstance,
                (uint8_t *)&usbDeviceThisInstance->getStatusResponse, 2 );
    }
    else if( setupPkt->bRequest == USB_REQUEST_CLEAR_FEATURE )
    {
        if( setupPkt->wValue == USB_FEATURE_SELECTOR_ENDPOINT_HALT )
        {
            usbDeviceThisInstance->driverInterface->deviceEndpointStallClear
            (usbDeviceThisInstance->usbCDHandle, usbEndpoint );
            USB_DEVICE_ControlStatus((USB_DEVICE_HANDLE)usbDeviceThisInstance,
                                     USB_DEVICE_CONTROL_STATUS_OK );
        }
    }
    else if (setupPkt->bRequest == USB_REQUEST_SET_FEATURE )
    {
        if( setupPkt->wValue == USB_FEATURE_SELECTOR_ENDPOINT_HALT )
        {
            usbEndpoint = setupPkt->bEPID;
            usbDeviceThisInstance->driverInterface->deviceEndpointStall
                                 (usbDeviceThisInstance->usbCDHandle, usbEndpoint );
            USB_DEVICE_ControlStatus((USB_DEVICE_HANDLE)usbDeviceThisInstance,
                                     USB_DEVICE_CONTROL_STATUS_OK );
        }
    }

    /* Additional code is not shown due to space constraints */
}
```

Transferring Data to the Host

Provides information on transferring data to the Host.

Description

The USB Device Layer, the USBCD client, needs to transfer data to the Host in response to enumeration requests for general operation on the device. The USB uses a concept of Input Output Request Packet (IRP) to transfer data to and from the Host. IRPs are transported over endpoints which are enabled by calling the USBCD Endpoint Enable function.

A Device IRP is a USB_DEVICE_IRP type data structure. The IRP is created by the Device Layer and submitted to the USBCD for processing through the deviceIRPSubmit function. At the time of submitting the IRP, the endpoint over which the IRP must be transported is specified. The data request in the IRP is transported using the attributes of the endpoint. When an IRP is submitted to the USBCD, it is owned by the USBCD and cannot be modified by the Device Layer until the USBCD issues an IRP callback. The USBCD will issue the IRP callback when it has completed or terminated processing of the IRP.

An IRP does not have its own transfer type. It inherits the properties of the endpoint to which it is submitted. Hence an IRP becomes a control transfer IRP it was submitted to a control endpoint. An endpoint allows multiple IRPs to be queued. This allows the Device Layer to submit IRPs to an endpoint even while an IRP is being processed on the endpoint. The USBCD will process an IRP in the order that it was received. The following code example shows the USB_DEVICE_IRP data structure:

```
/* This code example shows the USB_DEVICE_IPR structure. The Device Layer
 * uses such a structure to transfer data through the driver. A structure of
 * this type is allocated by the Device Layer and the other function drivers and
 * passed to the deviceIRPSubmit function. */

typedef struct _USB_DEVICE_IRP
{
    /* Pointer to the data buffer */
    void * data;

    /* Size of the data buffer */
    unsigned int size;

    /* Status of the IRP */
    USB_DEVICE_IRP_STATUS status;

    /* IRP Callback. If this is NULL, then there is no callback generated */
    void (*callback)(struct _USB_DEVICE_IRP * irp);

    /* Request specific flags */
    USB_DEVICE_IRP_FLAG flags;

    /* User data */
    uintptr_t userData;

    /**********************************
     * The following members should not
     * be modified by the client
     **********************************/
    uint32_t privateData[3];

} USB_DEVICE_IRP;
```

The `data` member of the USB_DEVICE_IRP structure points to a data buffer. This data buffer will contain the data that needs to be sent to the Host for the data stage of an IN transfer. For an OUT transfer, it will contain the data that was received from the Host. Any hardware specific cache coherency and address alignment requirements must be considered while allocating this data buffer. The Driver Client should not modify or examine the contents of the IRP after the IRP has been submitted and is being processed. It can be examined after the driver has released the IRP.

The `size` member of the USB_DEVICE_IRP structure specifies the size of the data buffer. The transfer will end when the device has sent or received size number of bytes. While sending data to the Host, the IRP size can exceed the size of the transaction (which is equal to the size of the endpoint). The USBCD in such a case will split up the transfer into transactions. This process does not require external intervention. The driver uses receive and transmit IRPs to process control transfers. When the driver receives a Setup packet, the IRP completion status would be USB_DEVICE_IRP_STATUS. The Driver Client should then use additional receive and transmit IRPs to complete the control transfer.

For interrupt and isochronous transfers, the size of transfer specified in the IRP cannot exceed the size of the transaction. If size is specified as 0, then the driver will send or expect a zero length packet. The `size` parameter of the IRP is updated by the driver when IRP processing is completed. This will contain the size of the completed transfer.

The `status` member of the IRP provides the completion status of the IRP and should be checked only when the IRP processing has completed. This is indicated by the driver calling the IRP callback function. The IRP status is a USB_DEVICE_IRP_STATUS type. The following code example shows the different possible values of the `status` member and example usage of IRPs to transfer data between the device and the Host.

```
/* The followoing code shows example usage of the device IRP. The submit status
 * of the IRP is available when IRP submit function returns. The completion
 * status of the IRP is available when the IRP has terminated and the IRP
 * callback function is invoked. The IRP callback
 * function shown in this example shows the possible complete status of the IRP.
 * The end application may or may not handle all the cases. Multiple IRPs can be
 * queued on an endpoint. */

void IRP_Callback(USB_DEVICE_IRP * irp)
{
```

```
    /* irp is pointing to the IRP for which the callback has occurred. In most
     * cases this function will execute in an interrupt context. The application
     * should not perform any hardware access or interrupt unsafe operations in
     * this function. */

    switch(irp->status)
    {
        case USB_DEVICE_IRP_STATUS_TERMINATED_BY_HOST:
            /* The IRP was aborted because the Host cleared the stall on the
             * endpoint */
            break;

        case USB_DEVICE_IRP_STATUS_ABORTED_ENDPOINT_HALT:
            /* IRP was aborted because the endpoint halted */
            break;


        case USB_DEVICE_IRP_STATUS_ABORTED:
            /* USB Device IRP was aborted by the function driver */
            break;

        case USB_DEVICE_IRP_STATUS_ERROR:
            /* An error occurred on the bus when the IRP was being processed */
            break;

        case USB_DEVICE_IRP_STATUS_COMPLETED:
            /* The IRP was completed */
            break;

        case USB_DEVICE_IRP_STATUS_COMPLETED_SHORT:
            /* The IRP was completed but the amount of data received was less
             * than the requested size */
            break;

        default:
            break;

    }
}

/* In the following example, the IRP is submitted to Endpoint 0x84. This is
 * interpreted as an IN direction endpoint (MSB of 0x84 is 1) and Endpoint 4.
 * The data contained in source will be sent to the USB Host. Assuming
 * the endpoint size is 64, the 130 bytes of data in this case will be sent to
 * the Host in three transaction of 64, 64 and 2 bytes. A transaction completes
 * when the Host polls (sends an IN token) the device.  The callback function
 * will then called indicating the completion status of the IRP. The application
 * should not modify the privateData field of the IRP. If the IRP was submitted
 * successfully, the buffer will be owned by the driver until the IRP callback
 * function has been called. Because the size of the transfer is not a multiple
 * of the endpoint size, the IRP flag must be set
 * USB_DEVICE_IRP_FLAG_DATA_COMPLETE. This directs the driver to not perform any
 * explicit signaling to the Host to indicate end of transfer. The last packet
 * in this case is a short packet and this signals the end of the transfer. */

USB_DEVICE_IRP irp;
USB_ERROR result;
uint8_t source[130];

irp.data = source;
irp.size = 130;
irp.called = IRP_Callback;
flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
userData = &someApplicationObject;

result = DRV_USBFS_DEVICE_IRPSubmit(driverHandle, 0x84, &irp);

switch(result)
{
```

```
    case USB_ERROR_PARAMETER_INVALID:
        /* This can happen if the driverHandle is invalid */
        break;

    case USB_ERROR_DEVICE_IRP_IN_USE:
        /* This can happen if the IRP is being resubmitted while it is still in
         * process (it was submitted before but processing has not completed */
        break;

    case USB_ERROR_DEVICE_ENDPOINT_INVALID;
        /* The endpoint to which this IRP is being submitted is not provisioned
         * in the system. This is controller by DRV_USBFS_ENDPOINTS_NUMBER
         * configuration parameter. */
        break;

    case USB_ERROR_ENDPOINT_NOT_CONFIGURED:
        /* The endpoint to which this IRP is being submitted is not enabled. It
         * must be enabled by calling the DRV_USBFS_DEVICE_EndpointEnable()
         * function. */
        break;

    case USB_ERROR_PARAMETER_INVALID:
        /* The USB_DEVICE_IRP_FLAG_DATA_PENDING flag was specified but the
         * transfer size is not a multiple of the endpoint size. If the IRP was
         * submitted to a receive endpoint, this error can occur if the size is
         * not a multiple of the endpoint size. */
        break;

    case USB_ERROR_OSAL_FUNCTION:
        /* An error occurred while trying to grab a mutex. This is applicable
         * when the driver is running with a RTOS. */
        break;

    case USB_ERROR_NONE:
        /* The IRP was submitted successfully. */
        break;

    default:
        break;
}

/* The following code example shows how an IRP is submitted to an OUT endpoint.
 * In this case data will be pointing to a buffer where the received data will
 * be stored. Note that the size of the IRP should be a multiple of the endpoint
 * size. The flags parameter is ignored in the data receive case. The IRP
 * terminates when the specified size of bytes has been received (the Host sends
 * OUT packets) or when a short packet has been received. */

USB_DEVICE_IRP irp;
USB_ERROR result;
uint8_t destination[128];

irp.data = destination;
irp.size = 128;
irp.called = IRP_Callback;
userData = &someApplicationObject;

result = DRV_USBFS_DEVICE_IRPSubmit(driverHandle, 0x04, &irp);
```

For IRPs submitted to an Interrupt or Isochronous endpoints, the driver will always send either the less than or equal to the maximum endpoint packet size worth of bytes in a transaction. The application could either submit an IRP per Interrupt/Isochronous polling interval or it could submit one IRP for multiple polling intervals.

The `flags` member of the USB_DEVICE_IRP structure specifies flags which affect the behavior of the IRP. The USB_DEVICE_IRP_FLAG enumeration specifies the available option. The USB_DEVICE_IRP_FLAG_DATA_COMPLETE causes the driver to add a Zero Length Packet (ZLP) to the data stage of the IN transfer when the transfer size is an exact multiple of the endpoint size. If the transfer size is not a multiple of the endpoint size, no ZLP will be sent. The USB_DEVICE_IRP_FLAG_PENDING flag will cause the driver to not send a ZLP in a case where the size of the IN transfer is an exact multiple of the endpoint size. The following code example demonstrates this.

```
/* In the following code example, the IRP is submitted to an IN endpoint whose size
 * is 64. The transfer size is 128, which is an exact multiple of the endpoint
```

```
 * size. The flag is set to USB_DEVICE_IRP_FLAG_DATA_COMPLETE. The driver
 * will send two transactions of 64 bytes each and will then automatically send a
 * Zero Length Packet (ZLP), thus completing the transfer. The IRP callback will
 * be invoked when the ZLP transaction has completed. */

USB_DEVICE_IRP irp;
USB_ERROR result;
uint8_t source[128];

irp.data = source;
irp.size = 128;
irp.called = IRP_Callback;
flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
userData = &someApplicationObject;

result = DRV_USBFS_DEVICE_IRPSubmit(driverHandle, 0x84, &irp);

/* In the following code example, the IRP is submitted to an IN endpoint whose size
 * is 64. The transfer size is 128, which is an exact multiple of the endpoint
 * size. The flag is set to to USB_DEVICE_IRP_FLAG_DATA_PENDING. The driver will
 * send two transactions of 64 bytes each but will not send a ZLP. The USB Host
 * can then consider that there is more data pending in the transfer. The IRP
 * callback will be invoked when the two transactions have completed. */

USB_DEVICE_IRP irp;
USB_ERROR result;
uint8_t source[128];

irp.data = source;
irp.size = 128;
irp.called = IRP_Callback;
flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
userData = &someApplicationObject;

result = DRV_USBFS_DEVICE_IRPSubmit(driverHandle, 0x84, &irp);
```

The `callback` member of the USB_DEVICE_IRP structure points to a function which the driver calls when the IRP processing is completed. The Driver Client must implement this function and assign the pointer to this function to the `callback` member of the IRP. Every IRP can have its own callback function or one common callback function could be used. The callback function will execute in an interrupt context. The Driver Client should not execute interrupt unsafe, blocking or computationally intensive operations in the callback function. The client can call deviceIRPSubmit function in the IRP callback function to submit another IRP or resubmit the same IRP. The client can check the status and size of the IRP in the callback function.

The `userData` member of the USB_DEVICE_IRP structure can be used by the client to associate a client specific context with the Host. This context can then be used by the client, in the IRP callback function to identify the context in which the IRP was submitted. This member is particularly useful if the client wants to implement one callback function for all IRPs.

The `privateData` member of the IRP is used by the driver and should not be accessed or manipulated by the Driver Client.

PIC32MX USB Driver

Provides information on the USB Driver specific to PIC32MX devices.

Description

The PIC32MX USB Driver in MPLAB Harmony provides API functions that allow the MPLAB Harmony USB Host and Device Stack to access the USB while operating on a PIC32MX microcontroller. The driver implements the USB Driver Common Interface required by the USB Host and Device Stack. It abstracts the USB module operational details from the Host and Device Stack and provides the stacks with a modular access mechanism to the USB. The PIC32MX USB Driver features the following:

- USB 2.0 Full Speed operation in Peripheral mode
- USB 2.0 Full Speed and Low Speed USB Peripheral Support in Host mode
- Designed for Dual Role Operation
- Capable of operating multiple USB modules
- Features non-blocking function and is interoperable with other MPLAB Harmony modules
- Features thread safe functions when operating within an RTOS
- Capable of operating in Polled and Interrupt modes
- Implements the USB Driver Common Interface required by the MPLAB Harmony USB Host and Device Stack
- Completely configurable through MPLAB Harmony Configurator (MHC) tool
- Implements feature separation (Host and Device mode functions are implemented across different files)

Note: This help section only discusses features that are unique to the PIC32MX USB Driver and are not a part of the USB Driver Common Interface. The driver functions that implement the USB Driver Common Interface are described in the Common Interface Help section.

While the PIC32MX USB module supports USB "On-The-Go" (OTG), this release of the PIC32MX Driver does not implement USB OTG protocol support.

This help section only provides relevant information about the operation of the USB. The reader is encouraged to refer to the USB 2.0 Specification available at www.usb.org for a detailed explanation of USB protocol.

Using the Library

This topic describes the basic architecture of the PIC32MX USB Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_usbfs.h`

The interface to the PIC32MX USB Driver library is defined in the `drv_usbfs.h` header file.

Please refer to the What is MPLAB Harmony? section for how the Driver interacts with the framework.

Library Overview

Provides an overview of the library.

Description

The PIC32MX USB Driver will typically be used by a USB Host and/or Device Stack. The USB Host and Device Stack operate as driver client applications. The driver is initialized as part of the MPLAB Harmony System Initialization. The driver initialization data structure specifies the operation mode (Host, Device, or Dual Role) of the driver. The driver features task routines to be called in the MPLAB Harmony application tasks function (SYS_Tasks function) and the USB Module Interrupt Service Routine (ISR).

The Host and the Device Stack can open the driver only when initialization has completed. It will continue to return an invalid driver handle while the initialization is in progress. Once opened, the Device Mode function can be called if the driver is operating in Device mode. The Host Mode function can be called if the driver is operating in Host mode. In Dual Role operation mode, the driver supports Host and Device operation in the same application. Even then, the driver will either operate as a USB Host or Device. OTG operation is not supported.

The PIC32MX USB Driver features RTOS thread-safe functions. This allows the driver client application to safely call driver functions across different RTOS threads. Not all of the driver functions are interrupt-safe.

In addition to the USB Driver, which implements the USB Driver Common Interface, the PIC32MX USB Driver implements functions which are required for its operation in the MPLAB Harmony framework. The following table lists the different categories of functions in the PIC32MX USB Driver.

Library Interface Section	Description
System Function	These functions are accessed by the MPLAB Harmony System module. They allow the driver to be initialized, deinitialized and maintained. These functions are implemented in the `drv_usbfs.c` source file.
Client Core Functions	These functions allow the USB Host and Device Stack to open, close and perform other general driver operations. These functions are a part of the USB Driver Common Interface and are implemented in `drv_usbfs.c` source file.
Device Mode Operation Functions	These functions allow the USB Device Stack to perform USB Device mode specific driver operations. These functions are a part of the USB Driver Common Interface and are implemented in `drv_usbfs_device.c` source file
Host Mode Operation Functions	These functions allow the USB Host Stack to perform USB Host mode specific driver operations. These functions are a part of the USB Driver Common Interface and are implemented in `drv_usbfs_host.c` source file.
Root Hub Functions	These functions allow the USB Host Stack to access the driver Root hub operation. These functions are a part of the USB Driver Common Interface and are implemented in `drv_usbfs_host.c` source file.

Abstraction Model

Provides information on the abstraction model for the library.

Description

The PIC32MX USB Driver implements the abstraction model defined by the USB Driver Common interface. This interface abstracts USB module specific details and provides a module independent interface to the driver client applications.

While operating in Device mode, the driver expects the client application (the USB Device Stack) to enable endpoints and then submit I/O request

packet (IRP) requests to the enabled endpoints. Multiple IRPs can be queued on an endpoint. The driver calls the IRP callback function when the IRP is processed. The driver allows the client application to also attach and detach the device on the bus. It generates events which indicate USB states.

While operating in Host mode, the driver expects the client application (the USB Host Stack) to open pipes to endpoints on the connected device. The client application can then submit IRPs to the pipes. Multiple IRPs can be queued on a pipe. The driver calls the IRP callback function when the IRP is processed. The driver will call application defined functions to enumerate and denumerate a device. These functions are called when the driver detect device attach and detach respectively. The driver also exports root hub functions to the client application. This allows the client application to treat the driver as a single port hub

Please refer to the PIC32 USB Driver Common Interface help section for more details on the driver abstraction model.

How the Library Works

Provides information on how the library works.

Description

This section only explains aspects of driver operation which are unique to the PIC32MX USB Driver. Major driver operations are described in the PIC32 USB Driver Common Interface help section.

Driver Initialization

Note: While generating a MPLAB Harmony USB project with MHC, the initialization code for the driver is generated automatically based on selections made in the USB Host stack or Device Stack Configuration trees.

The PIC32MX USB Driver must be initialized so that a client application can open. The client application will not be able to open the driver if the initialization is in progress or has failed. The driver is initialized by calling the DRV_USBFS_Initialize function. This function is called from the SYS_Initialize function in the MPLAB Harmony application project and accepts two input parameters. The index parameter defines the instance of the USB Driver to be initialized. This becomes significant when the PIC32MX microcontroller has more than one USB module. The init parameter is a driver specific data structure of the type DRV_USBFS_INIT. This structure is shown in the following code example.

```
/* This code snippet show the PIC32MX USB Driver Initialization data structure.
 * A structure of this type must be provided to the DRV_USBFS_Initialize()
 * function. */

typedef struct
{
    /* System Module Initialization */
    SYS_MODULE_INIT moduleInit;

    /* Identifies the USB peripheral to be used. This should be the USB PLIB
       module instance identifier. */
    uint8_t usbID;

    /* This should be set to true if the USB module must stop operation in IDLE
       mode */
    bool stopInIdle;

    /* This should be set to true if the USB module must suspend when the CPU
       enters sleep mode. */
    bool suspendInSleep;

    /* Specify the interrupt source for the USB module. This should be Interrupt
       PLIB Interrupt source identifier for the USB module instance specified in
       usbID. */
    INT_SOURCE interruptSource;

    /* Specify the operational speed of the USB module. This should always be
       set to USB_SPEED_FULL. The use of this parameter is deprecated. */
    USB_SPEED operationSpeed;

    /* Specify the operation mode of the USB module. This defines if the USB
     * module will support Device, Host or Dual Role operation */
    DRV_USBFS_OPMODES operationMode;

    /* A pointer to the endpoint descriptor table. This should be aligned at 512
       byte address boundary. The size of the table is equal to the
       DRV_USBFS_ENDPOINT_TABLE_ENTRY_SIZE times the number of endpoints needed
       in the application. */
    void * endpointTable;
```

```
    /* Root hub available current in mA. This specifies the amount of current
       that root hub can provide to the attached device. This should be
       specified in mA. This is required when the driver is required to operate
       in host mode. */
    uint32_t rootHubAvailableCurrent;

    /* When operating in Host mode, the application can specify a Root Hub port
       enable function. This parameter should point to Root Hub port enable
       function. If this parameter is NULL, it implies that the Port is always
       enabled. */
    DRV_USBFS_ROOT_HUB_PORT_POWER_ENABLE portPowerEnable;

    /* When operating in Host mode, the application can specify a Root Port
       Indication. This parameter should point to the Root Port Indication
       function. If this parameter is NULL, it implies that Root Port Indication
       is not supported. */
    DRV_USBFS_ROOT_HUB_PORT_INDICATION portIndication;

    /* When operating is Host mode, the application can specify a Root Port
       Overcurrent detection. This parameter should point to the Root Port
       Indication function. If this parameter is NULL, it implies that
       Overcurrent detection is not supported. */
    DRV_USBFS_ROOT_HUB_PORT_OVER_CURRENT_DETECT portOverCurrentDetect;

} DRV_USBFS_INIT;
```

The operationMode parameter defines by the driver operation mode. parameter in the initialization data structure. This can be set DRV_USBFS_OPMODE_DEVICE, DRV_USBFS_OPMODE_HOST or DRV_USBFS_OPMODE_DUAL_ROLE for device, host and dual role operation respectively.

The `endpointTable` parameter must point to a byte array. The size of the array depends on the maximum number of device endpoints that application needs. A direction of an endpoint counts as one endpoint. Each endpoint requires 32 bytes. Therefore, if the USB Device requires 3 endpoints (Endpoint 0 + Endpoint 1 IN + Endpoint 2 OUT), the size of the array will 96 bytes (32 * 3). The byte array start address must be located on a 512 byte boundary. When operating in host mode, the driver will use only one endpoint and size of the endpoint table array should be set to 32.

The `rootHubAvailableCurrent` parameter should be set to the maximum current that VBUS power supply can provide on the bus. The driver does not use this information directly. It provides this data to the client application while operating in host mode.

The `portPowerEnable` parameter must point to a Port Power Enable function. The driver, while operating in host mode, will call this function to enable the VBUS switch. This function should activate the VBUS switch if the driver calls this function with the enable parameter set to true. It should deactivate the switch if the driver calls this function with the enable parameter set to false. This parameter should be set to NULL if such a switch (of the switch control) is not available in the application.

The `portIndication` parameter must point to a Port Indication function. The driver, while operating in host mode, will call this function to indicate the current state of the port. The driver will call this function with LED color status as defined in the Chapter 11 of the USB 2.0 Specification. This parameter should be set to NULL if such a LED indication is not available in the application.

The `portOverCurrentDetect` parameter must point to a Port Overcurrent Detect function. The driver, while operating in Host mode, will call this function periodically to check if the attached device is overdrawing current. If the function should return true if such a condition exists. This parameter should be set to NULL if such detection is not available in the application.

The following code example shows initialization of the driver for device mode operation.

```
/* This code shows an example of DRV_USBFS_INIT data structure for
 * device mode operation. Here the driver is initialized to work with USB1 USB
 * module. Note how the endPointTable is defined. It should be aligned on a 512
 * byte boundary. */

DRV_USBFS_INIT init;
SYS_MODULE_OBJ usbDriverObj;

uint8_t __attribute__((aligned(512))) endPointTable[DRV_USBFS_ENDPOINTS_NUMBER * 32];

const DRV_USBFS_INIT drvUSBInit =
{
    /* Assign the endpoint table */
    .endpointTable = endPointTable,

    /* Interrupt Source for USB module */
    .interruptSource = INT_SOURCE_USB_1,

    /* System module initialization. */
    .moduleInit = {SYS_MODULE_POWER_RUN_FULL},
```

```
    /* Configure the driver for device mode operation. */
    .operationMode = DRV_USBFS_OPMODE_DEVICE,

    /* Configure the driver to operate at full speed. */
    .operationSpeed = USB_SPEED_FULL,

    /* Stop in idle */
    .stopInIdle = false,

    /* Suspend in sleep */
    .suspendInSleep = false,

    /* Identifies peripheral (PLIB-level) ID */
    .usbID = USB_ID_1
};

void SYS_Initialize(void)
{
    /* Initialize the USB Driver. Note how the init parameter is typecasted to
     * SYS_MODULE_INIT type. The SYS_MODULE_OBJ returned by this function call
     * is passed to the driver tasks routine. DRV_USBFS_INDEX_0 is helper
     * constant defined in drv_usbfs.h */

    usbDriverObj = DRV_USBFS_Initialize(DRV_USBFS_INDEX_0, (SYS_MODULE_INIT *)(drvUSBInit));
}

void SYS_Tasks(void)
{
    /* The polled state of the USB driver is updated by calling the
     * DRV_USBFS_Tasks function in the SYS_Tasks() function. The
     * DRV_USBFS_Tasks() takes the driver module object returned by the
     * DRV_USBFS_Initialize funciton as a parameter. */

    DRV_USBFS_Tasks(usbDriverObj);
}

void __ISR(_USB_1_VECTOR, ipl4AUTO) _IntHandlerUSBInstance0(void)
{
    /* The DRV_USBFS_Tasks_ISR function update the interrupt state of the USB
     * Driver. If the driver is configured for polling mode, this function need
     * not be invoked or included in the project. */

    DRV_USBFS_Tasks_ISR(sysObj.drvUSBObject);
}
```

The following code example shows initialization of the driver for host mode operation.

```
/* This code shows an example of the USBFS driver can be configured for
 * host mode operation. For host mode operation, only one endpoint is needed and
 * hence the size of the endpoint table is 32 bytes (for one endpoint). In this
 * example, the BSP_USBVBUSSwitchOverCurrentDetect function checks for over
 * current condition and the BSP_USBVBUSPowerEnable function enables the VBUS
 * power. The port indication function is not implemented and hence the
 * portIndication member of the initialization data structure is set to NULL. */

/* The implementation of the port over current detect, indication and the VBUS
 * power supply functions is discussed later in this help section. */

uint8_t __attribute__((aligned(512))) endpointTable[32];

DRV_USBFS_INIT drvUSBFSInit =
{
    /* Pointer to the endpoint table */
    .endpointTable = endpointTable,

    /* Interrupt Source for the USB module */
    .interruptSource = INT_SOURCE_USB_1,

    /* This should always be set to SYS_MODULE_POWER_RUN_FULL. */
    .moduleInit = {SYS_MODULE_POWER_RUN_FULL},
```

```
    /* Configure for host mode operation. */
    .operationMode = DRV_USBFS_OPMODE_HOST,

    /* The driver should run at full speed. */
    .operationSpeed = USB_SPEED_FULL,

    /* Port indication function is not implemented. */
    .portIndication = NULL,

    /* This is the over current detect function. */
    .portOverCurrentDetect = BSP_USBVBUSSwitchOverCurrentDetect,

    /* This is the VBUS Power enable function */
    .portPowerEnable = BSP_USBVBUSPowerEnable,

    /* Here we state that the VBUS power supply can provide at most 500 mA of
     * current */
    .rootHubAvailableCurrent = 500,

    /* Module will operate in IDLE. */
    .stopInIdle = false,

    /* Module will not suspend automatically in sleep */
    .suspendInSleep = false,

    /* USB Module ID is 1 */
    .usbID = USB_ID_1
};

void SYS_Initialize(void)
{
    /* Initialize the USB Driver. Note how the init parameter is typecasted to
     * SYS_MODULE_INIT type. The SYS_MODULE_OBJ returned by this function call
     * is passed to the driver tasks routine. DRV_USBFS_INDEX_0 is helper
     * constant defined in drv_usbfs.h */

    usbDriverObj = DRV_USBFS_Initialize(DRV_USBFS_INDEX_0,
                                        (SYS_MODULE_INIT *)(drvUSBInit));
}

void SYS_Tasks(void)
{
    /* The polled state of the USB driver is updated by calling the
     * DRV_USBFS_Tasks function in the SYS_Tasks() function. The
     * DRV_USBFS_Tasks() takes the driver module object returned by the
     * DRV_USBFS_Initialize funciton as a parameter. */

    DRV_USBFS_Tasks(usbDriverObj);
}

void __ISR(_USB_1_VECTOR, ipl4AUTO) _IntHandlerUSBInstance0(void)
{
    /* The DRV_USBFS_Tasks_ISR function update the interrupt state of the USB
     * Driver. If the driver is configured for polling mode, this function need
     * not be invoked or included in the project. */

    DRV_USBFS_Tasks_ISR(sysObj.drvUSBObject);
}
```

The PIC32MX USB Driver requires definition of configuration constants to be available in the `system_config.h` file of the MPLAB Harmony Application Project Configuration. Refer to the Configuring the Library section for details.

Multi-client Operation

The PIC32MX USB Driver supports multi-client operation. In that, it can be opened by two application clients. This is required where Dual Operation is desired. The following should be noted when using multi-client operation:

- The driver should be initialized for Dual Role Operation mode.
- The DRV_USBFS_Open function can be called at the most twice in the application. The driver supports a maximum of two clients.

- A client can access either the host or device functionality of the driver. It cannot do both.
- It is possible for the two clients to operate in two different threads while operating with an RTOS.

Note: The typical the application clients for PIC32MX USB Driver would be the MPLAB Harmony USB Host and Device Stack. The complexity of operating the driver in Dual Role mode is handled by the stack operation. The MHC will configure the driver for Dual Role operation when such operation is selected in USB Stack configuration tree.

USB Driver Common Interface

The PIC32MX USB Driver exports its implementation of the USB Driver Common Interface to the Host and Device Layer via the DRV_USBFS_HOST_INTERFACE and DRV_USBFS_DEVICE_INTERFACE structures. The DRV_USBFS_HOST_INTERFACE structure is defined in the `drv_usbfs_host.c` file. The following code example shows this structure.

```
/*********************************************************
 * This structure is a set of pointer to the USBFS driver
 * functions. It is provided to the host and device layer
 * as the interface to the driver.
 * *******************************************************/

DRV_USB_HOST_INTERFACE gDrvUSBFSHostInterface =
{
    .open = DRV_USBFS_Open,
    .close = DRV_USBFS_Close,
    .eventHandlerSet = DRV_USBFS_ClientEventCallBackSet,
    .hostIRPSubmit = DRV_USBFS_HOST_IRPSubmit,
    .hostIRPCancel = DRV_USBFS_HOST_IRPCancel,
    .hostPipeSetup = DRV_USBFS_HOST_PipeSetup,
    .hostPipeClose = DRV_USBFS_HOST_PipeClose,
    .hostEventsDisable = DRV_USBFS_HOST_EventsDisable,
    .hostEventsEnable = DRV_USBFS_HOST_EventsEnable,
    .rootHubInterface.rootHubPortInterface.hubPortReset = DRV_USBFS_HOST_ROOT_HUB_PortReset,
    .rootHubInterface.rootHubPortInterface.hubPortSpeedGet = DRV_USBFS_HOST_ROOT_HUB_PortSpeedGet,
    .rootHubInterface.rootHubPortInterface.hubPortResetIsComplete =
DRV_USBFS_HOST_ROOT_HUB_PortResetIsComplete,
    .rootHubInterface.rootHubPortInterface.hubPortSuspend = DRV_USBFS_HOST_ROOT_HUB_PortSuspend,
    .rootHubInterface.rootHubPortInterface.hubPortResume = DRV_USBFS_HOST_ROOT_HUB_PortResume,
    .rootHubInterface.rootHubMaxCurrentGet = DRV_USBFS_HOST_ROOT_HUB_MaximumCurrentGet,
    .rootHubInterface.rootHubPortNumbersGet = DRV_USBFS_HOST_ROOT_HUB_PortNumbersGet,
    .rootHubInterface.rootHubSpeedGet = DRV_USBFS_HOST_ROOT_HUB_BusSpeedGet,
    .rootHubInterface.rootHubInitialize = DRV_USBFS_HOST_ROOT_HUB_Initialize,
    .rootHubInterface.rootHubOperationEnable = DRV_USBFS_HOST_ROOT_HUB_OperationEnable,
    .rootHubInterface.rootHubOperationIsEnabled = DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled,
};
```

The DRV_USBFS_DEVICE_INTERFACE structure is defined in the `drv_usbfs_device.c` file. The following code example shows this structure. The MPLAB Harmony USB Host and Device stack perform driver independent access through the function pointers contained in these structures.

```
/*************************************************
 * This structure is a pointer to a set of USB Driver
 * Device mode functions. This set is exported to the
 * device layer when the device layer must use the
 * PIC32MX USB Controller.
 ************************************************/

DRV_USB_DEVICE_INTERFACE gDrvUSBFSDeviceInterface =
{
    .open = DRV_USBFS_Open,
    .close = DRV_USBFS_Close,
    .eventHandlerSet = DRV_USBFS_ClientEventCallBackSet,
    .deviceAddressSet = DRV_USBFS_DEVICE_AddressSet,
    .deviceCurrentSpeedGet = DRV_USBFS_DEVICE_CurrentSpeedGet,
    .deviceSOFNumberGet = DRV_USBFS_DEVICE_SOFNumberGet,
    .deviceAttach = DRV_USBFS_DEVICE_Attach,
    .deviceDetach = DRV_USBFS_DEVICE_Detach,
    .deviceEndpointEnable = DRV_USBFS_DEVICE_EndpointEnable,
    .deviceEndpointDisable = DRV_USBFS_DEVICE_EndpointDisable,
    .deviceEndpointStall = DRV_USBFS_DEVICE_EndpointStall,
    .deviceEndpointStallClear = DRV_USBFS_DEVICE_EndpointStallClear,
    .deviceEndpointIsEnabled = DRV_USBFS_DEVICE_EndpointIsEnabled,
    .deviceEndpointIsStalled = DRV_USBFS_DEVICE_EndpointIsStalled,
    .deviceIRPSubmit = DRV_USBFS_DEVICE_IRPSubmit,
```

```
    .deviceIRPCancel = DRV_USBFS_DEVICE_IRPCancel,
    .deviceIRPCancelAll = DRV_USBFS_DEVICE_IRPCancelAll,
    .deviceRemoteWakeupStop = DRV_USBFS_DEVICE_RemoteWakeupStop,
    .deviceRemoteWakeupStart = DRV_USBFS_DEVICE_RemoteWakeupStart,
    .deviceTestModeEnter = NULL


};
```

Operation with RTOS

The PIC32MX USB Driver is designed to operate with a RTOS. The driver implementation uses the MPLAB Harmony Operating System Abstraction Layer (OSAL). This allows the driver to function with entire range of RTOSes supported in MPLAB Harmony. The following points must be considered while using the driver with an RTOS.

- The driver can be opened from different threads
- In Device mode, an enabled endpoint should only be accessed from one thread. For example, if an application requires two endpoints, Endpoint 2 and Endpoint 3, the application could contain two threads, one accessing Endpoint 2 and another accessing Endpoint 3. The thread accessing Endpoint 2 cannot access Endpoint 3.
- While operating in Host mode, endpoint pipes can be opened from different threads. A pipe handle to an open pipe cannot be shared across threads.

Host Mode Attach Detach Operation

When the PIC32MX USB Driver operating in Host mode detects a device attach or detach, it implements debouncing before signaling attach detach signal to the USB Host Stack. When the device is attached, the driver waits for DRV_USBFS_HOST_ATTACH_DEBOUNCE_DURATION milliseconds to allow for the mechanical chatter, which occurs when the device is inserted into the host receptacle, to settle. If the device is still attached after the DRV_USBFS_HOST_ATTACH_DEBOUNCE_DURATION expires, the driver calls the USB_HOST_DeviceEnumerate function to let the host stack enumerate the device. It also starts checking for Device Detach.

When the device is detached, the driver waits for DRV_USBFS_POST_DETACH_DELAY milliseconds to allow for the detach operation to settle. If the device is indeed detached after the DRV_USBFS_POST_DETACH_DELAY delay expires, the driver calls USB_HOST_DeviceDenumerate function to let the USB Host stack denumerate the device. It then starts checking for device attach.

Root Hub Operation

The PIC32MX USB Driver implements a Root Hub Driver Interface. This allows the driver to emulate a hub. The USB Host Stack enumerates the Root Hub as a device. The Host Stack then does not differentiate between an external hub and the root hub. While emulating a hub, the PIC32MX USB Driver Root Hub appears as a single port hub.

As a part of the root hub interface, the PIC32MX USB Driver requires the application to supply functions for hub features that it does not implement. These features are:

- Port Overcurrent Detect
- VBUS Switch Control
- Port Indication

A pointer to these functions (if implemented) must be supplied through the driver initialization data (of the type DRV_USBFS_INIT) structure at the time of driver initialization. The application has the option of not implementing these functions. In such a case, the function pointers for the unimplemented function, in the initialization data structure should be set to NULL.

The root hub driver must also be able to communicate the maximum current capability of its port to the USB Host Layer. The PIC32MX USB Controller does not contain built-in (hardware implemented) functionality for controlling the root hub port current. To facilitate this request, the driver will report the current capability that was specified in the `rootHubAvailableCurrent` parameter of the driver initialization data structure. The application must set this parameter to report the current supply capability of the VBUS power supply. The USB Host Layer uses this value to manage the bus current budget. If a connected device reports a configuration that requires more current than what the VBUS power supply can provide, the host will not set the configuration.

Port Overcurrent Detect

The Root Hub operation in PIC32MX USB Driver will periodically call a Port Overcurrent Detect function to detect if an overcurrent condition is active on the port. The application must supply this function if port overcurrent detection is needed. The PIC32MX USB Controller does not contain built-in (hardware implemented) functionality for checking overcurrent condition. The overcurrent condition on the port can occur in a case where the attached device has malfunctioned or when the USB VBUS line has short circuited to ground.

The signature of the function and an example implementation is shown in the following code example. The function must return (and must continue to return) true if an overcurrent condition exists on the port.

```
/* This code shows an example implementation of the
 * portOverCurrentDetect function. The PIC32MX USB Driver will call this
 * function periodically to check if an over current condition exists on the
 * port. In this example, we assume that the over current detect pin from an
 * external circuit in the system, is connected to  port RD0 and the pin logic
 * is active high. The function must return true if an over current condition is
 * present on this pin */
```

```
bool BSP_USBVBUSSwitchOverCurrentDetect(uint8_t port)
{
    if(PLIB_PORTS_PinGet(PORTS_ID_0, PORT_CHANNEL_D, 0) == 1)
    {
        return(true);
    }
    else
    {
        return(false);
    }
}
```

VBUS Switch Control

The PIC32MX USB Driver Root Hub operation will attempt to control the VBUS power supply to the port. Because the PIC32MX USB Controller does not contain built-in (hardware implemented) functionality for checking controlling VBUS, such a control function must be supplied by the application. The root hub operation will access this function when the PIC32MX USB Driver will call the portPowerEnable function as a part of the Bus Enable sequence.

The following code shows an example of how this function can be implemented.

```
/* This code shows an example implementation of the VBUS Power Enable
 * function. The PIC32MX USB Driver will call this function as a part of bus
 * enable function. In this example, it is assumed that system contains an
 * external VBUS power switch and this is control by port RB5.
 */

void BSP_USBVBUSPowerEnable(uint8_t port, bool enable)
{
    if(enable)
    {
        PLIB_PORTS_PinSet(PORTS_ID_0, PORT_CHANNEL_B, PORTS_BIT_POS_5);
    }
    else
    {
        PLIB_PORTS_PinClear(PORTS_ID_0, PORT_CHANNEL_B, PORTS_BIT_POS_5);
    }
}
```

Port Indication function

The Root Hub Operation in the PIC32MX USB Driver allows display of Port LED status. If the application requires this indication, it must implement a function which the Root Hub operation would call when a change in the Root Hub port has occurred. The port indication operation is specified in Section 11.5.3 of the USB 2.0 Specification.

```
/* This code shows an example implementation of the port indication
 * function. The PIC32MX USB Driver calls this function when it wants to indicate
 * port status. It is assumed that three function to switch off, blink and
 * switch on an LED are available. It is further assumed that these function
 * accept the color of the LED to operated on. */

void BSP_RootHubPortIndication
(
    uint8_t port,
    USB_HUB_PORT_INDICATOR_COLOR color,
    USB_HUB_PORT_INDICATOR_STATE state
)
{
    /* The color parameter indicates the color of the LED to be affected. The
     * color will be either USB_HUB_PORT_INDICATOR_COLOR_GREEN or
     * USB_HUB_PORT_INDICATOR_COLOR_AMBER. */

    switch (state)
    {
        case USB_HUB_PORT_INDICATOR_STATE_OFF:
            BSP_SwitchLEDOff(color);
            break;
        case USB_HUB_PORT_INDICATOR_STATE_BLINKING:
            BSP_LEDBlink(color);
            break;
        case USB_HUB_PORT_INDICATOR_STATE_ON:
            BSP_SwitchLEDOn(color);
```

```
            break;
        default:
            break;
    }
}
```

Configuring the Library

Provides information on the configuring the library.

Macros

	Name	Description
	DRV_USBFS_DEVICE_SUPPORT	Determines if the USB Device Functionality should be enabled.
	DRV_USBFS_ENDPOINTS_NUMBER	Configures the number of endpoints to be provisioned in the driver.
	DRV_USBFS_HOST_ATTACH_DEBOUNCE_DURATION	Configures the time duration (in milliseconds) that the driver will wait to re-confirm a device attach.
	DRV_USBFS_HOST_NAK_LIMIT	Configures the NAK Limit for Host Mode Control Transfers.
	DRV_USBFS_HOST_PIPES_NUMBER	Configures the maximum number of pipes that are can be opened when the driver is operating in Host mode.
	DRV_USBFS_HOST_RESET_DURATION	Configures the time duration (in milliseconds) of the Reset Signal.
	DRV_USBFS_HOST_SUPPORT	Determines if the USB Host Functionality should be enabled.
	DRV_USBFS_INSTANCES_NUMBER	Specifies the number of driver instances to be enabled in the application.
	DRV_USBFS_INTERRUPT_MODE	Configures the driver for interrupt or polling mode operation.

Description

The PIC32MX USB Driver requires the specification of compile-time configuration macros. These macros define resource usage, feature availability, and dynamic behavior of the driver. These configuration macros should be defined in the `system_config.h` file.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_USBFS_DEVICE_SUPPORT Macro

Determines if the USB Device Functionality should be enabled.

File

drv_usbfs_config_template.h

C

```
#define DRV_USBFS_DEVICE_SUPPORT true
```

Description

USB Full Speed Driver Device Mode Support.

This constant should be set to true if USB device support is required in the application. It should be set to false if device support is not required.

Remarks

This constant should always be defined.

DRV_USBFS_ENDPOINTS_NUMBER Macro

Configures the number of endpoints to be provisioned in the driver.

File

drv_usbfs_config_template.h

C

```
#define DRV_USBFS_ENDPOINTS_NUMBER 3
```

Description

USB Full Speed Driver Endpoint Numbers.

This constant configures the number of endpoints that the driver needs to manage. When DRV_USBFS_DEVICE_SUPPORT is enabled, this

constant should be set to the total number of endpoints to be enabled in the device. When enabled, a endpoint can be used for communication. Using any direction of an endpoint will require that entire endpoint to be enabled.

Consider the case of a composite USB Device that containing a CDC and MSD function. The CDC function will require 1 Bulk endpoint (OUT and IN directions) and 1 Interrupt endpoint (IN direction). The MSD function will require 1 Bulk endpoint (IN and OUT directions). This design can be implemented by using 4 endpoints. Endpoint 0 is used for the mandatory control interface. Endpoint 1 is used for CDC Bulk interface. Endpoint 2 is used for CDC interrupt interface and endpoint 3 is used for MSD Bulk Interface. The constant should then be set to 4.

For Host mode operation, this constant should be set to 1. Setting this to greater than 1 will result in unused data memory allocation.

Remarks

This constant should always be defined.

DRV_USBFS_HOST_ATTACH_DEBOUNCE_DURATION Macro

Configures the time duration (in milliseconds) that the driver will wait to re-confirm a device attach.

File

drv_usbfs_config_template.h

C

```
#define DRV_USBFS_HOST_ATTACH_DEBOUNCE_DURATION 500
```

Description

USB Full Speed Driver Host Mode Attach Debounce Duration.

This constant configures the time duration (in milliseconds) that driver will wait to re-confirm a device attach. When the driver first detects device attach, it start, it will start a timer for the duration specified by the constant. When the timer expires, the driver will check if the device is still attached. If so, the driver will then signal attach to the host stack. The duration allows for device attach to become electro-mechanically stable.

Remarks

This constant should always be defined when DRV_USBFS_HOST_SUPPORT is set to true.

DRV_USBFS_HOST_NAK_LIMIT Macro

Configures the NAK Limit for Host Mode Control Transfers.

File

drv_usbfs_config_template.h

C

```
#define DRV_USBFS_HOST_NAK_LIMIT 2000
```

Description

USB Full Speed Driver Host Mode Control Transfers NAK Limit.

This constant configures the number of NAKs that the driver can accept from the device in the data stage of a control transfer before aborting the control transfer with a USB_HOST_IRP_STATUS_ERROR_NAK_TIMEOUT. Setting this constant to 0 will disable NAK limit checking. This constant should be adjusted to enable USB host compatibility with USB Devices which require more time to process control transfers.

Remarks

This constant should always be defined when DRV_USBFS_HOST_SUPPORT is set to true.

DRV_USBFS_HOST_PIPES_NUMBER Macro

Configures the maximum number of pipes that are can be opened when the driver is operating in Host mode.

File

drv_usbfs_config_template.h

C

```
#define DRV_USBFS_HOST_PIPES_NUMBER 10
```

Description

USB Full Speed Driver Host Mode Pipes Number.

This constant configures the maximum number of pipes that can be opened when the driver is operating in Host mode. Calling the

DRV_USBFS_HOST_PipeSetup function will cause a pipe to be opened. Calling this function when DRV_USBFS_HOST_PIPES_NUMBER number of pipes have already been opened will cause the function to return an Invalid Pipe Handle. This constant should be configured to account for the maximum number of devices and the device types to be supported by the host application.

For example if the USB Host application must support 2 USB Mass Storage devices and 1 CDC device, it must set this constant 9 (4 bulk pipes for 2 Mass Storage devices + 2 bulk pipes and 1 interrupt pipe for 1 CDC device and 2 control pipes for 2 devices). Allocating pipes consumes data memory.

Remarks

This constant should always be defined when DRV_USBFS_HOST_SUPPORT is set to true.

DRV_USBFS_HOST_RESET_DURATION Macro

Configures the time duration (in milliseconds) of the Reset Signal.

File

drv_usbfs_config_template.h

C

```
#define DRV_USBFS_HOST_RESET_DURATION 100
```

Description

USB Full Speed Driver Host Mode Reset Duration.

This constant configures the duration of the reset signal. The driver generates reset signal when the USB Host stack requests for root hub port reset. The driver will generate the reset signal for the duration specified by this constant and will then stop generating the reset signal.

Remarks

This constant should always be defined when DRV_USBFS_HOST_SUPPORT is set to true.

DRV_USBFS_HOST_SUPPORT Macro

Determines if the USB Host Functionality should be enabled.

File

drv_usbfs_config_template.h

C

```
#define DRV_USBFS_HOST_SUPPORT false
```

Description

USB Full Speed Driver Host Mode Support.

This constant should be set to true if USB Host mode support is required in the application. It should be set to false if host support is not required.

Remarks

This constant should always be defined.

DRV_USBFS_INSTANCES_NUMBER Macro

Specifies the number of driver instances to be enabled in the application.

File

drv_usbfs_config_template.h

C

```
#define DRV_USBFS_INSTANCES_NUMBER 1
```

Description

USB Full Speed Driver Instances Number.

This constant defines the number of driver instances to be enabled in the application. This will be typically be the number of USB controllers to be used in the application. On PIC32MX microcontrollers that have one USB controller, this value will always be 1. On PIC32MX microcontrollers which have 2 USB controllers, this value could 1 or 2, depending on whether 1 or 2 USB segments are required. To conserve data memory, this constant should be set to exactly the number of USB controller that are required in the system.

Remarks

This constant should always be defined.

DRV_USBFS_INTERRUPT_MODE Macro

Configures the driver for interrupt or polling mode operation.

File

drv_usbfs_config_template.h

C

```
#define DRV_USBFS_INTERRUPT_MODE true
```

Description

USB Full Speed Driver Interrupt Mode.

This constant configures the driver for interrupt or polling operation. If this flag is set to true, the driver will operate in interrupt mode. If the flag is set to false, the driver will operate in polled mode. In polled, the driver interrupt state machine gets updated in the SYS_Tasks(). If the driver is configured interrupt mode, the driver interrupt state machine gets updated in the driver interrupt service routine. It is always recommended for the driver to operate in interrupt mode.

Remarks

This constant should always be defined.

Building the Library

This section lists the files that are available in the PIC32MX USB Driver Library.

Description

This section list the files that are available in the `\src` folder of the PIC32MX USB Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/usb/usbfs`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_usbfs.h	This file should be included by any `.c` file which accesses the PIC32MX USB Driver API. This one file contains the prototypes for all driver API.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_usbfs.c	This file should always be included in the project when using the PIC3MX USB Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
/src/dynamic/drv_usbfs_device.c	This file should be included in the project if Device mode operation is required.
/src/dynamic/drv_usbfs_host.c	This file should be included in the project if Host mode operation is required.

Module Dependencies

The PIC32MX USB Driver Library depends on the following modules:

- Interrupt System Service Library

Library Interface

a) System Functions

	Name	Description
	DRV_USBFS_Status	Provides the current status of the USB Driver module.
	DRV_USBFS_Tasks	Maintains the driver's state machine when the driver is configured for Polled mode.
	DRV_USBFS_Tasks_ISR	Maintains the driver's Interrupt state machine and implements its ISR.

b) Client Core Functions

	Name	Description
	DRV_USBFS_ClientEventCallBackSet	This function sets up the event callback function that is invoked by the USB controller driver to notify the client of USB bus events.
	DRV_USBFS_Close	Closes an opened-instance of the USB Driver.
	DRV_USBFS_Initialize	Initializes the USB Driver.
	DRV_USBFS_Open	Opens the specified USB Driver instance and returns a handle to it.

c) Device Mode Operation Functions

	Name	Description
	DRV_USBFS_DEVICE_AddressSet	This function will set the USB module address that is obtained from the Host.
	DRV_USBFS_DEVICE_Attach	This function will enable the attach signaling resistors on the D+ and D- lines thus letting the USB Host know that a device has been attached on the bus.
	DRV_USBFS_DEVICE_CurrentSpeedGet	This function returns the USB speed at which the device is operating.
	DRV_USBFS_DEVICE_Detach	This function will disable the attach signaling resistors on the D+ and D- lines thus letting the USB Host know that the device has detached from the bus.
	DRV_USBFS_DEVICE_EndpointDisable	This function disables an endpoint.
	DRV_USBFS_DEVICE_EndpointDisableAll	This function disables all provisioned endpoints.
	DRV_USBFS_DEVICE_EndpointEnable	This function enables an endpoint for the specified direction and endpoint size.
	DRV_USBFS_DEVICE_EndpointIsEnabled	This function returns the enable/disable status of the specified endpoint and direction.
	DRV_USBFS_DEVICE_EndpointIsStalled	This function returns the stall status of the specified endpoint and direction.
	DRV_USBFS_DEVICE_EndpointStall	This function stalls an endpoint in the specified direction.
	DRV_USBFS_DEVICE_EndpointStallClear	This function clears the stall on an endpoint in the specified direction.
	DRV_USBFS_DEVICE_IRPCancel	This function cancels the specific IRP that are queued and in progress at the specified endpoint.
	DRV_USBFS_DEVICE_IRPCancelAll	This function cancels all IRPs that are queued and in progress at the specified endpoint.
	DRV_USBFS_DEVICE_IRPSubmit	This function submits an I/O Request Packet (IRP) for processing to the Hi-Speed USB Driver.
	DRV_USBFS_DEVICE_RemoteWakeupStart	This function causes the device to start Remote Wakeup Signalling on the bus.
	DRV_USBFS_DEVICE_RemoteWakeupStop	This function causes the device to stop the Remote Wakeup Signalling on the bus.
	DRV_USBFS_DEVICE_SOFNumberGet	This function will return the USB SOF packet number.

d) Host Mode Operation Functions

	Name	Description
	DRV_USBFS_HOST_EventsDisable	Disables Host mode events.
	DRV_USBFS_HOST_EventsEnable	Restores the events to the specified the original value.
	DRV_USBFS_HOST_IRPCancel	Cancels the specified IRP.
	DRV_USBFS_HOST_IRPSubmit	Submits an IRP on a pipe.
	DRV_USBFS_HOST_PipeClose	Closes an open pipe.
	DRV_USBFS_HOST_PipeSetup	Open a pipe with the specified attributes.

e) Root Hub Functions

	Name	Description
	DRV_USBFS_HOST_ROOT_HUB_BusSpeedGet	This function returns the operating speed of the bus to which this root hub is connected.
	DRV_USBFS_HOST_ROOT_HUB_Initialize	This function initializes the root hub driver.

	DRV_USBFS_HOST_ROOT_HUB_MaximumCurrentGet	Returns the maximum amount of current that this root hub can provide on the bus.
	DRV_USBFS_HOST_ROOT_HUB_OperationEnable	This function enables or disables root hub operation.
	DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled	Returns the operation enabled status of the root hub.
	DRV_USBFS_HOST_ROOT_HUB_PortNumbersGet	Returns the number of ports this root hub contains.
	DRV_USBFS_HOST_ROOT_HUB_PortReset	Resets the specified root hub port.
	DRV_USBFS_HOST_ROOT_HUB_PortResetIsComplete	Returns true if the root hub has completed the port reset operation.
	DRV_USBFS_HOST_ROOT_HUB_PortResume	Resumes the specified root hub port.
	DRV_USBFS_HOST_ROOT_HUB_PortSpeedGet	Returns the speed of at which the port is operating.
	DRV_USBFS_HOST_ROOT_HUB_PortSuspend	Suspends the specified root hub port.

f) Data Types and Constants

	Name	Description
	DRV_USBFS_EVENT	Identifies the different events that the USB Driver provides.
	DRV_USBFS_EVENT_CALLBACK	Type of the USB Driver event callback function.
	DRV_USBFS_HOST_PIPE_HANDLE	Defines the USB Driver Host Pipe Handle type.
	DRV_USBFS_INIT	This type definition defines the Driver Initialization Data Structure.
	DRV_USBFS_OPMODES	Identifies the operating modes supported by the USB Driver.
	DRV_USBFS_ROOT_HUB_PORT_INDICATION	USB Root hub Application Hooks (Port Indication).
	DRV_USBFS_ROOT_HUB_PORT_OVER_CURRENT_DETECT	USB Root hub Application Hooks (Port Overcurrent detection).
	DRV_USBFS_ROOT_HUB_PORT_POWER_ENABLE	USB Root hub Application Hooks (Port Power Enable/ Disable).
	DRV_USBFS_DEVICE_INTERFACE	USB Driver Device Mode Interface Functions.
	DRV_USBFS_ENDPOINT_TABLE_ENTRY_SIZE	USB Driver Endpoint Table Entry Size in bytes.
	DRV_USBFS_HOST_INTERFACE	USB Driver Host Mode Interface Functions.
	DRV_USBFS_HOST_PIPE_HANDLE_INVALID	Value of an Invalid Host Pipe Handle.
	DRV_USBFS_INDEX_0	USB Driver Module Index 0 Definition.
	DRV_USBFS_INDEX_1	USB Driver Module Index 1 Definition.

Description

This section describes the functions of the PIC32MX USB Driver Library.

Refer to each section for a detailed description.

a) System Functions

DRV_USBFS_Status Function

Provides the current status of the USB Driver module.

File

drv_usbfs.h

C

```
SYS_STATUS DRV_USBFS_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that the driver is ready.
- SYS_STATUS_UNINITIALIZED - Indicates that the driver has never been initialized.

Description

This function provides the current status of the USB Driver module.

Remarks

None.

Preconditions

The DRV_USBFS_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_USBFS_Initialize
SYS_STATUS          status;
DRV_USBFS_INIT moduleInit;

uint8_t __attribute__((aligned(512))) endpointTable[DRV_USBFS_ENDPOINT_TABLE_ENTRY_SIZE * 2];

usbInitData.usbID               = USB_ID_1;
usbInitData.opMode              = DRV_USBFS_OPMODE_DEVICE;
usbInitData.stopInIdle          = false;
usbInitData.suspendInSleep      = false;
usbInitData.operationSpeed      = USB_SPEED_FULL;
usbInitData.interruptSource     = INT_SOURCE_USB;

usbInitData.sysModuleInit.powerState = SYS_MODULE_POWER_RUN_FULL ;

// This is how this data structure is passed to the initialize
// function.

DRV_USBFS_Initialize(DRV_USBFS_INDEX_0, (SYS_MODULE_INIT *) &usbInitData);

// The status of the driver can be checked.
status = DRV_USBFS_Status(object);
if(SYS_STATUS_READY == status)
{
    // Driver is ready to be opened.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_USBFS_Initialize function.

Function

SYS_STATUS DRV_USBFS_Status (SYS_MODULE_OBJ object)

DRV_USBFS_Tasks Function

Maintains the driver's state machine when the driver is configured for Polled mode.

File

drv_usbfs.h

C

```
void DRV_USBFS_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Maintains the driver's Polled state machine. This function should be called from the SYS_Tasks function.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks). This function will never block.

Preconditions

The DRV_USBFS_Initialize function must have been called for the specified USB Driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_USBFS_Initialize

while (true)
{
    DRV_USBFS_Tasks(object);
```

```
    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_USBFS_Initialize function).

Function

void DRV_USBFS_Tasks(SYS_MODULE_OBJ object)

DRV_USBFS_Tasks_ISR Function

Maintains the driver's Interrupt state machine and implements its ISR.

File

drv_usbfs.h

C

```
void DRV_USBFS_Tasks_ISR(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the driver's internal Interrupt state machine and implement its ISR for interrupt-driven implementations.

Remarks

This routine should be called from the USB interrupt service routine. In case of multiple USB modules, it should be ensured that the correct USB driver system module object is passed to this routine.

Preconditions

The DRV_USBFS_Initialize function must have been called for the specified USB Driver instance.

Example

```
SYS_MODULE_OBJ object;     // Returned from DRV_USBFS_Initialize

while (true)
{
    DRV_USBFS_Tasks_ISR (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_USBFS_Initialize).

Function

void DRV_USBFS_Tasks_ISR(SYS_MODULE_OBJ object)

b) Client Core Functions

DRV_USBFS_ClientEventCallBackSet Function

This function sets up the event callback function that is invoked by the USB controller driver to notify the client of USB bus events.

File

drv_usbfs.h

C

```
void DRV_USBFS_ClientEventCallBackSet(DRV_HANDLE handle, uintptr_t hReferenceData, DRV_USB_EVENT_CALLBACK
```

```
myEventCallBack);
```

Returns

None.

Description

This function sets up the event callback function that is invoked by the USB controller driver to notify the client of USB bus events. The callback is disabled by either not calling this function after the DRV_USBFS_Open function has been called or by setting the myEventCallBack argument as NULL. When the callback function is called, the hReferenceData argument is returned.

Remarks

Typical usage of the USB Driver requires a client to register a callback.

Preconditions

None.

Example

```
// Set the client event callback for the Device Layer.  The
// USBDeviceLayerEventHandler function is the event handler. When this
// event handler is invoked by the driver, the driver returns back the
// second argument specified in the following function (which in this case
// is the Device Layer data structure). This allows the application
// firmware to identify, as an example, the Device Layer object associated
// with this callback.

DRV_USBFS_ClientEventCallBackSet(myUSBDevice.usbDriverHandle, (uintptr_t)&myUSBDevice,
USBDeviceLayerEventHandler);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).
hReferenceData	Object (could be a pointer) that is returned with the callback.
myEventCallBack	Callback function for all USB events.

Function

void DRV_USBFS_ClientEventCallBackSet

(

DRV_HANDLE handle,

uintptr_t hReferenceData,

DRV_USBFS_EVENT_CALLBACK myEventCallBack

);

DRV_USBFS_Close Function

Closes an opened-instance of the USB Driver.

File

drv_usbfs.h

C

```
void DRV_USBFS_Close(DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened-instance of the USB Driver, invalidating the handle.

Remarks

After calling this function, the handle passed in handle parameter must not be used with any of the other driver functions. A new handle must be obtained by calling DRV_USBFS_Open function before the caller may use the driver again.

Preconditions

The DRV_USBFS_Initialize function must have been called for the specified USB Driver instance. DRV_USBFS_Open function must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_USBFS_Open

DRV_USBFS_Close(handle);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).

Function

void DRV_USBFS_Close(DRV_HANDLE handle)

DRV_USBFS_Initialize Function

Initializes the USB Driver.

File

drv_usbfs.h

C

```
SYS_MODULE_OBJ DRV_USBFS_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init);
```

Returns

- SYS_MODULE_OBJ_INVALID - The driver initialization failed.
- A valid System Module Object - The driver initialization was able to start. It may have not completed and requires the DRV_USBFS_Tasks function to be called periodically. This value will never be the same as SYS_MODULE_OBJ_INVALID.

Description

This function initializes the USB Driver, making it ready for clients to open. The driver initialization does not complete when this function returns. The DRV_USBFS_Tasks function must called periodically to complete the driver initialization. The DRV_USBHS_Open function will fail if the driver was not initialized or if initialization has not completed.

Remarks

This routine must be called before any other USB driver routine is called. This routine should only be called once during system initialization unless DRV_USBFS_Deinitialize is called to deinitialize the driver instance.

Preconditions

None.

Example

```
 // The following code shows an example initialization of the
 // driver. The USB module to be used is USB1.  The module should not
 // automatically suspend when the  microcontroller enters Sleep mode.  The
 // module should continue  operation when the CPU enters Idle mode.  The
 // power state is set to run at full clock speeds. Device Mode operation
 // should be at FULL speed. The size of the endpoint table is set for 2
 // endpoints.

DRV_USBFS_INIT moduleInit;

uint8_t __attribute__((aligned(512))) endpointTable[DRV_USBFS_ENDPOINT_TABLE_ENTRY_SIZE * 2];

usbInitData.usbID               = USB_ID_1;
usbInitData.opMode              = DRV_USBFS_OPMODE_DEVICE;
usbInitData.stopInIdle          = false;
usbInitData.suspendInSleep      = false;
usbInitData.operationSpeed      = USB_SPEED_FULL;
usbInitData.interruptSource     = INT_SOURCE_USB;

usbInitData.sysModuleInit.powerState = SYS_MODULE_POWER_RUN_FULL ;
```

```
// This is how this data structure is passed to the initialize
// function.

DRV_USBFS_Initialize(DRV_USBFS_INDEX_0, (SYS_MODULE_INIT *) &usbInitData);
```

Parameters

Parameters	Description
drvIndex	Ordinal number of driver instance to be initialized. This should be set to DRV_USBFS_INDEX_0 if driver instance 0 needs to be initialized.
init	Pointer to a data structure containing data necessary to initialize the driver. This should be a DRV_USBFS_INIT structure reference typecast to SYS_MODULE_INIT reference.

Function

SYS_MODULE_OBJ DRV_USBHS_Initialize

(

const SYS_MODULE_INDEX drvIndex,

const SYS_MODULE_INIT * const init

)

DRV_USBFS_Open Function

Opens the specified USB Driver instance and returns a handle to it.

File

drv_usbfs.h

C

```
DRV_HANDLE DRV_USBFS_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

- DRV_HANDLE_INVALID - The driver could not be opened successfully.This can happen if the driver initialization was not complete or if an internal error has occurred.
- A Valid Driver Handle - This is an arbitrary value and is returned if the function was successful. This value will never be the same as DRV_HANDLE_INVALID.

Description

This function opens the specified USB Driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The intent flag should always be DRV_IO_INTENT_EXCLUSIVE|DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_NON_BLOCKING. Any other setting of the intent flag will return a invalid driver handle. A driver instance can only support one client. Trying to open a driver that has an existing client will result in an unsuccessful function call.

Remarks

The handle returned is valid until the DRV_USBFS_Close function is called. The function will typically return DRV_HANDLE_INVALID if the driver was not initialized. In such a case the client should try to open the driver again.

Preconditions

Function DRV_USBFS_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

// This code assumes that the driver has been initialized.
handle = DRV_USBFS_Open(DRV_USBFS_INDEX_0, DRV_IO_INTENT_EXCLUSIVE| DRV_IO_INTENT_READWRITE|
DRV_IO_INTENT_NON_BLOCKING);

if(DRV_HANDLE_INVALID == handle)
{
    // The application should try opening the driver again.
}
```

Parameters

Parameters	Description
drvIndex	Identifies the driver instance to be opened. As an example, this value can be set to DRV_USBFS_INDEX_0 if instance 0 of the driver has to be opened.
intent	Should always be (DRV_IO_INTENT_EXCLUSIVE|DRV_IO_INTENT_READWRITE| DRV_IO_INTENT_NON_BLOCKING).

Function

```
DRV_HANDLE DRV_USBFS_Open
(
const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT intent
)
```

c) Device Mode Operation Functions

DRV_USBFS_DEVICE_AddressSet Function

This function will set the USB module address that is obtained from the Host.

File

drv_usbfs.h

C

```
void DRV_USBFS_DEVICE_AddressSet(DRV_HANDLE handle, uint8_t address);
```

Returns

None.

Description

This function will set the USB module address that is obtained from the Host in a setup transaction. The address is obtained from the SET_ADDRESS command issued by the Host. The primary (first) client of the driver uses this function to set the module's USB address after decoding the setup transaction from the Host.

Remarks

None.

Preconditions

None.

Example

```
// This function should be called by the first client of the driver,
// which is typically the Device Layer. The address to set is obtained
// from the Host during enumeration.

DRV_USBFS_DEVICE_AddressSet(deviceLayer, 4);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).
address	The address of this module on the USB bus.

Function

void DRV_USBFS_DEVICE_AddressSet(DRV_HANDLE handle, uint8_t address);

DRV_USBFS_DEVICE_Attach Function

This function will enable the attach signaling resistors on the D+ and D- lines thus letting the USB Host know that a device has been attached on

the bus.

File

drv_usbfs.h

C

```
void DRV_USBFS_DEVICE_Attach(DRV_HANDLE handle);
```

Returns

None.

Description

This function enables the pull-up resistors on the D+ or D- lines thus letting the USB Host know that a device has been attached on the bus . This function should be called when the driver client is ready to receive communication from the Host (typically after all initialization is complete). The USB 2.0 specification requires VBUS to be detected before the data line pull-ups are enabled. The application must ensure the same.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// Open the device driver and attach the device to the USB.
handle = DRV_USBFS_Open(DRV_USBFS_INDEX_0, DRV_IO_INTENT_EXCLUSIVE| DRV_IO_INTENT_READWRITE|
DRV_IO_INTENT_NON_BLOCKING);

// Register a callback
DRV_USBFS_ClientEventCallBackSet(handle, (uintptr_t)&myDeviceLayer, MyDeviceLayerEventCallback);

// The device can be attached when VBUS Session Valid event occurs
void MyDeviceLayerEventCallback(uintptr_t handle, DRV_USBFS_EVENT event, void * hReferenceData)
{
    switch(event)
    {
        case DRV_USBFS_EVENT_DEVICE_SESSION_VALID:
            // A valid VBUS was detected.
            DRV_USBFS_DEVICE_Attach(handle);
            break;

        case DRV_USBFS_EVENT_DEVICE_SESSION_INVALID:
            // VBUS is not valid anymore. The device can be disconnected.
            DRV_USBFS_DEVICE_Detach(handle);
            break;

        default:
            break;
        }
    }
}
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).

Function

void DRV_USBFS_DEVICE_Attach(DRV_HANDLE handle);

DRV_USBFS_DEVICE_CurrentSpeedGet Function

This function returns the USB speed at which the device is operating.

File

drv_usbfs.h

C

```
USB_SPEED DRV_USBFS_DEVICE_CurrentSpeedGet(DRV_HANDLE handle);
```

Returns

- USB_SPEED_ERROR - The device speed is not valid.
- USB_SPEED_FULL - The device is operating at Full speed.

Description

This function returns the USB speed at which the device is operating.

Remarks

None.

Preconditions

Only valid after the device is attached to the Host and Host has completed reset signaling.

Example

```
// Get the current speed.

USB_SPEED deviceSpeed;

deviceSpeed = DRV_USBFS_DEVICE_CurrentSpeedGet(deviceLayer);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).

Function

USB_SPEED DRV_USBFS_DEVICE_CurrentSpeedGet(DRV_HANDLE handle);

DRV_USBFS_DEVICE_Detach Function

This function will disable the attach signaling resistors on the D+ and D- lines thus letting the USB Host know that the device has detached from the bus.

File

drv_usbfs.h

C

```
void DRV_USBFS_DEVICE_Detach(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables the pull-up resistors on the D+ or D- lines. This function should be called when the application wants to disconnect the device from the bus (typically to implement a soft detach or switch to Host mode operation). A self-powered device should be detached from the bus when the VBUS is not valid.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// Open the device driver and attach the device to the USB.
handle = DRV_USBFS_Open(DRV_USBFS_INDEX_0, DRV_IO_INTENT_EXCLUSIVE| DRV_IO_INTENT_READWRITE|
DRV_IO_INTENT_NON_BLOCKING);

// Register a callback
DRV_USBFS_ClientEventCallBackSet(handle, (uintptr_t)&myDeviceLayer, MyDeviceLayerEventCallback);
```

```
// The device can be detached when VBUS Session Invalid event occurs
void MyDeviceLayerEventCallback(uintptr_t handle, DRV_USBFS_EVENT event, void * hReferenceData)
{
    switch(event)
    {
        case DRV_USBFS_EVENT_DEVICE_SESSION_VALID:
            // A valid VBUS was detected.
            DRV_USBFS_DEVICE_Attach(handle);
            break;

        case DRV_USBFS_EVENT_DEVICE_SESSION_INVALID:
            // VBUS is not valid anymore. The device can be disconnected.
            DRV_USBFS_DEVICE_Detach(handle);
            break;

        default:
            break;
        }
    }
}
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).

Function

void DRV_USBFS_DEVICE_Detach(DRV_HANDLE handle);

DRV_USBFS_DEVICE_EndpointDisable Function

This function disables an endpoint.

File

drv_usbfs.h

C

```
USB_ERROR DRV_USBFS_DEVICE_EndpointDisable(DRV_HANDLE handle, USB_ENDPOINT endpointAndDirection);
```

Returns

- USB_ERROR_NONE - The endpoint was successfully enabled.
- USB_ERROR_DEVICE_ENDPOINT_INVALID - The endpoint that is being accessed is not a valid endpoint (endpoint was not provisioned through the DRV_USBFS_ENDPOINTS_NUMBER configuration constant) defined for this driver instance.

Description

This function disables an endpoint. If the endpoint type is a control endpoint type, both directions are disabled. For non-control endpoints, the function disables the specified direction only. The direction to be disabled is specified by the Most Significant Bit (MSB) of the endpointAndDirection parameter.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to disable
// a control endpoint. Note that the direction parameter is ignored.
// For a control endpoint, both the directions are disabled.

USB_ENDPOINT ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 0);

DRV_USBFS_DEVICE_EndpointDisable(handle, ep );
```

```
// This code shows an example of how to disable a BULK IN
// endpoint

USB_ENDPOINT ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

DRV_USBFS_DEVICE_EndpointDisable(handle, ep );
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).
endpointAndDirection	Specifies the endpoint and direction.

Function

USB_ERROR DRV_USBFS_DEVICE_EndpointDisable
(
DRV_HANDLE handle,
USB_ENDPOINT endpointAndDirection
)

DRV_USBFS_DEVICE_EndpointDisableAll Function

This function disables all provisioned endpoints.

File

drv_usbfs.h

C

```
USB_ERROR DRV_USBFS_DEVICE_EndpointDisableAll(DRV_HANDLE handle);
```

Returns

- USB_ERROR_NONE - The function exited successfully.
- USB_ERROR_PARAMETER_INVALID - The driver handle is invalid.

Description

This function disables all provisioned endpoints in both directions.

Remarks

This function is typically called by the USB Device Layer to disable all endpoints upon detecting a bus reset.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to disable all endpoints.

DRV_USBFS_DEVICE_EndpointDisableAll(handle);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).

Function

USB_ERROR DRV_USBFS_DEVICE_EndpointDisableAll(DRV_HANDLE handle)

DRV_USBFS_DEVICE_EndpointEnable Function

This function enables an endpoint for the specified direction and endpoint size.

File

drv_usbfs.h

C

```
USB_ERROR DRV_USBFS_DEVICE_EndpointEnable(DRV_HANDLE handle, USB_ENDPOINT endpointAndDirection,
USB_TRANSFER_TYPE transferType, uint16_t endpointSize);
```

Returns

- USB_ERROR_NONE - The endpoint was successfully enabled.
- USB_ERROR_DEVICE_ENDPOINT_INVALID - If the endpoint that is being accessed is not a valid endpoint defined for this driver instance. The value of DRV_USBFS_ENDPOINTS_NUMBER configuration constant should be adjusted.
- USB_ERROR_PARAMETER_INVALID - The driver handle is invalid.

Description

This function enables an endpoint for the specified direction and endpoint size. The function will enable the endpoint for communication in one direction at a time. It must be called twice if the endpoint is required to communicate in both the directions, with the exception of control endpoints. If the endpoint type is a control endpoint, the endpoint is always bidirectional and the function needs to be called only once.

The size of the endpoint must match the wMaxPacketSize reported in the endpoint descriptor for this endpoint. A transfer that is scheduled over this endpoint will be scheduled in wMaxPacketSize transactions. The function does not check if the endpoint is already in use. It is the client's responsibility to make sure that a endpoint is not accidentally reused.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to enable Endpoint
// 0 for control transfers. Note that for a control endpoint, the
// direction parameter is ignored. A control endpoint is always
// bidirectional. Endpoint size is 64 bytes.

uint8_t ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 0);

DRV_USBFS_DEVICE_EndpointEnable(handle, ep, USB_TRANSFER_TYPE_CONTROL, 64);

// This code shows an example of how to set up a endpoint
// for BULK IN transfer. For an IN transfer, data moves from device
// to Host. In this example, Endpoint 1 is enabled. The maximum
// packet size is 64.

uint8_t ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

DRV_USBFS_DEVICE_EndpointEnable(handle, ep, USB_TRANSFER_TYPE_BULK, 64);

// If Endpoint 1 must also be set up for BULK OUT, the
// DRV_USBFS_DEVICE_EndpointEnable function must be called again, as shown
// here.

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_HOST_TO_DEVICE, 1);

DRV_USBFS_DEVICE_EndpointEnable(handle, ep, USB_TRANSFER_TYPE_BULK, 64);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).
endpointAndDirection	Specifies the endpoint and direction.
transferType	Should be USB_TRANSFER_TYPE_CONTROL for control endpoint, USB_TRANSFER_TYPE_BULK for bulk endpoint, USB_TRANSFER_TYPE_INTERRUPT for interrupt endpoint and USB_TRANSFER_TYPE_ISOCHRONOUS for isochronous endpoint.
endpointSize	Maximum size (in bytes) of the endpoint as reported in the endpoint descriptor.

Function

```
USB_ERROR DRV_USBFS_DEVICE_EndpointEnable
(
    DRV_HANDLE handle,
USB_ENDPOINT endpointAndDirection,
USB_TRANSFER_TYPE transferType,
uint16_t endpointSize
);
```

DRV_USBFS_DEVICE_EndpointIsEnabled Function

This function returns the enable/disable status of the specified endpoint and direction.

File

drv_usbfs.h

C

```
bool DRV_USBFS_DEVICE_EndpointIsEnabled(DRV_HANDLE client, USB_ENDPOINT endpointAndDirection);
```

Returns

- true - The endpoint is enabled.
- false - The endpoint is disabled.

Description

This function returns the enable/disable status of the specified endpoint and direction.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how the
// DRV_USBFS_DEVICE_EndpointIsEnabled function can be used to obtain the
// status of Endpoint 1 and IN direction.

USB_ENDPOINT ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

if(DRV_USBFS_ENDPOINT_STATE_DISABLED ==
            DRV_USBFS_DEVICE_EndpointIsEnabled(handle, ep))
{
    // Endpoint is disabled. Enable endpoint.

    DRV_USBFS_DEVICE_EndpointEnable(handle, ep, USB_ENDPOINT_TYPE_BULK, 64);

}
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).
endpointAndDirection	Specifies the endpoint and direction.

Function

bool DRV_USBFS_DEVICE_EndpointIsEnabled
(
DRV_HANDLE handle,
USB_ENDPOINT endpointAndDirection
)

DRV_USBFS_DEVICE_EndpointIsStalled Function

This function returns the stall status of the specified endpoint and direction.

File

drv_usbfs.h

C

```
bool DRV_USBFS_DEVICE_EndpointIsStalled(DRV_HANDLE client, USB_ENDPOINT endpoint);
```

Returns

- true - The endpoint is stalled.
- false - The endpoint is not stalled.

Description

This function returns the stall status of the specified endpoint and direction.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how the
// DRV_USBFS_DEVICE_EndpointIsStalled function can be used to obtain the
// stall status of Endpoint 1 and IN direction.

USB_ENDPOINT ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

if(true == DRV_USBFS_DEVICE_EndpointIsStalled (handle, ep))
{
    // Endpoint stall is enabled. Clear the stall.

    DRV_USBFS_DEVICE_EndpointStallClear(handle, ep);

}
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).
endpointAndDirection	Specifies the endpoint and direction.

Function

bool DRV_USBFS_DEVICE_EndpointIsStalled
(

DRV_HANDLE handle,
USB_ENDPOINT endpointAndDirection
)

DRV_USBFS_DEVICE_EndpointStall Function

This function stalls an endpoint in the specified direction.

File

drv_usbfs.h

C

```
USB_ERROR DRV_USBFS_DEVICE_EndpointStall(DRV_HANDLE handle, USB_ENDPOINT endpointAndDirection);
```

Returns

- USB_ERROR_NONE - The endpoint was successfully enabled.
- USB_ERROR_PARAMETER_INVALID - The driver handle is not valid.
- USB_ERROR_DEVICE_ENDPOINT_INVALID - If the endpoint that is being accessed is out of the valid endpoint defined for this driver instance.
- USB_ERROR_OSAL_FUNCTION - An error with an OSAL function called in this function.

Description

This function stalls an endpoint in the specified direction.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to stall an endpoint. In
// this example, Endpoint 1 IN direction is stalled.

USB_ENDPOINT ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

DRV_USBFS_DEVICE_EndpointStall(handle, ep);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).
endpointAndDirection	Specifies the endpoint and direction.

Function

USB_ERROR DRV_USBFS_DEVICE_EndpointStall
(
DRV_HANDLE handle,
USB_ENDPOINT endpointAndDirection
)

DRV_USBFS_DEVICE_EndpointStallClear Function

This function clears the stall on an endpoint in the specified direction.

File

drv_usbfs.h

C

```
USB_ERROR DRV_USBFS_DEVICE_EndpointStallClear(DRV_HANDLE handle, USB_ENDPOINT endpointAndDirection);
```

Returns

- USB_ERROR_NONE - The endpoint was successfully enabled.
- USB_ERROR_PARAMETER_INVALID - The driver handle is not valid.
- USB_ERROR_DEVICE_ENDPOINT_INVALID - If the endpoint that is being accessed is out of the valid endpoint defined for this driver instance.

Description

This function clears the stall on an endpoint in the specified direction.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to clear a stall. In this
// example, the stall condition on Endpoint 1 IN direction is cleared.

USB_ENDPOINT ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

DRV_USBFS_DEVICE_EndpointStallClear(handle, ep);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).
endpointAndDirection	Specifies the endpoint and direction.

Function

USB_ERROR DRV_USBFS_DEVICE_EndpointStallClear

(

DRV_HANDLE handle,

USB_ENDPOINT endpointAndDirection

)

DRV_USBFS_DEVICE_IRPCancel Function

This function cancels the specific IRP that are queued and in progress at the specified endpoint.

File

drv_usbfs.h

C

```
USB_ERROR DRV_USBFS_DEVICE_IRPCancel(DRV_HANDLE client, USB_DEVICE_IRP * irp);
```

Returns

- USB_ERROR_NONE - The IRP have been canceled successfully.
- USB_ERROR_PARAMETER_INVALID - Invalid parameter or the IRP already has been aborted or completed
- USB_ERROR_OSAL_FUNCTION - An OSAL function called in this function did not execute successfully.

Description

This function attempts to cancel the processing of a queued IRP. An IRP that was in the queue but yet to be processed will be cancelled successfully and the IRP callback function will be called from this function with the USB_DEVICE_IRP_STATUS_ABORTED status. The application can release the data buffer memory used by the IRP when this callback occurs. If the IRP was in progress (a transaction in on the bus) when the cancel function was called, the IRP will be canceled only when an ongoing or the next transaction has completed. The IRP callback

function will then be called in an interrupt context. The application should not release the related data buffer unless the IRP callback has occurred.

Remarks

The size returned after the ABORT callback will be always 0 regardless of the amount of data that has been sent or received. The client should not assume any data transaction has happened for an canceled IRP. If the last transaction of the IRP was in progress, the IRP cancel does not have any effect. The first transaction of any ongoing IRP cannot be canceled.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to cancel IRP.  In this example the IRP
// has been scheduled from a device to the Host.

USB_ENDPOINT ep;
USB_DEVICE_IRP irp;

ep.direction = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

irp.data = myDataBufferToSend;
irp.size = 130;
irp.flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
irp.callback = MyIRPCompletionCallback;
irp.referenceData = (uintptr_t)&myDeviceLayerObj;

if (DRV_USBFS_DEVICE_IRPSubmit(handle, ep, &irp) != USB_ERROR_NONE)
{
    // This means there was an error.
}
else
{
    // Check the status of the IRP.
    if(irp.status != USB_DEVICE_IRP_STATUS_COMPLETED)
    {
        // Cancel the submitted IRP.
        if (DRV_USBFS_DEVICE_IRPCancel(handle, &irp) != USB_ERROR_NONE)
        {
            // The IRP Cancel request submission was successful.
            // IRP cancel status will be notified through the callback
            // function.
        }
        else
        {
            // The IRP may have been completed before IRP cancel operation.
            // could start. No callback notification will be generated.
        }
    }
    else
    {
        // The IRP processing must have been completed before IRP cancel was
        // submitted.
    }
}

void MyIRPCallback(USB_DEVICE_IRP * irp)
{
    // Check if the IRP callback is for a Cancel request
    if(irp->status == USB_DEVICE_IRP_STATUS_ABORTED)
    {
        // IRP cancel completed
    }
 }
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).
irp	Pointer to the IRP to cancel.

Function

USB_ERROR DRV_USBFS_DEVICE_IRPCancel
(
DRV_HANDLE client,
USB_DEVICE_IRP * irp
)

DRV_USBFS_DEVICE_IRPCancelAll Function

This function cancels all IRPs that are queued and in progress at the specified endpoint.

File

drv_usbfs.h

C

```
USB_ERROR DRV_USBFS_DEVICE_IRPCancelAll(DRV_HANDLE client, USB_ENDPOINT endpointAndDirection);
```

Returns

- USB_ERROR_NONE - The endpoint was successfully enabled.
- USB_ERROR_DEVICE_ENDPOINT_INVALID - If the endpoint that is being accessed is out of the valid endpoint defined for this driver instance.
- USB_ERROR_PARAMETER_INVALID - The driver handle is not valid.
- USB_ERROR_OSAL_FUNCTION - An OSAL function called in this function did not execute successfully.

Description

This function cancels all IRPs that are queued and in progress at the specified endpoint.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to cancel all IRPs.

void MyIRPCallback(USB_DEVICE_IRP * irp)
{
    // Check if this is setup command

    if(irp->status == USB_DEVICE_IRP_STATUS_SETUP)
    {
        if(IsSetupCommandSupported(irp->data) == false)
        {
            // This means that this setup command is not
            // supported. Stall the some related endpoint and cancel all
            // queue IRPs.

            DRV_USBFS_DEVICE_EndpointStall(handle, ep);
            DRV_USBFS_DEVICE_IRPCancelAll(handle, ep);
        }
     }
 }
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).
endpointAndDirection	Specifies the endpoint and direction.

Function

USB_ERROR DRV_USBFS_DEVICE_IRPCancelAll

```
(
    DRV_HANDLE client,
USB_ENDPOINT endpointAndDirection
);
```

DRV_USBFS_DEVICE_IRPSubmit Function

This function submits an I/O Request Packet (IRP) for processing to the Hi-Speed USB Driver.

File

drv_usbfs.h

C

```
USB_ERROR DRV_USBFS_DEVICE_IRPSubmit(DRV_HANDLE client, USB_ENDPOINT endpointAndDirection, USB_DEVICE_IRP *
irp);
```

Returns

- USB_ERROR_NONE - if the IRP was submitted successful.
- USB_ERROR_IRP_SIZE_INVALID - if the size parameter of the IRP is not correct.
- USB_ERROR_PARAMETER_INVALID - If the client handle is not valid.
- USB_ERROR_ENDPOINT_NOT_CONFIGURED - If the endpoint is not enabled.
- USB_ERROR_DEVICE_ENDPOINT_INVALID - The specified endpoint is not valid.
- USB_ERROR_OSAL_FUNCTION - An OSAL call in the function did not complete successfully.

Description

This function submits an I/O Request Packet (IRP) for processing to the USB Driver. The IRP allows a client to send and receive data from the USB Host. The data will be sent or received through the specified endpoint. The direction of the data transfer is indicated by the direction flag in the endpointAndDirection parameter. Submitting an IRP arms the endpoint to either send data to or receive data from the Host. If an IRP is already being processed on the endpoint, the subsequent IRP submit operation will be queued. The contents of the IRP (including the application buffers) should not be changed until the IRP has been processed.

Particular attention should be paid to the size parameter of IRP. The following should be noted:

- The size parameter while sending data to the Host can be less than, greater than, equal to, or be an exact multiple of the maximum packet size for the endpoint. The maximum packet size for the endpoint determines the number of transactions required to process the IRP.
- If the size parameter, while sending data to the Host is less than the maximum packet size, the transfer will complete in one transaction.
- If the size parameter, while sending data to the Host is greater than the maximum packet size, the IRP will be processed in multiple transactions.
- If the size parameter, while sending data to the Host is equal to or an exact multiple of the maximum packet size, the client can optionally ask the driver to send a Zero Length Packet(ZLP) by specifying the USB_DEVICE_IRP_FLAG_DATA_COMPLETE flag as the flag parameter.
- The size parameter, while receiving data from the Host must be an exact multiple of the maximum packet size of the endpoint. If this is not the case, the driver will return a USB_ERROR_IRP_SIZE_INVALID result. If while processing the IRP, the driver receives less than maximum packet size or a ZLP from the Host, the driver considers the IRP as processed. The size parameter at this point contains the actual amount of data received from the Host. The IRP status is returned as USB_DEVICE_IRP_STATUS_COMPLETED_SHORT.
- If a ZLP needs to be sent to Host, the IRP size should be specified as 0 and the flag parameter should be set as USB_DEVICE_IRP_FLAG_DATA_COMPLETE.
- If the IRP size is an exact multiple of the endpoint size, the client can request the driver to not send a ZLP by setting the flag parameter to USB_DEVICE_IRP_FLAG_DATA_PENDING. This flag indicates that there is more data pending in this transfer.
- Specifying a size less than the endpoint size along with the USB_DEVICE_IRP_FLAG_DATA_PENDING flag will cause the driver to return a USB_ERROR_IRP_SIZE_INVALID.
- If the size is greater than but not a multiple of the endpoint size, and the flag is specified as USB_DEVICE_IRP_FLAG_DATA_PENDING, the driver will send multiple of endpoint size number of bytes. For example, if the IRP size is 130 and the endpoint size if 64, the number of bytes sent will 128.

Remarks

This function can be called from the ISR of the USB module to associated with the client.

Preconditions

The Client handle should be valid.

Example

```
// The following code shows an example of how to schedule a IRP to send data
// from a device to the Host. Assume that the max packet size is 64 and
```

```
// and this data needs to sent over Endpoint 1. In this example, the
// transfer is processed as three transactions of 64, 64 and 2 bytes.

USB_ENDPOINT ep;
USB_DEVICE_IRP irp;

ep.direction = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

irp.data = myDataBufferToSend;
irp.size = 130;
irp.flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
irp.callback = MyIRPCompletionCallback;
irp.referenceData = (uintptr_t)&myDeviceLayerObj;

if (DRV_USBFS_DEVICE_IRPSubmit(handle, ep, &irp) != USB_ERROR_NONE)
{
    // This means there was an error.
}
else
{
    // The status of the IRP can be checked.
    while(irp.status != USB_DEVICE_IRP_STATUS_COMPLETED)
    {
        // Wait or run a task function.
    }
}

// The following code shows how the client can request
// the driver to send a ZLP when the size is an exact multiple of
// endpoint size.

irp.data = myDataBufferToSend;
irp.size = 128;
irp.flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
irp.callback = MyIRPCompletionCallback;
irp.referenceData = (uintptr_t)&myDeviceLayerObj;

// Note that while  receiving data from the Host, the size should be an
// exact multiple of the maximum packet size of the endpoint. In the
// following example, the DRV_USBFS_DEVICE_IRPSubmit function will return a
// USB_DEVICE_IRP_SIZE_INVALID value.

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_HOST_TO_DEVICE, 1);

irp.data = myDataBufferToSend;
irp.size = 60; // THIS SIZE IS NOT CORRECT
irp.flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
irp.callback = MyIRPCompletionCallback;
irp.referenceData = (uintptr_t)&myDeviceLayerObj;
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).
endpointAndDirection	Specifies the endpoint and direction.
irp	Pointer to the IRP to be added to the queue for processing.

Function

USB_ERROR DRV_USBFS_DEVICE_IRPSubmit

(

DRV_HANDLE client,

USB_ENDPOINT endpointAndDirection,

USB_DEVICE_IRP * irp

);

DRV_USBFS_DEVICE_RemoteWakeupStart Function

This function causes the device to start Remote Wakeup Signalling on the bus.

File

drv_usbfs.h

C

```
void DRV_USBFS_DEVICE_RemoteWakeupStart(DRV_HANDLE handle);
```

Returns

None.

Description

This function causes the device to start Remote Wakeup Signalling on the bus. This function should be called when the device, presently placed in suspend mode by the Host, wants to be wakeup. Note that the device can do this only when the Host has enabled the device's Remote Wakeup capability.

Remarks

None.

Preconditions

The handle should be valid.

Example

```
DRV_HANDLE handle;

// If the Host has enabled the Remote Wakeup capability, and if the device
// is in suspend mode, then start Remote Wakeup signaling.

if(deviceIsSuspended && deviceRemoteWakeupEnabled)
{
    DRV_USBFS_DEVICE_RemoteWakeupStart(handle);
}
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBFS_Open function).

Function

void DRV_USBFS_DEVICE_RemoteWakeupStart(DRV_HANDLE handle);

DRV_USBFS_DEVICE_RemoteWakeupStop Function

This function causes the device to stop the Remote Wakeup Signalling on the bus.

File

drv_usbfs.h

C

```
void DRV_USBFS_DEVICE_RemoteWakeupStop(DRV_HANDLE handle);
```

Returns

None.

Description

This function causes the device to stop Remote Wakeup Signalling on the bus. This function should be called after the DRV_USBFS_DEVICE_RemoteWakeupStart function was called to start the Remote Wakeup signaling on the bus.

Remarks

This function should be 1 to 15 milliseconds after the DRV_USBFS_DEVICE_RemoteWakeupStart function was called.

Preconditions

The handle should be valid. The DRV_USBFS_DEVICE_RemoteWakeupStart function was called to start the Remote Wakeup signaling on the bus.

Example

```
DRV_HANDLE handle;

// If the Host has enabled the Remote Wakeup capability, and if the device
// is in suspend mode, then start Remote Wakeup signaling. Wait for 10
// milliseconds and then stop the Remote Wakeup signaling

if(deviceIsSuspended && deviceRemoteWakeupEnabled)
{
    DRV_USBFS_DEVICE_RemoteWakeupStart(handle);
    DelayMilliSeconds(10);
    DRV_USBFS_DEVICE_RemoteWakeupStop(handle);
}
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBFS_Open function).

Function

void DRV_USBFS_DEVICE_RemoteWakeupStop(DRV_HANDLE handle);

DRV_USBFS_DEVICE_SOFNumberGet Function

This function will return the USB SOF packet number.

File

drv_usbfs.h

C

```
uint16_t DRV_USBFS_DEVICE_SOFNumberGet(DRV_HANDLE handle);
```

Returns

The SOF packet number.

Description

This function will return the USB SOF packet number..

Remarks

None.

Preconditions

This function will return a valid value only when the device is attached to the bus. The SOF packet count will not increment if the bus is suspended.

Example

```
// This code shows how the DRV_USBFS_DEVICE_SOFNumberGet function is called
// to read the current SOF number.

DRV_HANDLE handle;
uint16_t sofNumber;

sofNumber = DRV_USBFS_DEVICE_SOFNumberGet(handle);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).

Function

uint16_t DRV_USBFS_DEVICE_SOFNumberGet(DRV_HANDLE handle);

d) Host Mode Operation Functions

DRV_USBFS_HOST_EventsDisable Function

Disables Host mode events.

File

drv_usbfs.h

C

```
bool DRV_USBFS_HOST_EventsDisable(DRV_HANDLE handle);
```

Returns

- true - Driver event generation was enabled when this function was called.
- false - Driver event generation was not enabled when this function was called.

Description

This function disables the Host mode events. This function is called by the Host Layer when it wants to execute code atomically.

Remarks

None.

Preconditions

The handle should be valid.

Example

```
// This code shows how the DRV_USBFS_HOST_EventsDisable and
// DRV_USBFS_HOST_EventsEnable function can be called to disable and enable
// events.

DRV_HANDLE driverHandle;
bool eventsWereEnabled;

// Disable the driver events.
eventsWereEnabled = DRV_USBFS_HOST_EventsDisable(driverHandle);

// Code in this region will not be interrupted by driver events.

// Enable the driver events.
DRV_USBFS_HOST_EventsEnable(driverHandle, eventsWereEnabled);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBFS_Open function).

Function

bool DRV_USBFS_HOST_EventsDisable

(

DRV_HANDLE handle

);

DRV_USBFS_HOST_EventsEnable Function

Restores the events to the specified the original value.

File

drv_usbfs.h

C

```
void DRV_USBFS_HOST_EventsEnable(DRV_HANDLE handle, bool eventContext);
```

Returns

None.

Description

This function will restore the enable disable state of the events. The eventRestoreContext parameter should be equal to the value returned by the DRV_USBFS_HOST_EventsDisable function.

Remarks

None.

Preconditions

The handle should be valid.

Example

```
// This code shows how the DRV_USBFS_HOST_EventsDisable and
// DRV_USBFS_HOST_EventsEnable function can be called to disable and enable
// events.

DRV_HANDLE driverHandle;
bool eventsWereEnabled;

// Disable the driver events.
eventsWereEnabled = DRV_USBFS_HOST_EventsDisable(driverHandle);

// Code in this region will not be interrupted by driver events.

// Enable the driver events.
DRV_USBFS_HOST_EventsEnable(driverHandle, eventsWereEnabled);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBFS_Open function).
eventRestoreContext	Value returned by the DRV_USBFS_HOST_EventsDisable function.

Function

void DRV_USBFS_HOST_EventsEnable

(

DRV_HANDLE handle

bool eventRestoreContext

);

DRV_USBFS_HOST_IRPCancel Function

Cancels the specified IRP.

File

drv_usbfs.h

C

```
void DRV_USBFS_HOST_IRPCancel(USB_HOST_IRP * inputIRP);
```

Returns

None.

Description

This function attempts to cancel the specified IRP. If the IRP is queued and its processing has not started, it will be cancelled successfully. If the IRP in progress, the ongoing transaction will be allowed to complete.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how a submitted IRP can be cancelled.

USB_HOST_IRP irp;
USB_ERROR result;
USB_HOST_PIPE_HANDLE controlPipe;
USB_SETUP_PACKET setup;
uint8_t controlTransferData[32];

irp.setup = setup;
irp.data = controlTransferData;
irp.size = 32;
irp.flags = USB_HOST_IRP_FLAG_NONE ;
irp.userData = &someApplicationObject;
irp.callback = IRP_Callback;

DRV_USBFS_HOST_IRPSubmit(controlPipeHandle, &irp);

// Additional application logic may come here. This logic may decide to
// cancel the submitted IRP.

DRV_USBFS_HOST_IRPCancel(&irp);
```

Parameters

Parameters	Description
inputIRP	Pointer to the IRP to cancel.

Function

void DRV_USBFS_HOST_IRPCancel(USB_HOST_IRP * inputIRP);

DRV_USBFS_HOST_IRPSubmit Function

Submits an IRP on a pipe.

File

drv_usbfs.h

C

```
USB_ERROR DRV_USBFS_HOST_IRPSubmit(DRV_USBFS_HOST_PIPE_HANDLE hPipe, USB_HOST_IRP * pinputIRP);
```

Returns

- USB_ERROR_NONE - The IRP was submitted successfully.
- USB_ERROR_PARAMETER_INVALID - The pipe handle is not valid.
- USB_ERROR_OSAL_FUNCTION - An error occurred in an OSAL function called in this function.

Description

This function submits an IRP on the specified pipe. The IRP will be added to the queue and will be processed in turn. The data will be transferred on the bus based on the USB bus scheduling rules. When the IRP has been processed, the callback function specified in the IRP will be called. The IRP status will be updated to reflect the completion status of the IRP.

Remarks

An IRP can also be submitted in an IRP callback function.

Preconditions

The pipe handle should be valid.

Example

```
// The following code shows an example of how the host layer populates
// the IRP object and then submits it. IRP_Callback function is called when an
// IRP has completed processing. The status of the IRP at completion can be
// checked in the status flag. The size field of the irp will contain the amount
// of data transferred.

void IRP_Callback(USB_HOST_IRP * irp)
{
    // irp is pointing to the IRP for which the callback has occurred. In most
    // cases this function will execute in an interrupt context. The application
    // should not perform any hardware access or interrupt un-safe operations in
    // this function.

    switch(irp->status)
    {
        case USB_HOST_IRP_STATUS_ERROR_UNKNOWN:
            // IRP was terminated due to an unknown error
            break;

        case USB_HOST_IRP_STATUS_ABORTED:
            // IRP was terminated by the application
            break;

        case USB_HOST_IRP_STATUS_ERROR_BUS:
            // IRP was terminated due to a bus error
            break;

        case USB_HOST_IRP_STATUS_ERROR_DATA:
            // IRP was terminated due to data error
            break;

        case USB_HOST_IRP_STATUS_ERROR_NAK_TIMEOUT:
            // IRP was terminated because of a NAK timeout
            break;

        case USB_HOST_IRP_STATUS_ERROR_STALL:
            // IRP was terminated because of a device sent a STALL
            break;

        case USB_HOST_IRP_STATUS_COMPLETED:
            // IRP has been completed
            break;

        case USB_HOST_IRP_STATUS_COMPLETED_SHORT:
            // IRP has been completed but the amount of data processed was less
            // than requested.
            break;

        default:
            break;
    }
}

// In the following code snippet the a control transfer IRP is submitted to a
// control pipe. The setup parameter of the IRP points to the Setup command of
// the control transfer. The direction of the data stage is specified by the
// Setup packet.

USB_HOST_IRP irp;
USB_ERROR result;
USB_HOST_PIPE_HANDLE controlPipe;
USB_SETUP_PACKET setup;
uint8_t controlTransferData[32];
```

```
irp.setup = setup;
irp.data = controlTransferData;
irp.size = 32;
irp.flags = USB_HOST_IRP_FLAG_NONE ;
irp.userData = &someApplicationObject;
irp.callback = IRP_Callback;

result = DRV_USBFS_HOST_IRPSubmit(controlPipeHandle, &irp);
```

Parameters

Parameters	Description
hPipe	Handle to the pipe to which the IRP has to be submitted.
pInputIRP	Pointer to the IRP.

Function

USB_ERROR DRV_USBFS_HOST_IRPSubmit

(

DRV_USBFS_HOST_PIPE_HANDLE hPipe,

USB_HOST_IRP * pInputIRP

);

DRV_USBFS_HOST_PipeClose Function

Closes an open pipe.

File

drv_usbfs.h

C

```
void DRV_USBFS_HOST_PipeClose(DRV_USBFS_HOST_PIPE_HANDLE pipeHandle);
```

Returns

None.

Description

This function closes an open pipe. Any IRPs scheduled on the pipe will be aborted and IRP callback functions will be called with the status as DRV_USB_HOST_IRP_STATE_ABORTED. The pipe handle will become invalid and the pipe will not accept IRPs.

Remarks

None.

Preconditions

The pipe handle should be valid.

Example

```
// This code shows how an open Host pipe can be closed.

DRV_HANDLE driverHandle;
DRV_USBFS_HOST_PIPE_HANDLE pipeHandle;

// Close the pipe.
DRV_USBFS_HOST_PipeClose(pipeHandle);
```

Parameters

Parameters	Description
pipeHandle	Handle to the pipe to close.

Function

void DRV_USBFS_HOST_PipeClose

(

```
    DRV_USBFS_HOST_PIPE_HANDLE pipeHandle
);
```

DRV_USBFS_HOST_PipeSetup Function

Open a pipe with the specified attributes.

File

drv_usbfs.h

C

```
DRV_USBFS_HOST_PIPE_HANDLE DRV_USBFS_HOST_PipeSetup(DRV_HANDLE client, uint8_t deviceAddress, USB_ENDPOINT
endpointAndDirection, uint8_t hubAddress, uint8_t hubPort, USB_TRANSFER_TYPE pipeType, uint8_t bInterval,
uint16_t wMaxPacketSize, USB_SPEED speed);
```

Returns

- DRV_USB_HOST_PIPE_HANDLE_INVALID - The pipe could not be created.
- A valid Pipe Handle - The pipe was created successfully. This is an arbitrary value and will never be the same as DRV_USB_HOST_PIPE_HANDLE_INVALID.

Description

This function opens a communication pipe between the Host and the device endpoint. The transfer type and other attributes are specified through the function parameters. The driver does not check for available bus bandwidth, which should be done by the application (the USB Host Layer in this case)

Remarks

None.

Preconditions

The driver handle should be valid.

Example

```
// This code shows how the DRV_USBFS_HOST_PipeSetup function is called for
// create a communication pipe. In this example, Bulk pipe is created
// between the Host and a device. The Device address is 2 and the target
// endpoint on this device is 4 . The direction of the data transfer over
// this pipe is from the Host to the device. The device is connected to Port
// 1 of a Hub, whose USB address is 3. The maximum size of a transaction
// on this pipe is 64 bytes. This is a Bulk Pipe and hence the bInterval
// field is set to 0. The target device is operating at Full Speed.

DRV_HANDLE driverHandle;
DRV_USBFS_HOST_PIPE_HANDLE pipeHandle;

pipeHandle = DRV_USBFS_HOST_PipeSetup(driverHandle, 0x02, 0x14, 0x03, 0x01, USB_TRANSFER_TYPE_BULK, 0, 64,
USB_SPEED_FULL);

if(pipeHandle != DRV_USBFS_HOST_PIPE_HANDLE_INVALID)
{
    // The pipe was created successfully.
}
```

Parameters

Parameters	Description
client	Handle to the driver (returned from DRV_USBFS_Open function).
deviceAddress	USB Address of the device to connect to.
endpoint	Endpoint on the device to connect to.
hubAddress	Address of the hub to which this device is connected. If not connected to a hub, this value should be set to 0.
hubPort	Port number of the hub to which this device is connected.
pipeType	Transfer type of the pipe to open.
bInterval	Polling interval for periodic transfers. This should be specified as defined by the USB 2.0 Specification.

wMaxPacketSize	This should be set to the endpoint size reported by the device in its configuration descriptors. This defines the maximum size of the transaction in a transfer on this pipe.
speed	The speed of the pipe. This should match the speed at which the device connected to the Host.

Function

```
DRV_USBFS_HOST_PIPE_HANDLE DRV_USBFS_HOST_PipeSetup
(
    DRV_HANDLE client,
uint8_t deviceAddress,
USB_ENDPOINT endpointAndDirection,
uint8_t hubAddress,
uint8_t hubPort,
USB_TRANSFER_TYPE pipeType,
uint8_t bInterval,
uint16_t wMaxPacketSize,
USB_SPEED speed
);
```

e) Root Hub Functions

DRV_USBFS_HOST_ROOT_HUB_BusSpeedGet Function

This function returns the operating speed of the bus to which this root hub is connected.

File

drv_usbfs.h

C

```
USB_SPEED DRV_USBFS_HOST_ROOT_HUB_BusSpeedGet(DRV_HANDLE handle);
```

Returns

- USB_SPEED_FULL - The Root hub is connected to a bus that is operating at Full Speed.

Description

This function returns the operating speed of the bus to which this root hub is connected.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how the DRV_USBFS_HOST_ROOT_HUB_BusSpeedGet function is
// called to know the operating speed of the bus to which this Root hub is
// connected.

DRV_HANDLE driverHandle;
USB_SPEED speed;

speed = DRV_USBFS_HOST_ROOT_HUB_BusSpeedGet(driverHandle);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBFS_Open function).

Function

USB_SPEED DRV_USBFS_HOST_ROOT_HUB_BusSpeedGet(DRV_HANDLE handle);

DRV_USBFS_HOST_ROOT_HUB_Initialize Function

This function initializes the root hub driver.

File

drv_usbfs.h

C

```
void DRV_USBFS_HOST_ROOT_HUB_Initialize(DRV_HANDLE handle, USB_HOST_DEVICE_OBJ_HANDLE usbHostDeviceInfo);
```

Returns

None.

Description

This function initializes the root hub driver. It is called by the Host Layer at the time of processing the root hub devices. The Host Layer assigns a USB_HOST_DEVICE_INFO reference to this root hub driver. This identifies the relationship between the root hub and the Host Layer.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how the USB Host Layer calls the
// DRV_USBFS_HOST_ROOT_HUB_Initialize function. The usbHostDeviceInfo
// parameter is an arbitrary identifier assigned by the USB Host Layer. Its
// interpretation is opaque to the Root hub Driver.

DRV_HANDLE drvHandle;
USB_HOST_DEVICE_OBJ_HANDLE usbHostDeviceInfo = 0x10003000;

DRV_USBFS_HOST_ROOT_HUB_Initialize(drvHandle, usbHostDeviceInfo);
```

Parameters

Parameters	Description
handle	Handle to the driver.
usbHostDeviceInfo	Reference provided by the Host.

Function

void DRV_USBFS_HOST_ROOT_HUB_Initialize

(

DRV_HANDLE handle,

USB_HOST_DEVICE_OBJ_HANDLE usbHostDeviceInfo,

)

DRV_USBFS_HOST_ROOT_HUB_MaximumCurrentGet Function

Returns the maximum amount of current that this root hub can provide on the bus.

File

drv_usbfs.h

C

```
uint32_t DRV_USBFS_HOST_ROOT_HUB_MaximumCurrentGet(DRV_HANDLE handle);
```

Returns

Returns the maximum current (in milliamperes) that the root hub can supply.

Description

This function returns the maximum amount of current that this root hub can provide on the bus.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how the DRV_USBFS_HOST_ROOT_HUB_MaximumCurrentGet
// function is called to obtain the maximum VBUS current that the Root hub
// can supply.

DRV_HANDLE driverHandle;
uint32_t currentMilliAmperes;

currentMilliAmperes = DRV_USBFS_HOST_ROOT_HUB_MaximumCurrentGet(driverHandle);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBFS_Open function).

Function

uint32_t DRV_USBFS_HOST_ROOT_HUB_MaximumCurrentGet(DRV_HANDLE);

DRV_USBFS_HOST_ROOT_HUB_OperationEnable Function

This function enables or disables root hub operation.

File

drv_usbfs.h

C

```
void DRV_USBFS_HOST_ROOT_HUB_OperationEnable(DRV_HANDLE handle, bool enable);
```

Returns

None.

Description

This function enables or disables root hub operation. When enabled, the root hub will detect devices attached to the port and will request the Host Layer to enumerate the device. This function is called by the Host Layer when it is ready to receive enumeration requests from the Host. If the operation is disabled, the root hub will not detect attached devices.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how the DRV_USBFS_HOST_ROOT_HUB_OperationEnable and the
// DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled functions are called to enable
// the Root hub operation.

DRV_HANDLE driverHandle;

// Enable Root hub operation.
```

```
DRV_USBFS_HOST_ROOT_HUB_OperationEnable(driverHandle);

// Wait till the Root hub operation is enabled.
if(DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled(driverHandle) == false)
{
    // The operation has not completed. Call the
    // DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled function again to check if
    // the operation has completed. Note that the DRV_USBFS_Tasks function
    // must be allowed to run at periodic intervals to allow the enable
    // operation to completed.
}
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBFS_Open function).
enable	If this is set to true, root hub operation is enabled. If this is set to false, root hub operation is disabled.

Function

void DRV_USBFS_HOST_ROOT_HUB_OperationEnable

(

DRV_HANDLE handle,

bool enable

);

DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled Function

Returns the operation enabled status of the root hub.

File

drv_usbfs.h

C

```
bool DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled(DRV_HANDLE handle);
```

Returns

- true - Root hub operation is enabled.
- false - Root hub operation is not enabled.

Description

This function returns true if the DRV_USBFS_HOST_ROOT_HUB_OperationEnable function has completed enabling the Host.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how the DRV_USBFS_HOST_ROOT_HUB_OperationEnable and the
// DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled functions are called to enable
// the Root hub operation.

DRV_HANDLE driverHandle;

// Enable Root hub operation.
DRV_USBFS_HOST_ROOT_HUB_OperationEnable(driverHandle);

// Wait till the Root hub operation is enabled.
if(DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled(driverHandle) == false)
{
    // The operation has not completed. Call the
    // DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled function again to check if
    // the operation has completed. Note that the DRV_USBFS_Tasks function
```

```
    // must be allowed to run at periodic intervals to allow the enable
    // operation to completed.
}
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBFS_Open function).

Function

bool DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled(DRV_HANDLE handle);

DRV_USBFS_HOST_ROOT_HUB_PortNumbersGet Function

Returns the number of ports this root hub contains.

File

drv_usbfs.h

C

```
uint8_t DRV_USBFS_HOST_ROOT_HUB_PortNumbersGet(DRV_HANDLE handle);
```

Returns

This function will always return 1.

Description

This function returns the number of ports that this root hub contains.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how DRV_USBFS_HOST_ROOT_HUB_PortNumbersGet function can
// be called to obtain the number of Root hub ports.

DRV_HANDLE driverHandle;
uint8_t nPorts;

nPorts = DRV_USBFS_HOST_ROOT_HUB_PortNumbersGet(driverHandle);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBFS_Open function).

Function

uint8_t DRV_USBFS_HOST_ROOT_HUB_PortNumbersGet(DRV_HANDLE handle);

DRV_USBFS_HOST_ROOT_HUB_PortReset Function

Resets the specified root hub port.

File

drv_usbfs.h

C

```
USB_ERROR DRV_USBFS_HOST_ROOT_HUB_PortReset(DRV_HANDLE handle, uint8_t port);
```

Returns

None.

Description

This function resets the root hub port. The reset duration is defined by DRV_USBFS_ROOT_HUB_RESET_DURATION. The status of the reset signaling can be checked using the DRV_USBFS_ROOT_HUB_PortResetIsComplete function.

Remarks

The root hub on the PIC32MZ USB controller contains only one port - port 0.

Preconditions

None.

Example

```
// This code shows how the DRV_USB_HOST_ROOT_HUB_PortReset and the
// DRV_USBFS_ROOT_HUB_PortResetIsComplete functions are called to complete a
// port reset sequence.

DRV_HANDLE driverHandle;

// Reset Port 0.
DRV_USB_HOST_ROOT_HUB_PortReset(driverHandle, 0);

// Check if the Reset operation has completed.
if(DRV_USBFS_ROOT_HUB_PortResetIsComplete(driverHandle, 0) == false)
{
    // This means that the Port Reset operation has not completed yet. The
    // DRV_USBFS_ROOT_HUB_PortResetIsComplete function should be called
    // again after some time to check the status.
}
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBFS_Open function).
port	Port to reset.

Function

void DRV_USBFS_ROOT_HUB_PortReset(DRV_HANDLE handle, uint8_t port);

DRV_USBFS_HOST_ROOT_HUB_PortResetIsComplete Function

Returns true if the root hub has completed the port reset operation.

File

drv_usbfs.h

C

```
bool DRV_USBFS_HOST_ROOT_HUB_PortResetIsComplete(DRV_HANDLE handle, uint8_t port);
```

Returns

- true - The reset signaling has completed.
- false - The reset signaling has not completed.

Description

This function returns true if the port reset operation has completed. It should be called after the DRV_USB_HOST_ROOT_HUB_PortReset function to check if the reset operation has completed.

Remarks

The root hub on this particular hardware only contains one port - port 0.

Preconditions

None.

Example

```
// This code shows how the DRV_USB_HOST_ROOT_HUB_PortReset and the
// DRV_USBFS_ROOT_HUB_PortResetIsComplete functions are called to complete a
// port reset sequence.

DRV_HANDLE driverHandle;

// Reset Port 0.
DRV_USB_HOST_ROOT_HUB_PortReset(driverHandle, 0);

// Check if the Reset operation has completed.
if(DRV_USBFS_ROOT_HUB_PortResetIsComplete(driverHandle, 0) == false)
{
    // This means that the Port Reset operation has not completed yet. The
    // DRV_USBFS_ROOT_HUB_PortResetIsComplete function should be called
    // again after some time to check the status.
}
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBFS_Open function).
port	Port to check

Function

bool DRV_USBFS_ROOT_HUB_PortResetIsComplete

(

DRV_HANDLE handle,

uint8_t port

);

DRV_USBFS_HOST_ROOT_HUB_PortResume Function

Resumes the specified root hub port.

File

drv_usbfs.h

C

```
USB_ERROR DRV_USBFS_HOST_ROOT_HUB_PortResume(DRV_HANDLE handle, uint8_t port);
```

Returns

- USB_ERROR_NONE - The function executed successfully.
- USB_ERROR_PARAMETER_INVALID - The driver handle is not valid or the port number does not exist.

Description

This function resumes the root hub. The resume duration is defined by DRV_USBFS_ROOT_HUB_RESUME_DURATION. The status of the resume signaling can be checked using the DRV_USBFS_ROOT_HUB_PortResumeIsComplete function.

Remarks

The root hub on this particular hardware only contains one port - port 0.

Preconditions

None.

Example

```
// This code shows how the DRV_USBFS_HOST_ROOT_HUB_PortResume function is
// called to resume the specified port.

DRV_HANDLE driverHandle;

// Resume Port 0.
DRV_USBFS_HOST_ROOT_HUB_PortResume(driverHandle, 0);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBFS_Open function).
port	Port to resume.

Function

USB_ERROR DRV_USBFS_HOST_ROOT_HUB_PortResume
(
DRV_HANDLE handle,
uint8_t port
);

DRV_USBFS_HOST_ROOT_HUB_PortSpeedGet Function

Returns the speed of at which the port is operating.

File

drv_usbfs.h

C

```
USB_SPEED DRV_USBFS_HOST_ROOT_HUB_PortSpeedGet(DRV_HANDLE handle, uint8_t port);
```

Returns

- USB_SPEED_ERROR - This value is returned if the driver handle is not or if the speed information is not available or if the specified port is not valid.
- USB_SPEED_FULL - A Full Speed device has been connected to the port.
- USB_SPEED_LOW - A Low Speed device has been connected to the port.

Description

This function returns the speed at which the port is operating.

Remarks

The root hub on this particular hardware only contains one port - port 0.

Preconditions

None.

Example

```
// This code shows how the DRV_USBFS_HOST_ROOT_HUB_PortSpeedGet function is
// called to know the operating speed of the port. This also indicates the
// operating speed of the device connected to this port.

DRV_HANDLE driverHandle;
USB_SPEED speed;

speed = DRV_USBFS_HOST_ROOT_HUB_PortSpeedGet(driverHandle, 0);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBFS_Open function).
port	Port number of the port to be analyzed..

Function

USB_SPEED DRV_USBFS_HOST_ROOT_HUB_PortSpeedGet
(
DRV_HANDLE handle,
uint8_t port
);

DRV_USBFS_HOST_ROOT_HUB_PortSuspend Function

Suspends the specified root hub port.

File

drv_usbfs.h

C

```
USB_ERROR DRV_USBFS_HOST_ROOT_HUB_PortSuspend(DRV_HANDLE handle, uint8_t port);
```

Returns

- USB_ERROR_NONE - The function executed successfully.
- USB_ERROR_PARAMETER_INVALID - The driver handle is not valid or the port number does not exist.

Description

This function suspends the root hub port.

Remarks

The root hub on this particular hardware only contains one port - port 0.

Preconditions

None.

Example

```
// This code shows how the DRV_USBFS_HOST_ROOT_HUB_PortSuspend function is
// called to suspend the specified port.

DRV_HANDLE driverHandle;

// Suspend Port 0.
DRV_USBFS_HOST_ROOT_HUB_PortSuspend(driverHandle, 0);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBFS_Open function).
port	Port to suspend.

Function

USB_ERROR DRV_USBFS_ROOT_HUB_PortSuspend(DRV_HANDLE handle, uint8_t port);

f) Data Types and Constants

DRV_USBFS_EVENT Enumeration

Identifies the different events that the USB Driver provides.

File

drv_usbfs.h

C

```
typedef enum {
  DRV_USBFS_EVENT_ERROR = DRV_USB_EVENT_ERROR,
  DRV_USBFS_EVENT_RESET_DETECT = DRV_USB_EVENT_RESET_DETECT,
  DRV_USBFS_EVENT_RESUME_DETECT = DRV_USB_EVENT_RESUME_DETECT,
  DRV_USBFS_EVENT_IDLE_DETECT = DRV_USB_EVENT_IDLE_DETECT,
  DRV_USBFS_EVENT_STALL = DRV_USB_EVENT_STALL,
  DRV_USBFS_EVENT_SOF_DETECT = DRV_USB_EVENT_SOF_DETECT,
  DRV_USBFS_EVENT_DEVICE_SESSION_VALID = DRV_USB_EVENT_DEVICE_SESSION_VALID,
  DRV_USBFS_EVENT_DEVICE_SESSION_INVALID = DRV_USB_EVENT_DEVICE_SESSION_INVALID
} DRV_USBFS_EVENT;
```

Members

Members	Description
DRV_USBFS_EVENT_ERROR = DRV_USB_EVENT_ERROR	Bus error occurred and was reported
DRV_USBFS_EVENT_RESET_DETECT = DRV_USB_EVENT_RESET_DETECT	Host has issued a device reset
DRV_USBFS_EVENT_RESUME_DETECT = DRV_USB_EVENT_RESUME_DETECT	Resume detected while USB in suspend mode
DRV_USBFS_EVENT_IDLE_DETECT = DRV_USB_EVENT_IDLE_DETECT	Idle detected
DRV_USBFS_EVENT_STALL = DRV_USB_EVENT_STALL	Stall handshake has occurred
DRV_USBFS_EVENT_SOF_DETECT = DRV_USB_EVENT_SOF_DETECT	Either Device received SOF or SOF threshold was reached in the Host mode operation
DRV_USBFS_EVENT_DEVICE_SESSION_VALID = DRV_USB_EVENT_DEVICE_SESSION_VALID	Session valid
DRV_USBFS_EVENT_DEVICE_SESSION_INVALID = DRV_USB_EVENT_DEVICE_SESSION_INVALID	Session Invalid

Description

USB Driver Events Enumeration.

This enumeration identifies the different events that are generated by the USB Driver.

Remarks

None.

DRV_USBFS_EVENT_CALLBACK Type

Type of the USB Driver event callback function.

File

drv_usbfs.h

C

```
typedef void (* DRV_USBFS_EVENT_CALLBACK)(uintptr_t hClient, DRV_USBFS_EVENT eventType, void * eventData);
```

Returns

None.

Description

Type of the USB Driver Event Callback Function.

Define the type of the USB Driver event callback function. The client should register an event callback function of this type when it intends to receive events from the USB Driver. The event callback function is registered using the DRV_USBFS_ClientEventCallBackSet function.

Remarks

None.

Parameters

Parameters	Description
hClient	Handle to the driver client that registered this callback function.
eventType	This parameter identifies the event that caused the callback function to be called.
eventData	Pointer to a data structure that is related to this event. This value will be NULL if the event has no related data.

DRV_USBFS_HOST_PIPE_HANDLE Type

Defines the USB Driver Host Pipe Handle type.

File

drv_usbfs.h

C

```
typedef uintptr_t DRV_USBFS_HOST_PIPE_HANDLE;
```

Description

USB Driver Host Pipe Handle.

This type definition defines the type of the USB Driver Host Pipe Handle.

Remarks

None.

DRV_USBFS_INIT Structure

This type definition defines the Driver Initialization Data Structure.

File

drv_usbfs.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  USB_MODULE_ID usbID;
  bool stopInIdle;
  bool suspendInSleep;
  INT_SOURCE interruptSource;
  USB_SPEED operationSpeed;
  DRV_USBFS_OPMODES operationMode;
  void * endpointTable;
  uint32_t rootHubAvailableCurrent;
  DRV_USBFS_ROOT_HUB_PORT_POWER_ENABLE portPowerEnable;
  DRV_USBFS_ROOT_HUB_PORT_INDICATION portIndication;
  DRV_USBFS_ROOT_HUB_PORT_OVER_CURRENT_DETECT portOverCurrentDetect;
} DRV_USBFS_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System Module Initialization
USB_MODULE_ID usbID;	Identifies the USB peripheral to be used. This should be the USB PLIB module instance identifier.
bool stopInIdle;	This should be set to true if the USB module must stop operation in IDLE mode
bool suspendInSleep;	This should be set to true if the USB module must suspend when the CPU enters sleep mode.
INT_SOURCE interruptSource;	Specify the interrupt source for the USB module. This should be the interrupt source identifier for the USB module instance specified in usbID.
USB_SPEED operationSpeed;	Specify the operational speed of the USB module. This should always be set to USB_SPEED_FULL.
DRV_USBFS_OPMODES operationMode;	Specify the operation mode of the USB module. This specifies if the USB module should operate as a Device, Host, or both (Dual Role operation).
void * endpointTable;	A pointer to the endpoint descriptor table. This should be aligned at 512 byte address boundary. The size of the table is equal to DRV_USBFS_ENDPOINT_TABLE_ENTRY_SIZE times the number of endpoints needed in the application.
uint32_t rootHubAvailableCurrent;	Root hub available current in milliamperes. This specifies the amount of current that root hub can provide to the attached device. This should be specified in mA. This is required when the driver is required to operate in host mode.
DRV_USBFS_ROOT_HUB_PORT_POWER_ENABLE portPowerEnable;	When operating in Host mode, the application can specify a Root Hub port enable function. This parameter should point to Root Hub port enable function. If this parameter is NULL, it implies that the Port is always enabled.
DRV_USBFS_ROOT_HUB_PORT_INDICATION portIndication;	When operating in Host mode, the application can specify a Root Port Indication. This parameter should point to the Root Port Indication function. If this parameter is NULL, it implies that Root Port Indication is not supported.
DRV_USBFS_ROOT_HUB_PORT_OVER_CURRENT_DETECT portOverCurrentDetect;	When operating is Host mode, the application can specify a Root Port Overcurrent detection. This parameter should point to the Root Port Indication function. If this parameter is NULL, it implies that Overcurrent detection is not supported.

Description

USB Device Driver Initialization Data.

This structure contains all the data necessary to initialize the USB Driver. A pointer to a structure of this type, containing the desired initialization data, must be passed into the DRV_USBFS_Initialize function.

Remarks

None.

DRV_USBFS_OPMODES Enumeration

Identifies the operating modes supported by the USB Driver.

File

drv_usbfs.h

C

```
typedef enum {
  DRV_USBFS_OPMODE_DUAL_ROLE = DRV_USB_OPMODE_DUAL_ROLE,
  DRV_USBFS_OPMODE_DEVICE = DRV_USB_OPMODE_DEVICE,
  DRV_USBFS_OPMODE_HOST = DRV_USB_OPMODE_HOST,
  DRV_USBFS_OPMODE_OTG = DRV_USB_OPMODE_OTG
} DRV_USBFS_OPMODES;
```

Members

Members	Description
DRV_USBFS_OPMODE_DUAL_ROLE = DRV_USB_OPMODE_DUAL_ROLE	The driver should be able to switch between host and device mode
DRV_USBFS_OPMODE_DEVICE = DRV_USB_OPMODE_DEVICE	The driver should support device mode operation only
DRV_USBFS_OPMODE_HOST = DRV_USB_OPMODE_HOST	The driver should support host mode operation only
DRV_USBFS_OPMODE_OTG = DRV_USB_OPMODE_OTG	The driver should support the OTG protocol

Description

USB Operating Modes Enumeration.

This enumeration identifies the operating modes supported by the USB Driver.

Remarks

None.

DRV_USBFS_ROOT_HUB_PORT_INDICATION Type

USB Root hub Application Hooks (Port Indication).

File

drv_usbfs.h

C

```
typedef void (* DRV_USBFS_ROOT_HUB_PORT_INDICATION)(uint8_t port, USB_HUB_PORT_INDICATOR_COLOR color,
USB_HUB_PORT_INDICATOR_STATE state);
```

Description

USB Root hub Application Hooks (Port Indication).

A function of the type defined here should be provided to the driver root to implement Port Indication. The root hub driver calls this function when it needs to update the state of the port indication LEDs. The application can choose to implement the Amber and Green colors as one LED or two different LEDs. The root hub driver specifies the color and the indicator attribute (on, off or blinking) when it calls this function.

Remarks

None.

DRV_USBFS_ROOT_HUB_PORT_OVER_CURRENT_DETECT Type

USB Root hub Application Hooks (Port Overcurrent detection).

File

drv_usbfs.h

C

```
typedef bool (* DRV_USBFS_ROOT_HUB_PORT_OVER_CURRENT_DETECT)(uint8_t port);
```

Description

USB Root hub Application Hooks (Port Overcurrent detection).

A function of the type defined here should be provided to the driver root hub to check for port over current condition. This function will be called periodically by the root hub driver to check the Overcurrent status of the port. It should continue to return true while the Overcurrent condition exists on the port. It should return false when the Overcurrent condition does not exist.

Remarks

None.

DRV_USBFS_ROOT_HUB_PORT_POWER_ENABLE Type

USB Root hub Application Hooks (Port Power Enable/ Disable).

File

drv_usbfs.h

C

```
typedef void (* DRV_USBFS_ROOT_HUB_PORT_POWER_ENABLE)(uint8_t port, bool control);
```

Description

USB Root hub Application Hooks (Port Power Enable/ Disable).

A function of the type defined here should be provided to the driver root to control port power. The root hub driver will call this function when it needs to enable port power. If the application circuit contains a VBUS switch, the switch should be accessed and controlled by this function. If the enable parameter is true, the switch should be enabled and VBUS should be available on the port. If the enable parameter is false, the switch should be disabled and VBUS should not be available on the port.

Remarks

None.

DRV_USBFS_DEVICE_INTERFACE Macro

USB Driver Device Mode Interface Functions.

File

drv_usbfs.h

C

```
#define DRV_USBFS_DEVICE_INTERFACE
```

Description

USB Driver Device Mode Interface Functions.

The Device Driver interface in the Device Layer Initialization data structure should be set to this value so that Device Layer can access the USB Driver Device Mode functions.

Remarks

None.

DRV_USBFS_ENDPOINT_TABLE_ENTRY_SIZE Macro

USB Driver Endpoint Table Entry Size in bytes.

File

drv_usbfs.h

C

```
#define DRV_USBFS_ENDPOINT_TABLE_ENTRY_SIZE 32
```

Description

USB Driver Endpoint Table Entry Size in bytes.

This constant defines the size (in bytes) of an entry in the endpoint table.

Remarks

None.

DRV_USBFS_HOST_INTERFACE Macro

USB Driver Host Mode Interface Functions.

File

drv_usbfs.h

C

```
#define DRV_USBFS_HOST_INTERFACE
```

Description

USB Driver Host Mode Interface Functions.

The Host Controller Driver interface in the Host Layer Initialization data structure should be set to this value so that Host Layer can access the USB Driver Host Mode functions.

Remarks

None.

DRV_USBFS_HOST_PIPE_HANDLE_INVALID Macro

Value of an Invalid Host Pipe Handle.

File

drv_usbfs.h

C

```
#define DRV_USBFS_HOST_PIPE_HANDLE_INVALID ((DRV_USBFS_HOST_PIPE_HANDLE)(-1))
```

Description

USB Driver Invalid Host Pipe Handle.

This constant defines the value of an Invalid Host Pipe Handle.

Remarks

None.

DRV_USBFS_INDEX_0 Macro

USB Driver Module Index 0 Definition.

File

drv_usbfs.h

C

```
#define DRV_USBFS_INDEX_0 0
```

Description

USB Driver Module Index 0 Definition.

This constant defines the value of USB Driver Index 0. The SYS_MODULE_INDEX parameter of the DRV_USBFS_Initialize and

DRV_USBFS_Open functions should be set to this value to identify instance 0 of the driver.

Remarks

These constants should be used in place of hard-coded numeric literals and should be passed into the DRV_USBFS_Initialize and DRV_USBFS_Open functions to identify the driver instance in use. These are not indicative of the number of modules that are actually supported by the microcontroller.

DRV_USBFS_INDEX_1 Macro

USB Driver Module Index 1 Definition.

File

drv_usbfs.h

C

```
#define DRV_USBFS_INDEX_1 1
```

Description

USB Driver Module Index 1 Definition.

This constant defines the value of USB Driver Index 1. The SYS_MODULE_INDEX parameter of the DRV_USBFS_Initialize and DRV_USBFS_Open functions should be set to this value to identify instance 1 of the driver.

Remarks

These constants should be used in place of hard-coded numeric literals and should be passed into the DRV_USBFS_Initialize and DRV_USBFS_Open functions to identify the driver instance in use. These are not indicative of the number of modules that are actually supported by the microcontroller.

Files

Files

Name	Description
drv_usbfs.h	PIC32MX USB Module Driver Interface File.
drv_usbfs_config_template.h	USB Full Speed (USBFS) Driver Configuration Template.

Description

drv_usbfs.h

PIC32MX USB Module Driver Interface File.

Enumerations

	Name	Description
	DRV_USBFS_EVENT	Identifies the different events that the USB Driver provides.
	DRV_USBFS_OPMODES	Identifies the operating modes supported by the USB Driver.

Functions

	Name	Description
	DRV_USBFS_ClientEventCallBackSet	This function sets up the event callback function that is invoked by the USB controller driver to notify the client of USB bus events.
	DRV_USBFS_Close	Closes an opened-instance of the USB Driver.
	DRV_USBFS_DEVICE_AddressSet	This function will set the USB module address that is obtained from the Host.
	DRV_USBFS_DEVICE_Attach	This function will enable the attach signaling resistors on the D+ and D- lines thus letting the USB Host know that a device has been attached on the bus.
	DRV_USBFS_DEVICE_CurrentSpeedGet	This function returns the USB speed at which the device is operating.
	DRV_USBFS_DEVICE_Detach	This function will disable the attach signaling resistors on the D+ and D- lines thus letting the USB Host know that the device has detached from the bus.

	DRV_USBFS_DEVICE_EndpointDisable	This function disables an endpoint.
	DRV_USBFS_DEVICE_EndpointDisableAll	This function disables all provisioned endpoints.
	DRV_USBFS_DEVICE_EndpointEnable	This function enables an endpoint for the specified direction and endpoint size.
	DRV_USBFS_DEVICE_EndpointIsEnabled	This function returns the enable/disable status of the specified endpoint and direction.
	DRV_USBFS_DEVICE_EndpointIsStalled	This function returns the stall status of the specified endpoint and direction.
	DRV_USBFS_DEVICE_EndpointStall	This function stalls an endpoint in the specified direction.
	DRV_USBFS_DEVICE_EndpointStallClear	This function clears the stall on an endpoint in the specified direction.
	DRV_USBFS_DEVICE_IRPCancel	This function cancels the specific IRP that are queued and in progress at the specified endpoint.
	DRV_USBFS_DEVICE_IRPCancelAll	This function cancels all IRPs that are queued and in progress at the specified endpoint.
	DRV_USBFS_DEVICE_IRPSubmit	This function submits an I/O Request Packet (IRP) for processing to the Hi-Speed USB Driver.
	DRV_USBFS_DEVICE_RemoteWakeupStart	This function causes the device to start Remote Wakeup Signalling on the bus.
	DRV_USBFS_DEVICE_RemoteWakeupStop	This function causes the device to stop the Remote Wakeup Signalling on the bus.
	DRV_USBFS_DEVICE_SOFNumberGet	This function will return the USB SOF packet number.
	DRV_USBFS_HOST_EventsDisable	Disables Host mode events.
	DRV_USBFS_HOST_EventsEnable	Restores the events to the specified the original value.
	DRV_USBFS_HOST_IRPCancel	Cancels the specified IRP.
	DRV_USBFS_HOST_IRPSubmit	Submits an IRP on a pipe.
	DRV_USBFS_HOST_PipeClose	Closes an open pipe.
	DRV_USBFS_HOST_PipeSetup	Open a pipe with the specified attributes.
	DRV_USBFS_HOST_ROOT_HUB_BusSpeedGet	This function returns the operating speed of the bus to which this root hub is connected.
	DRV_USBFS_HOST_ROOT_HUB_Initialize	This function initializes the root hub driver.
	DRV_USBFS_HOST_ROOT_HUB_MaximumCurrentGet	Returns the maximum amount of current that this root hub can provide on the bus.
	DRV_USBFS_HOST_ROOT_HUB_OperationEnable	This function enables or disables root hub operation.
	DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled	Returns the operation enabled status of the root hub.
	DRV_USBFS_HOST_ROOT_HUB_PortNumbersGet	Returns the number of ports this root hub contains.
	DRV_USBFS_HOST_ROOT_HUB_PortReset	Resets the specified root hub port.
	DRV_USBFS_HOST_ROOT_HUB_PortResetIsComplete	Returns true if the root hub has completed the port reset operation.
	DRV_USBFS_HOST_ROOT_HUB_PortResume	Resumes the specified root hub port.
	DRV_USBFS_HOST_ROOT_HUB_PortSpeedGet	Returns the speed of at which the port is operating.
	DRV_USBFS_HOST_ROOT_HUB_PortSuspend	Suspends the specified root hub port.
	DRV_USBFS_Initialize	Initializes the USB Driver.
	DRV_USBFS_Open	Opens the specified USB Driver instance and returns a handle to it.
	DRV_USBFS_Status	Provides the current status of the USB Driver module.
	DRV_USBFS_Tasks	Maintains the driver's state machine when the driver is configured for Polled mode.
	DRV_USBFS_Tasks_ISR	Maintains the driver's Interrupt state machine and implements its ISR.

Macros

	Name	Description
	DRV_USBFS_DEVICE_INTERFACE	USB Driver Device Mode Interface Functions.
	DRV_USBFS_ENDPOINT_TABLE_ENTRY_SIZE	USB Driver Endpoint Table Entry Size in bytes.
	DRV_USBFS_HOST_INTERFACE	USB Driver Host Mode Interface Functions.
	DRV_USBFS_HOST_PIPE_HANDLE_INVALID	Value of an Invalid Host Pipe Handle.
	DRV_USBFS_INDEX_0	USB Driver Module Index 0 Definition.
	DRV_USBFS_INDEX_1	USB Driver Module Index 1 Definition.

Structures

	Name	Description
	DRV_USBFS_INIT	This type definition defines the Driver Initialization Data Structure.

Types

	Name	Description
	DRV_USBFS_EVENT_CALLBACK	Type of the USB Driver event callback function.
	DRV_USBFS_HOST_PIPE_HANDLE	Defines the USB Driver Host Pipe Handle type.
	DRV_USBFS_ROOT_HUB_PORT_INDICATION	USB Root hub Application Hooks (Port Indication).
	DRV_USBFS_ROOT_HUB_PORT_OVER_CURRENT_DETECT	USB Root hub Application Hooks (Port Overcurrent detection).
	DRV_USBFS_ROOT_HUB_PORT_POWER_ENABLE	USB Root hub Application Hooks (Port Power Enable/ Disable).

Description

PIC32MX USB Module Driver Interface Header File.

The PIC32MX Full speed USB Module driver provides a simple interface to manage the "USB" peripheral on PIC32MX microcontrollers. This file defines the interface definitions and prototypes for the USB driver. The driver interface meets the requirements of the MPLAB Harmony USB Host and Device Layer.

File Name

drv_usbfs.h

Company

Microchip Technology Inc.

drv_usbfs_config_template.h

USB Full Speed (USBFS) Driver Configuration Template.

Macros

	Name	Description
	DRV_USBFS_DEVICE_SUPPORT	Determines if the USB Device Functionality should be enabled.
	DRV_USBFS_ENDPOINTS_NUMBER	Configures the number of endpoints to be provisioned in the driver.
	DRV_USBFS_HOST_ATTACH_DEBOUNCE_DURATION	Configures the time duration (in milliseconds) that the driver will wait to re-confirm a device attach.
	DRV_USBFS_HOST_NAK_LIMIT	Configures the NAK Limit for Host Mode Control Transfers.
	DRV_USBFS_HOST_PIPES_NUMBER	Configures the maximum number of pipes that are can be opened when the driver is operating in Host mode.
	DRV_USBFS_HOST_RESET_DURATION	Configures the time duration (in milliseconds) of the Reset Signal.
	DRV_USBFS_HOST_SUPPORT	Determines if the USB Host Functionality should be enabled.
	DRV_USBFS_INSTANCES_NUMBER	Specifies the number of driver instances to be enabled in the application.
	DRV_USBFS_INTERRUPT_MODE	Configures the driver for interrupt or polling mode operation.

Description

USB Full Speed Driver Configuration Template.

This file lists all the configurations constants that affect the operation of the USBFS Driver.

File Name

drv_usbfs_config_template.h

Company

Microchip Technology Inc.

PIC32MZ USB Driver

Provides information on the USB Driver specific to PIC32MZ devices.

Description

The PIC32MZ USB Driver in MPLAB Harmony provides API functions that allow the MPLAB Harmony USB Host and Device Stack to access the

USB while operating on a PIC32MZ microcontroller. The driver implements the USB Driver Common Interface required by the USB Host and Device Stack. It abstracts the USB module operational details from the Host and Device Stack and provides the stacks with a modular access mechanism to the USB. The PIC32MZ USB Driver features the following:

- USB 2.0 High Speed and Full Speed operation in Peripheral mode
- USB 2.0 High Speed, Full Speed and Low Speed USB Peripheral Support in Host mode
- Designed for Dual Role Operation
- Capable of operating multiple USB modules
- Features non-blocking function and is interoperable with other MPLAB Harmony modules
- Features thread safe functions when operating within an RTOS
- Capable of operating in Polled and Interrupt modes
- Implements the USB Driver Common Interface required by the MPLAB Harmony USB Host and Device Stack
- Completely configurable through the MPLAB Harmony Configurator (MHC)
- Implements feature separation (Host and Device mode functions are implemented across different files)
- Designed to use the module's built-in DMA controller and transfer scheduler

Note: This help section only discusses features that are unique to the PIC32MZ USB Driver and are not a part of the USB Driver Common Interface. The driver functions that implement the USB Driver Common Interface are described in the Common Interface Help section.

While the PIC32MZ USB module supports USB "On-The-Go" (OTG), the PIC32MZ Driver does not currently implement USB OTG protocol support.

This help section only provides relevant information about the operation of the USB. The reader is encouraged to refer to the USB 2.0 Specification available at www.usb.org for a detailed explanation of USB protocol.

Using the Library

This topic describes the basic architecture of the USB PIC32MZ Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_usbhs.h`

The interface to the PIC32MZ USB Driver library is defined in the `drv_usbhs.h` header file.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Library Overview

Provides an overview of the library.

Description

The PIC32MZ USB Driver will typically be used by a USB Host and/or Device Stack. The USB Host and Device Stack operate as driver client applications. The driver is initialized as part of the MPLAB Harmony System Initialization. The driver initialization data structure specifies the operation mode (Host, Device, or Dual Role) of the driver. The driver features task routines to be called in the MPLAB Harmony application tasks function (SYS_Tasks function) and the USB Module Interrupt Service Routine (ISR).

The Host and the Device Stack can open the driver only when initialization has completed. It will continue to return an invalid driver handle while the initialization is in progress. Once opened, the Device Mode function can be called if the driver is operating in Device mode. The Host Mode function can be called if the driver is operating in Host mode. In Dual Role operation mode, the driver supports Host and Device operation in the same application. Even then, the driver will either operate as a USB Host or Device. OTG operation is not supported.

The PIC32MZ USB Driver features RTOS thread-safe functions. This allows the driver client application to safely call driver functions across different RTOS threads. Not all of the driver functions are interrupt-safe.

In addition to the USB Driver, which implements the USB Driver Common Interface, the PIC32MZ USB Driver implements functions which are required for its operation in the MPLAB Harmony framework. The following table lists the different categories of functions in the PIC32MZ USB Driver.

Library Interface Section	Description
System Function	These functions are accessed by the MPLAB Harmony System module. They allow the driver to be initialized, deinitialized and maintained. These functions are implemented in the `drv_usbhs.c` source file.
Client Core Functions	These functions allow the USB Host and Device Stack to open, close and perform other general driver operations. These functions are a part of the USB Driver Common Interface and are implemented in `drv_usbhs.c` source file.
Device Mode Operation Functions	These functions allow the USB Device Stack to perform USB Device mode specific driver operations. These functions are a part of the USB Driver Common Interface and are implemented in `drv_usbhs_device.c` source file

Host Mode Operation Functions	These functions allow the USB Host Stack to perform USB Host mode specific driver operations. These functions are a part of the USB Driver Common Interface and are implemented in `drv_usbhs_host.c` source file.
Root Hub Functions	These functions allow the USB Host Stack to access the driver Root hub operation. These functions are a part of the USB Driver Common Interface and are implemented in `drv_usbhs_host.c` source file.

Abstraction Model

Provides information on the abstraction model for the library.

Description

The PIC32MZ USB Driver implements the abstraction model defined by the USB Driver Common interface. This interface abstracts USB module specific details and provides a module independent interface to the driver client applications.

While operating in Device mode, the driver expects the client application (the USB Device Stack) to enable endpoints and then submit I/O request packet (IRP) requests to the enabled endpoints. Multiple IRPs can be queued on an endpoint. The driver calls the IRP callback function when the IRP is processed. The driver allows the client application to also attach and detach the device on the bus. It generates events which indicate USB states.

While operating in Host mode, the driver expects the client application (the USB Host Stack) to open pipes to endpoints on the connected device. The client application can then submit IRPs to the pipes. Multiple IRPs can be queued on a pipe. The driver calls the IRP callback function when the IRP is processed. The driver will call application defined functions to enumerate and denumerate a device. These functions are called when the driver detect device attach and detach respectively. The driver also exports root hub functions to the client application. This allows the client application to treat the driver as a single port hub

Please refer to the PIC32 USB Driver Common Interface help section for more details on the driver abstraction model.

How the Library Works

Provides information on how the library works.

Description

This section only explains aspects of driver operation which are unique to the PIC32MZ USB Driver. Major driver operations are described in the PIC32 USB Driver Common Interface help section.

Driver Initialization

Note: While generating a MPLAB Harmony USB project with MHC, the initialization code for the driver is generated automatically based on selections made in the USB Host stack or Device Stack Configuration trees.

The PIC32MZ USB Driver must be initialized so that a client application can open. The client application will not be able to open the driver if the initialization is in progress or has failed. The driver is initialized by calling the DRV_USBHS_Initialize function. This function is called from the SYS_Initialize function in the MPLAB Harmony application project and accepts two input parameters. The `index` parameter defines the instance of the USB Driver to be initialized. This becomes significant when the PIC32MZ microcontroller has more than one USB module. The `init` parameter is a driver-specific data structure of the type DRV_USBHS_INIT. This structure is shown in the following code example.

```
/* This code show the PIC32MZ USB Driver Initialization data structure.
 * A structure of this type must be provided to the DRV_USBHS_Initialize
 * function. */

typedef struct
{
    /* System Module Initialization */
    SYS_MODULE_INIT moduleInit;

    /* Identifies the USB peripheral to be used. This should be the USB PLIB
       module instance identifier. */
    uint8_t usbID;

    /* This should be set to true if the USB module must stop operation in Idle
       mode */
    bool stopInIdle;

    /* This should be set to true if the USB module must suspend when the CPU
       enters Sleep mode. */
    bool suspendInSleep;

    /* Specify the interrupt source for the USB module. This should be Interrupt
```

```
    PLIB Interrupt source identifier for the USB module instance specified in
    usbID. */
INT_SOURCE interruptSource;

/* Specify the interrupt source for the USB module specific DMA controller.
 * This should be the PLIB Interrupt source identified for the USB
 * module instance specified in usbID. */
INT_SOURCE interruptSourceUSBDma;

/* Specify the operational speed of the USB module. This should always be
   set to USB_SPEED_FULL. */
USB_SPEED operationSpeed;

/* Specify the operation mode of the USB module. This defines if the USB
 * module will support Device, Host or Dual Role operation */
DRV_USBHS_OPMODES operationMode;

/* A pointer to the endpoint descriptor table. This should be aligned at 512
   byte address boundary. The size of the table is equal to the
   DRV_USBHS_ENDPOINT_TABLE_ENTRY_SIZE times the number of endpoints needed
   in the application. */
void * endpointTable;

/* Root hub available current in mA. This specifies the amount of current
   that root hub can provide to the attached device. This should be
   specified in mA. This is required when the driver is required to operate
   in host mode. */
uint32_t rootHubAvailableCurrent;

/* When operating in Host mode, the application can specify a Root Hub port
   enable function. This parameter should point to Root Hub port enable
   function. If this parameter is NULL, it implies that the Port is always
   enabled. */
DRV_USBHS_ROOT_HUB_PORT_POWER_ENABLE portPowerEnable;

/* When operating in Host mode, the application can specify a Root Port
   Indication. This parameter should point to the Root Port Indication
   function. If this parameter is NULL, it implies that Root Port Indication
   is not supported. */
DRV_USBHS_ROOT_HUB_PORT_INDICATION portIndication;

/* When operating is Host mode, the application can specify a Root Port
   Overcurrent detection. This parameter should point to the Root Port
   Indication function. If this parameter is NULL, it implies that
   Overcurrent detection is not supported. */
DRV_USBHS_ROOT_HUB_PORT_OVER_CURRENT_DETECT portOverCurrentDetect;

} DRV_USBHS_INIT;
```

The operationMode parameter defines the driver operation mode. This can be set to DRV_USBFS_OPMODE_DEVICE, DRV_USBFS_OPMODE_HOST, or DRV_USBFS_OPMODE_DUAL_ROLE for Device, Host and Dual Role operation, respectively.

The `rootHubAvailableCurrent` parameter should be set to the maximum current that the VBUS power supply can provide on the bus. The driver does not use this information directly. It provides this data to the client application while operating in Host mode.

The `portPowerEnable` parameter must point to a Port Power Enable function. The driver, while operating in Host mode, will call this function to enable the VBUS switch. This function should activate the VBUS switch if the driver calls this function with the enable parameter set to true. It should deactivate the switch if the driver calls this function with the `enable` parameter set to false. This parameter should be set to NULL if such a switch (of the switch control) is not available in the application.

The `portIndication` parameter must point to a Port Indication function. The driver, while operating in Host mode, will call this function to indicate the current state of the port. The driver will call this function with LED color status as defined in Chapter 11 of the USB 2.0 Specification. This parameter should be set to NULL if such a LED indication is not available in the application.

The `portOverCurrentDetect` parameter must point to a Port Overcurrent Detect function. The driver, while operating in Host mode, will call this function periodically to check if the attached device is overdrawing current. If the function should return true if such a condition exists. This parameter should be set to NULL if such detection is not available in the application.

The following code example shows initialization of the driver for Device mode operation.

```
/* This code shows an example of DRV_USBHS_INIT data structure for
 * Device mode operation. Here the driver is initialized to work with USB0 USB
 * module. */
```

```
DRV_USBHS_INIT init;
SYS_MODULE_OBJ usbDriverObj;

const DRV_USBHS_INIT drvUSBInit =
{
    /* Interrupt Source for USB module */
    .interruptSource = INT_SOURCE_USB_1,

    /* DMA Interrupt Source for USB module */
    .interruptSourceUSBDma = INT_SOURCE_USB_1_DMA,

    /* System module initialization */
    .moduleInit = {SYS_MODULE_POWER_RUN_FULL},

    /* Module operate in device mode */
    .operationMode = DRV_USBHS_OPMODE_DEVICE,

    /* Module operated at high speed */
    .operationSpeed = USB_SPEED_HIGH,

    /* Stop in idle */
    .stopInIdle = false,

    /* Suspend in sleep */
    .suspendInSleep = false,

    /* Identifies peripheral (PLIB-level) ID */
    .usbID = USBHS_ID_0
};

void SYS_Initialize(void)
{
    /* Initialize the USB Driver. Note how the init parameter is typecast to
     * SYS_MODULE_INIT type. The SYS_MODULE_OBJ returned by this function call
     * is passed to the driver tasks routine. DRV_USBHS_INDEX_0 is helper
     * constant defined in drv_usbfs.h */

    usbDriverObj = DRV_USBHS_Initialize(DRV_USBHS_INDEX_0, (SYS_MODULE_INIT *)(drvUSBInit));
}

void SYS_Tasks(void)
{
    /* The polled state of the USB driver is updated by calling the
     * DRV_USBHS_Tasks function in the SYS_Tasks() function. The
     * DRV_USBHS_Tasks() takes the driver module object returned by the
     * DRV_USBHS_Initialize funciton as a parameter. */

    DRV_USBHS_Tasks(usbDriverObj);
}

void __ISR(_USB_VECTOR, ipl4AUTO) _IntHandlerUSBInstance0(void)
{
    /* The DRV_USBHS_Tasks_ISR function update the interrupt state of the USB
     * Driver. If the driver is configured for Polling mode, this function need
     * not be invoked or included in the project. */

    DRV_USBHS_Tasks_ISR(usbDriverObj);
}

void __ISR ( _USB_DMA_VECTOR,ipl4AUTO) _IntHandlerUSBInstance0_USBDMA ( void )
{
    DRV_USBHS_Tasks_ISR_USBDMA(usbDriverObj);
}
```

The following code example shows initialization of the driver for Host mode operation.

```
/* This code shows an example of how the Hi-Speed USB (USBHS) driver can be configured
 * for Host mode operation. In this example, the
 * BSP_USBVBUSSwitchOverCurrentDetect function checks for over current condition
 * and the BSP_USBVBUSPowerEnable function enables the VBUS power. The port
```

```
    * indication function is not implemented and hence the portIndication member of
    * the initialization data structure is set to NULL. */

   /* The implementation of the port over current detect, indication and the VBUS
    * power supply functions is discussed later in this help section. */

   DRV_USBHS_INIT drvUSBHSInit =
   {
       /* This should always be set to SYS_MODULE_POWER_RUN_FULL. */
       .moduleInit = {SYS_MODULE_POWER_RUN_FULL},

       /* Interrupt Source for the USB module */
       .interruptSource = INT_SOURCE_USB_1,

       /* Interrupt Source for the USB DMA module */
       .interruptSourceUSBDma = INT_SOURCE_USB_1_DMA,

       /* Configure for host mode operation. */
       .operationMode = DRV_USBHS_OPMODE_HOST,

       /* The driver should run at high speed. */
       .operationSpeed = USB_SPEED_HIGH,

       /* Port indication function is not implemented and is not available */
       .portIndication = NULL,

       /* This is the VBUS Power enable function */
       .portPowerEnable = BSP_USBVBUSPowerEnable,

       /* This is the over current detect function. */
       .portOverCurrentDetect = BSP_USBVBUSSwitchOverCurrentDetect,

       /* Here we state that the VBUS power supply can provide at most 500 mA of
        * current */
       .rootHubAvailableCurrent = 500,

       /* Moudule will operate in IDLE. */
       .stopInIdle = false,

       /* Module will not suspend automatically in sleep */
       .suspendInSleep = false,

       /* USB Module ID is 1 */
       .usbID = USBHS_ID_0

   };

   void SYS_Initialize(void)
   {
       /* Initialize the USB Driver. Note how the init parameter is typecast to
        * SYS_MODULE_INIT type. The SYS_MODULE_OBJ returned by this function call
        * is passed to the driver tasks routine. DRV_USBHS_INDEX_0 is helper
        * constant defined in drv_usbfs.h */

       usbDriverObj = DRV_USBHS_Initialize(DRV_USBHS_INDEX_0, (SYS_MODULE_INIT *)(drvUSBInit));
   }

   void SYS_Tasks(void)
   {
       /* The polled state of the USB driver is updated by calling the
        * DRV_USBHS_Tasks function in the SYS_Tasks() function. The
        * DRV_USBHS_Tasks takes the driver module object returned by the
        * DRV_USBHS_Initialize funciton as a parameter. */

       DRV_USBHS_Tasks(usbDriverObj);
   }

   void __ISR( _USB_VECTOR , IPL4AUTO)_IntHandler_USB_stub ( void )
   {
```

```
    /* The DRV_USBHS_Tasks_ISR function updates the interrupt state of the USB
     * Driver. If the driver is configured for polling mode, this function need
     * not be invoked or included in the project. */

    DRV_USBHS_Tasks_ISR(usbDriverObj);
}

void __ISR ( _USB_DMA_VECTOR, IPL4AUTO)  _IntHandlerUSBInstance0_USBDMA ( void )
{
    /* The DRV_USBHS_Tasks_ISR_USBDMA function update the DMA transfer state of
     * the USB Driver. */

     DRV_USBHS_Tasks_ISR_USBDMA(usbDriverObj);
}
```

The PIC32MX USB Driver requires definition of configuration constants to be available in the `system_config.h` file of the MPLAB Harmony Application Project Configuration. Refer to the Configuring the Library section for details.

Multi-client Operation

The PIC32MZ USB Driver supports multi-client operation. In that, it can be opened by two application clients. This is required where Dual Operation is desired. The following should be noted when using multi-client operation:

- The driver should be initialized for Dual Role Operation mode.
- The DRV_USBHS_Open function can be called at the most twice in the application. The driver supports a maximum of two clients.
- A client can access either the host or device functionality of the driver. It cannot do both.
- It is possible for the two clients to operate in two different threads while operating with an RTOS.

Note: The typical the application clients for PIC32MZ USB Driver would be the MPLAB Harmony USB Host and Device Stack. The complexity of operating the driver in Dual Role mode is handled by the stack operation. The MHC will configure the driver for Dual Role operation when such operation is selected in USB Stack configuration tree.

USB Driver Common Interface

The PIC32MZ USB Driver exports its implementation of the USB Driver Common Interface to the Host and Device Layer via the DRV_USBHS_HOST_INTERFACE and DRV_USBHS_DEVICE_INTERFACE structures. The DRV_USBHS_HOST_INTERFACE structure is defined in the `drv_usbhs_host.c` file. The following code example shows this structure.

```
/*********************************************************
 * This structure is a set of pointer to the USBHS driver
 * functions. It is provided to the host and device layer
 * as the interface to the driver.
 * *******************************************************/

DRV_USB_HOST_INTERFACE gDrvUSBHSHostInterface =
{
    .open = DRV_USBHS_Open,
    .close = DRV_USBHS_Close,
    .eventHandlerSet = DRV_USBHS_ClientEventCallBackSet,
    .hostIRPSubmit = DRV_USBHS_HOST_IRPSubmit,
    .hostIRPCancel = DRV_USBHS_HOST_IRPCancel,
    .hostPipeSetup = DRV_USBHS_HOST_PipeSetup,
    .hostPipeClose = DRV_USBHS_HOST_PipeClose,
    .hostEventsDisable = DRV_USBHS_HOST_EventsDisable,
    .hostEventsEnable = DRV_USBHS_HOST_EventsEnable,
    .rootHubInterface.rootHubPortInterface.hubPortReset = DRV_USBHS_HOST_ROOT_HUB_PortReset,
    .rootHubInterface.rootHubPortInterface.hubPortSpeedGet = DRV_USBHS_HOST_ROOT_HUB_PortSpeedGet,
    .rootHubInterface.rootHubPortInterface.hubPortResetIsComplete =
DRV_USBHS_HOST_ROOT_HUB_PortResetIsComplete,
    .rootHubInterface.rootHubPortInterface.hubPortSuspend = DRV_USBHS_HOST_ROOT_HUB_PortSuspend,
    .rootHubInterface.rootHubPortInterface.hubPortResume = DRV_USBHS_HOST_ROOT_HUB_PortResume,
    .rootHubInterface.rootHubMaxCurrentGet = DRV_USBHS_HOST_ROOT_HUB_MaximumCurrentGet,
    .rootHubInterface.rootHubPortNumbersGet = DRV_USBHS_HOST_ROOT_HUB_PortNumbersGet,
    .rootHubInterface.rootHubSpeedGet = DRV_USBHS_HOST_ROOT_HUB_BusSpeedGet,
    .rootHubInterface.rootHubInitialize = DRV_USBHS_HOST_ROOT_HUB_Initialize,
    .rootHubInterface.rootHubOperationEnable = DRV_USBHS_HOST_ROOT_HUB_OperationEnable,
    .rootHubInterface.rootHubOperationIsEnabled = DRV_USBHS_HOST_ROOT_HUB_OperationIsEnabled,
};
```

The DRV_USBFS_DEVICE_INTERFACE structure is defined in the `drv_usbhs_device.c` file. The following code example shows this structure.

The MPLAB Harmony USB Host and Device stack perform driver independent access through the function pointers contained in these structures.

```
/*************************************************
```

```
 * This structure is a pointer to a set of USB Driver
 * Device mode functions. This set is exported to the
 * device layer when the device layer must use the
 * PIC32MZ USB Controller.
 ************************************************/

DRV_USB_DEVICE_INTERFACE gDrvUSBHSDeviceInterface =
{
    .open = DRV_USBHS_Open,
    .close = DRV_USBHS_Close,
    .eventHandlerSet = DRV_USBHS_ClientEventCallBackSet,
    .deviceAddressSet = DRV_USBHS_DEVICE_AddressSet,
    .deviceCurrentSpeedGet = DRV_USBHS_DEVICE_CurrentSpeedGet,
    .deviceSOFNumberGet = DRV_USBHS_DEVICE_SOFNumberGet,
    .deviceAttach = DRV_USBHS_DEVICE_Attach,
    .deviceDetach = DRV_USBHS_DEVICE_Detach,
    .deviceEndpointEnable = DRV_USBHS_DEVICE_EndpointEnable,
    .deviceEndpointDisable = DRV_USBHS_DEVICE_EndpointDisable,
    .deviceEndpointStall = DRV_USBHS_DEVICE_EndpointStall,
    .deviceEndpointStallClear = DRV_USBHS_DEVICE_EndpointStallClear,
    .deviceEndpointIsEnabled = DRV_USBHS_DEVICE_EndpointIsEnabled,
    .deviceEndpointIsStalled = DRV_USBHS_DEVICE_EndpointIsStalled,
    .deviceIRPSubmit = DRV_USBHS_DEVICE_IRPSubmit,
    .deviceIRPCancelAll = DRV_USBHS_DEVICE_IRPCancelAll,
    .deviceRemoteWakeupStop = DRV_USBHS_DEVICE_RemoteWakeupStop,
    .deviceRemoteWakeupStart = DRV_USBHS_DEVICE_RemoteWakeupStart,
    .deviceTestModeEnter = DRV_USBHS_DEVICE_TestModeEnter


};
```

Operation with RTOS

The PIC32MZ USB Driver is designed to operate with a RTOS. The driver implementation uses the MPLAB Harmony Operating System Abstraction Layer (OSAL). This allows the driver to function with entire range of RTOSes supported in MPLAB Harmony. The following points must be considered while using the driver with an RTOS.

- The driver can be opened from different threads
- In Device mode, an enabled endpoint should only be accessed from one thread. For example, if an application requires two endpoints, Endpoint 2 and Endpoint 3, the application could contain two threads, one accessing Endpoint 2 and another accessing Endpoint 3. The thread accessing Endpoint 2 cannot access Endpoint 3.
- While operating in Host mode, endpoint pipes can be opened from different threads. A pipe handle to an open pipe cannot be shared across threads.

USB DMA Operation

The PIC32MZ USB module features a built-in DMA controller. This controller works independently of the PIC32MZ DMA controller. The PIC32MZ USB Driver uses USB DMA controller to expedite transfer of memory from the USB module FIFO to user application memory. The following should be noted for the USB DMA controller:

- If the PIC32MZ USB Driver could not allocate a DMA channel (all channels are busy), it will use the CPU instructions to unload the endpoint FIFOs
- The USB module and the USB DMA controller interrupt priorities should be the same
- The application buffer start address should always be aligned on a 16-byte boundary and should be placed in coherent memory. Refer to the description of the DRV_USBHS_HOST_IRPSubmit and DRV_USBHS_DEVICE_IRPSubmit functions for details on how the user application buffer should be allocated.

Root Hub Operation

The PIC32MZ USB Driver implements a Root Hub Driver Interface. This allows the driver to emulate a hub. The USB Host Stack enumerates the Root Hub as a device. The Host Stack then does not differentiate between an external hub and the root hub. While emulating a hub, the PIC32MZ USB Driver Root Hub appears as a single port hub.

As a part of the Root Hub interface, the PIC32MZ USB Driver requires the application to supply functions for hub features that it does not implement. These features are:

- Port Overcurrent Detect
- VBUS Switch Control
- Port Indication

A pointer to these functions (if implemented) must be supplied through the driver initialization data (of the type DRV_USBHS_INIT) structure at the time of driver initialization. The application has the option of not implementing these functions. In such a case, the function pointers for the unimplemented function, in the initialization data structure should be set to NULL.

The root hub driver must also be able to communicate the maximum current capability of its port to the USB Host Layer. The PIC32MZ USB Controller does not contain built-in (hardware implemented) functionality for controlling the root hub port current. To facilitate this request, the driver will report the current capability that was specified in the `rootHubAvailableCurrent` parameter of the driver initialization data structure. The application must set this parameter to report the current supply capability of the VBUS power supply. The USB Host Layer uses this value to manage the bus current budget. If a connected device reports a configuration that requires more current than what the VBUS power supply can provide, the host will not set the configuration.

Port Overcurrent Detect

The Root Hub operation in PIC32MZ USB Driver will periodically call a Port Overcurrent Detect function to detect if an overcurrent condition is active on the port. The application must supply this function if port overcurrent detection is needed. The PIC32MZ USB Controller does not contain built-in (hardware implemented) functionality for checking overcurrent condition. The overcurrent condition on the port can occur in a case where the attached device has malfunctioned or when the USB VBUS line has short circuited to ground.

The signature of the function and an example implementation is shown in the following code example. The function must return (and must continue to return) true if an overcurrent condition exists on the port.

```
/* This code shows an example implementation of the
 * portOverCurrentDetect function. The PIC32MZ USB Driver will call this
 * function periodically to check if an over current condition exists on the
 * port. In this example, we assume that the over current detect pin from an
 * external circuit in the system, is connected to  port RD0 and the pin logic
 * is active high. The function must return true if an over current condition is
 * present on this pin */


bool BSP_USBVBUSSwitchOverCurrentDetect(uint8_t port)
{
    if(PLIB_PORTS_PinGet(PORTS_ID_0, PORT_CHANNEL_D, 0) == 1)
    {
        return(true);
    }
    else
    {
        return(false);
    }
}
```

VBUS Switch Control

The PIC32MZ USB Driver Root Hub operation will attempt to control the VBUS power supply to the port. Because the PIC32MZ USB Controller does not contain built-in (hardware implemented) functionality for checking controlling VBUS, such a control function must be supplied by the application. The root hub operation will access this function when the PIC32MX USB Driver will call the portPowerEnable function as a part of the Bus Enable sequence.

The following code shows an example of how this function can be implemented.

```
/* This code shows an example implementation of the VBUS Power Enable
 * function. The PIC32MZ USB Driver will call this function as a part of bus
 * enable function. In this example, it is assumed that system contains an
 * external VBUS power switch and this is control by port RB5.
 */

void BSP_USBVBUSPowerEnable(uint8_t port, bool enable)
{
    if(enable)
    {
        PLIB_PORTS_PinSet(PORTS_ID_0, PORT_CHANNEL_B, PORTS_BIT_POS_5);
    }
    else
    {
        PLIB_PORTS_PinClear(PORTS_ID_0, PORT_CHANNEL_B, PORTS_BIT_POS_5);
    }
}
```

Port Indication Function

The Root Hub Operation in the PIC32MZ USB Driver allows display of Port LED status. If the application requires this indication, it must implement a function which the Root Hub operation would call when a change in the Root Hub port has occurred. The port indication operation is specified in Section 11.5.3 of the USB 2.0 Specification.

```
/* This code shows an example implementation of the port indication
 * function. The PIC32MZ USB Driver call this function when it wants to indicate
 * port status. It is assumed that three function to switch off, blink and
 * switch on an LED are available. It is further assumed that these function
```

```
 * accept the color of the LED to operated on. */

void BSP_RootHubPortIndication
(
    uint8_t port,
    USB_HUB_PORT_INDICATOR_COLOR color,
    USB_HUB_PORT_INDICATOR_STATE state
)
{
    /* The color parameter indicates the color of the LED to be affected. The
     * color will be either USB_HUB_PORT_INDICATOR_COLOR_GREEN or
     * USB_HUB_PORT_INDICATOR_COLOR_AMBER. */

    switch (state)
    {
        case USB_HUB_PORT_INDICATOR_STATE_OFF:
            BSP_SwitchLEDOff(color);
            break;
        case USB_HUB_PORT_INDICATOR_STATE_BLINKING:
            BSP_LEDBlink(color);
            break;
        case USB_HUB_PORT_INDICATOR_STATE_ON:
            BSP_SwitchLEDOn(color);
            break;
        default:
            break;
    }
}
```

Configuring the Library

Provides information on the configuring the library.

Macros

	Name	Description
	DRV_USBHS_DEVICE_SUPPORT	Determines if the USB Device Functionality should be enabled.
	DRV_USBHS_ENDPOINTS_NUMBER	Configures the number of endpoints to be provisioned in the driver.
	DRV_USBHS_HOST_ATTACH_DEBOUNCE_DURATION	Configures the time duration (in milliseconds) that the driver will wait to reconfirm a device attach.
	DRV_USBHS_HOST_NAK_LIMIT	Configures the NAK Limit for Host Mode Control Transfers.
	DRV_USBHS_HOST_PIPES_NUMBER	Configures the maximum number of pipes that are can be opened when the driver is operating in Host mode.
	DRV_USBHS_HOST_RESET_DURATION	Configures the time duration (in milliseconds) of the Reset Signal.
	DRV_USBHS_HOST_SUPPORT	Determines if the USB Host Functionality should be enabled.
	DRV_USBHS_INSTANCES_NUMBER	Specifies the number of driver instances to be enabled in the application.
	DRV_USBHS_INTERRUPT_MODE	Configures the driver for interrupt or polling mode operation.

Description

The PIC32MZ USB Driver requires the specification of compile-time configuration macros. These macros define resource usage, feature availability, and dynamic behavior of the driver. These configuration macros should be defined in the `system_config.h` file.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

DRV_USBHS_DEVICE_SUPPORT Macro

Determines if the USB Device Functionality should be enabled.

File

drv_usbhs_config_template.h

C

```
#define DRV_USBHS_DEVICE_SUPPORT true
```

Description

Hi-Speed USB Driver Device Mode Support.

This constant should be set to true if USB device support is required in the application. It should be set to false if device support is not required.

Remarks

This constant should always be defined.

DRV_USBHS_ENDPOINTS_NUMBER Macro

Configures the number of endpoints to be provisioned in the driver.

File

drv_usbhs_config_template.h

C

```
#define DRV_USBHS_ENDPOINTS_NUMBER 3
```

Description

Hi-Speed USB Driver Endpoint Numbers.

This constant configures the number of endpoints that the driver needs to manage. When DRV_USBHS_DEVICE_SUPPORT is enabled, this constant should be set to the total number of endpoints to be enabled in the device. When enabled, an endpoint can be used for communication. Using any direction of an endpoint will require that the entire endpoint to be enabled.

Consider the case of a composite USB Device that contains a CDC and MSD function. The CDC function will require one Bulk endpoint (OUT and IN directions) and one Interrupt endpoint (IN direction). The MSD function will require one Bulk endpoint (IN and OUT directions). This design can be implemented by using four endpoints. Endpoint 0 is used for the mandatory control interface. Endpoint 1 is used for CDC Bulk interface. Endpoint 2 is used for CDC Interrupt interface and Endpoint 3 is used for MSD Bulk Interface. The constant should then be set to 4.

For Host mode operation, this constant should be set to 1. Setting this value to greater than 1 will result in unused data memory allocation.

Remarks

This constant should always be defined.

DRV_USBHS_HOST_ATTACH_DEBOUNCE_DURATION Macro

Configures the time duration (in milliseconds) that the driver will wait to reconfirm a device attach.

File

drv_usbhs_config_template.h

C

```
#define DRV_USBHS_HOST_ATTACH_DEBOUNCE_DURATION 500
```

Description

Hi-Speed USB Driver Host Mode Attach Debounce Duration.

This constant configures the time duration (in milliseconds) that the driver will wait to reconfirm a device attach. When the driver first detects a device attach, it will start a timer for the duration specified by the constant. When the timer expires, the driver will check if the device is still attached. If so, the driver will then signal an attach event to the host stack. The duration allows for the device attach to become electro-mechanically stable.

Remarks

This constant should always be defined when DRV_USBHS_HOST_SUPPORT is set to true.

DRV_USBHS_HOST_NAK_LIMIT Macro

Configures the NAK Limit for Host Mode Control Transfers.

File

drv_usbhs_config_template.h

C

```
#define DRV_USBHS_HOST_NAK_LIMIT 2000
```

Description

Hi-Speed USB Driver Host Mode Control Transfers NAK Limit.

This constant configures the number of NAKs that the driver can accept from the device in the data stage of a control transfer before aborting the control transfer with a USB_HOST_IRP_STATUS_ERROR_NAK_TIMEOUT. Setting this constant to 0 will disable NAK limit checking. This constant should be adjusted to enable USB host compatibility with USB Devices that require more time to process control transfers.

Remarks

This constant should always be defined when DRV_USBHS_HOST_SUPPORT is set to true.

DRV_USBHS_HOST_PIPES_NUMBER Macro

Configures the maximum number of pipes that are can be opened when the driver is operating in Host mode.

File

drv_usbhs_config_template.h

C

```
#define DRV_USBHS_HOST_PIPES_NUMBER 10
```

Description

Hi-Speed USB Driver Host Mode Pipes Number.

This constant configures the maximum number of pipes that can be opened when the driver is operating in Host mode. Calling the DRV_USBHS_HOST_PipeSetup function will cause a pipe to be opened. Calling this function when DRV_USBHS_HOST_PIPES_NUMBER number of pipes have already been opened will cause the function to return an Invalid Pipe Handle. This constant should be configured to account for the maximum number of devices and the device types to be supported by the host application.

For example, if the USB Host application must support two USB Mass Storage devices and one CDC device. A CDC device requires four pipes and a Mass Storage Device requires three pipes. This constant should therefore be set to a value of 9 (four bulk pipes for two Mass Storage devices + two bulk pipes and one Interrupt pipe for one CDC device and two control pipes for two devices). Allocating pipes consumes data memory.

While enabling support for multiple devices, through a Hub, the application should consider the worst case requirement while configuring this constant. For example, a case where devices with the most number of pipe requirements are connected to the hub. At the same time, setting this constant to more than what is required will consume data memory.

Remarks

This constant should always be defined when DRV_USBHS_HOST_SUPPORT is set to true.

DRV_USBHS_HOST_RESET_DURATION Macro

Configures the time duration (in milliseconds) of the Reset Signal.

File

drv_usbhs_config_template.h

C

```
#define DRV_USBHS_HOST_RESET_DURATION 100
```

Description

Hi-Speed USB Driver Host Mode Reset Duration.

This constant configures the duration of the reset signal. The driver generates a reset signal when the USB Host stack requests for a root hub port reset. The driver will generate the reset signal for the duration specified by this constant and will then stop generating the reset signal.

Remarks

This constant should always be defined when DRV_USBHS_HOST_SUPPORT is set to true.

DRV_USBHS_HOST_SUPPORT Macro

Determines if the USB Host Functionality should be enabled.

File

drv_usbhs_config_template.h

C

```
#define DRV_USBHS_HOST_SUPPORT false
```

Description

Hi-Speed USB Driver Host Mode Support.

This constant should be set to true if USB Host mode support is required in the application. It should be set to false if host support is not required.

Remarks

This constant should always be defined.

DRV_USBHS_INSTANCES_NUMBER Macro

Specifies the number of driver instances to be enabled in the application.

File

drv_usbhs_config_template.h

C

```
#define DRV_USBHS_INSTANCES_NUMBER 1
```

Description

Hi-Speed USB Driver Instances Number.

This constant defines the number of driver instances to be enabled in the application. This will be typically be the number of USB controllers to be used in the application. On PIC32MZ microcontrollers that have one USB controller, this value will always be 1. On PIC32MZ microcontrollers that have two USB controllers, this value could be one or two, depending on whether one or two USB segments are required. To conserve data memory, this constant should be set to exactly the number of USB controllers that are required in the system.

Remarks

This constant should always be defined.

DRV_USBHS_INTERRUPT_MODE Macro

Configures the driver for interrupt or polling mode operation.

File

drv_usbhs_config_template.h

C

```
#define DRV_USBHS_INTERRUPT_MODE true
```

Description

Hi-Speed USB Driver Interrupt Mode.

This constant configures the driver for interrupt or polling operation. If this flag is set to true, the driver will operate in Interrupt mode. If the flag is set to false, the driver will operate in Polled mode. In Polled mode, the driver interrupt state machine gets updated in the SYS_Tasks function. If the driver is configured for Interrupt mode, the driver Interrupt state machine gets updated in the driver Interrupt Service Routine(ISR). It is always recommended for the driver to operate in Interrupt mode.

Remarks

This constant should always be defined.

Building the Library

This section lists the files that are available in the PIC32MZ USB Driver Library.

Description

This section list the files that are available in the `\src` folder of the PIC32MZ USB Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/usb/usbhs`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
/drv_usbhs.h	This file should be included by any .c file which accesses the PIC32MZ USB Driver API. This one file contains the prototypes for all driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/drv_usbhs.c	This file should always be included in the project when using the PIC3MZ USB Driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
/src/dynamic/drv_usbhs_device.c	This file should be included in the project if Device mode operation is required.
/src/dynamic/drv_usbhs_host.c	This file should be included in the project if Host mode operation is required.

Module Dependencies

The PIC32MZ USB Driver Library depends on the following modules:

- Interrupt System Service Library

Library Interface

a) System Functions

	Name	Description
	DRV_USBHS_Initialize	Initializes the Hi-Speed USB Driver.
	DRV_USBHS_Status	Provides the current status of the Hi-Speed USB Driver module.
	DRV_USBHS_Tasks	Maintains the driver's state machine when the driver is configured for Polled mode.
	DRV_USBHS_Tasks_ISR	Maintains the driver's Interrupt state machine and implements its ISR.
	DRV_USBHS_Tasks_ISR_USBDMA	Maintains the driver's DMA Transfer state machine and implements its ISR.

b) Client Core Functions

	Name	Description
	DRV_USBHS_ClientEventCallBackSet	This function sets up the event callback function that is invoked by the USB controller driver to notify the client of USB bus events.
	DRV_USBHS_Close	Closes an opened-instance of the Hi-Speed USB Driver.
	DRV_USBHS_Open	Opens the specified Hi-Speed USB Driver instance and returns a handle to it.

c) Device Mode Operation Functions

	Name	Description
	DRV_USBHS_DEVICE_AddressSet	This function will set the USB module address that is obtained from the Host.
	DRV_USBHS_DEVICE_Attach	This function will enable the attach signaling resistors on the D+ and D- lines thus letting the USB Host know that a device has been attached on the bus.
	DRV_USBHS_DEVICE_CurrentSpeedGet	This function will return the USB speed at which the device is operating.
	DRV_USBHS_DEVICE_Detach	This function will disable the attach signaling resistors on the D+ and D- lines thus letting the USB Host know that the device has detached from the bus.
	DRV_USBHS_DEVICE_EndpointDisable	This function disables an endpoint.
	DRV_USBHS_DEVICE_EndpointDisableAll	This function disables all provisioned endpoints.
	DRV_USBHS_DEVICE_EndpointEnable	This function enables an endpoint for the specified direction and endpoint size.
	DRV_USBHS_DEVICE_EndpointIsEnabled	This function returns the enable/disable status of the specified endpoint and direction.
	DRV_USBHS_DEVICE_EndpointIsStalled	This function returns the stall status of the specified endpoint and direction.
	DRV_USBHS_DEVICE_EndpointStall	This function stalls an endpoint in the specified direction.

	DRV_USBHS_DEVICE_EndpointStallClear	This function clears the stall on an endpoint in the specified direction.
	DRV_USBHS_DEVICE_IRPCancel	This function cancels the specific IRP that are queued and in progress at the specified endpoint.
	DRV_USBHS_DEVICE_IRPCancelAll	This function cancels all IRPs that are queued and in progress at the specified endpoint.
	DRV_USBHS_DEVICE_IRPSubmit	This function submits an I/O Request Packet (IRP) for processing to the Hi-Speed USB Driver.
	DRV_USBHS_DEVICE_RemoteWakeupStart	This function causes the device to start Remote Wakeup Signalling on the bus.
	DRV_USBHS_DEVICE_RemoteWakeupStop	This function causes the device to stop the Remote Wakeup Signalling on the bus.
	DRV_USBHS_DEVICE_SOFNumberGet	This function will return the USB SOF packet number.
	DRV_USBHS_DEVICE_TestModeEnter	This function enables the specified USB 2.0 Test Mode.
	DRV_USBHS_DEVICE_TestModeExit	This function disables the specified USB 2.0 Test Mode.

d) Host Mode Operation Functions

	Name	Description
	DRV_USBHS_HOST_EventsDisable	Disables Host mode events.
	DRV_USBHS_HOST_EventsEnable	Restores the events to the specified the original value.
	DRV_USBHS_HOST_IRPCancel	Cancels the specified IRP.
	DRV_USBHS_HOST_IRPSubmit	Submits an IRP on a pipe.
	DRV_USBHS_HOST_PipeClose	Closes an open pipe.
	DRV_USBHS_HOST_PipeSetup	Open a pipe with the specified attributes.

e) Root Hub Functions

	Name	Description
	DRV_USBHS_HOST_ROOT_HUB_BusSpeedGet	This function returns the operating speed of the bus to which this root hub is connected.
	DRV_USBHS_HOST_ROOT_HUB_Initialize	This function initializes the root hub driver.
	DRV_USBHS_HOST_ROOT_HUB_MaximumCurrentGet	Returns the maximum amount of current that this root hub can provide on the bus.
	DRV_USBHS_HOST_ROOT_HUB_OperationEnable	This function enables or disables root hub operation.
	DRV_USBHS_HOST_ROOT_HUB_OperationIsEnabled	Returns the operation enabled status of the root hub.
	DRV_USBHS_HOST_ROOT_HUB_PortNumbersGet	Returns the number of ports this root hub contains.
	DRV_USBHS_HOST_ROOT_HUB_PortReset	Resets the specified root hub port.
	DRV_USBHS_HOST_ROOT_HUB_PortResetIsComplete	Returns true if the root hub has completed the port reset operation.
	DRV_USBHS_HOST_ROOT_HUB_PortResume	Resumes the specified root hub port.
	DRV_USBHS_HOST_ROOT_HUB_PortSpeedGet	Returns the speed of at which the port is operating.
	DRV_USBHS_HOST_ROOT_HUB_PortSuspend	Suspends the specified root hub port.

f) Data Types and Constants

	Name	Description
	DRV_USBHS_EVENT	Identifies the different events that the Hi-Speed USB Driver provides.
	DRV_USBHS_EVENT_CALLBACK	Type of the Hi-Speed USB Driver event callback function.
	DRV_USBHS_HOST_PIPE_HANDLE	Defines the Hi-Speed USB Driver Host Pipe Handle type.
	DRV_USBHS_INIT	This type definition defines the Driver Initialization Data Structure.
	DRV_USBHS_OPMODES	Identifies the operating modes supported by the Hi-Speed USB Driver.
	DRV_USBHS_ROOT_HUB_PORT_INDICATION	USB Root hub Application Hooks (Port Indication).
	DRV_USBHS_ROOT_HUB_PORT_OVER_CURRENT_DETECT	USB Root hub Application Hooks (Port Overcurrent detection).
	DRV_USBHS_ROOT_HUB_PORT_POWER_ENABLE	USB Root hub Application Hooks (Port Power Enable/ Disable).
	DRV_USBHS_DEVICE_INTERFACE	Hi-Speed USB Driver Device Mode Interface Functions.
	DRV_USBHS_HOST_INTERFACE	Hi-Speed USB Driver Host Mode Interface Functions.
	DRV_USBHS_HOST_PIPE_HANDLE_INVALID	Value of an Invalid Host Pipe Handle.
	DRV_USBHS_INDEX_0	Hi-Speed USB Driver Module Index 0 Definition.

Description

This section describes the functions of the PIC32MZ USB Driver Library.

Refer to each section for a detailed description.

a) System Functions

DRV_USBHS_Initialize Function

Initializes the Hi-Speed USB Driver.

File

drv_usbhs.h

C

```
SYS_MODULE_OBJ DRV_USBHS_Initialize(const SYS_MODULE_INDEX drvIndex, const SYS_MODULE_INIT * const init);
```

Returns

- SYS_MODULE_OBJ_INVALID - The driver initialization failed.
- A valid System Module Object - The driver initialization was able to start. It may have not completed and requires the DRV_USBHS_Tasks function to be called periodically. This value will never be the same as SYS_MODULE_OBJ_INVALID.

Description

This function initializes the Hi-Speed USB Driver, making it ready for clients to open. The driver initialization does not complete when this function returns. The DRV_USBHS_Tasks function must called periodically to complete the driver initialization. The DRV_USBHS_Open function will fail if the driver was not initialized or if initialization has not completed.

Remarks

This function must be called before any other Hi-Speed USB Driver function is called. This function should only be called once during system initialization unless DRV_USBHS_Deinitialize is called to deinitialize the driver instance.

Preconditions

None.

Example

```
 // The following code shows an example initialization of the
 // driver. The USB module to be used is USB1.  The module should not
 // automatically suspend when the  microcontroller enters Sleep mode.  The
 // module should continue  operation when the module enters Idle mode.  The
 // power state is set to run at full clock speeds. Device Mode operation
 // should be at FULL speed. The size of the endpoint table is set for two
 // endpoints.

DRV_USBHS_INIT moduleInit;

usbInitData.usbID               = USBHS_ID_0;
usbInitData.opMode              = DRV_USBHS_OPMODE_DEVICE;
usbInitData.stopInIdle          = false;
usbInitData.suspendInSleep      = false;
usbInitData.operationSpeed      = USB_SPEED_FULL;
usbInitData.interruptSource     = INT_SOURCE_USB;

usbInitData.sysModuleInit.powerState = SYS_MODULE_POWER_RUN_FULL ;

// This is how this data structure is passed to the initialize
// function.

DRV_USBHS_Initialize(DRV_USBHS_INDEX_0, (SYS_MODULE_INIT *) &usbInitData);
```

Parameters

Parameters	Description
drvIndex	Ordinal number of driver instance to be initialized. This should be set to DRV_USBHS_INDEX_0 if driver instance 0 needs to be initialized.
init	Pointer to a data structure containing data necessary to initialize the driver. This should be a DRV_USBHS_INIT structure reference typecast to SYS_MODULE_INIT reference.

Function

SYS_MODULE_OBJ DRV_USBHS_Initialize
(
const SYS_MODULE_INDEX drvIndex,
const SYS_MODULE_INIT * const init
)

DRV_USBHS_Status Function

Provides the current status of the Hi-Speed USB Driver module.

File

drv_usbhs.h

C

```
SYS_STATUS DRV_USBHS_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that the driver is ready.
- SYS_STATUS_UNINITIALIZED - Indicates that the driver has never been initialized.

Description

This function provides the current status of the Hi-Speed USB Driver module.

Remarks

None.

Preconditions

The DRV_USBHS_Initialize function must have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_USBHS_Initialize
SYS_STATUS          status;
DRV_USBHS_INIT moduleInit;

usbInitData.usbID               = USBHS_ID_0;
usbInitData.opMode              = DRV_USBHS_OPMODE_DEVICE;
usbInitData.stopInIdle          = false;
usbInitData.suspendInSleep      = false;
usbInitData.operationSpeed      = USB_SPEED_FULL;
usbInitData.interruptSource     = INT_SOURCE_USB;

usbInitData.sysModuleInit.powerState = SYS_MODULE_POWER_RUN_FULL ;

// This is how this data structure is passed to the initialize
// function.

DRV_USBHS_Initialize(DRV_USBHS_INDEX_0, (SYS_MODULE_INIT *) &usbInitData);

// The status of the driver can be checked.
status = DRV_USBHS_Status(object);
if(SYS_STATUS_READY == status)
{
    // Driver is ready to be opened.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_USBHS_Initialize function.

Function

SYS_STATUS DRV_USBHS_Status (SYS_MODULE_OBJ object)

DRV_USBHS_Tasks Function

Maintains the driver's state machine when the driver is configured for Polled mode.

File

drv_usbhs.h

C

```
void DRV_USBHS_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Maintains the driver's Polled state machine. This function should be called from the SYS_Tasks function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks function (SYS_Tasks). This function will never block.

Preconditions

The DRV_USBHS_Initialize function must have been called for the specified Hi-Speed USB Driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_USBHS_Initialize

while (true)
{
    DRV_USBHS_Tasks(object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_USBHS_Initialize function).

Function

void DRV_USBHS_Tasks(SYS_MODULE_OBJ object)

DRV_USBHS_Tasks_ISR Function

Maintains the driver's Interrupt state machine and implements its ISR.

File

drv_usbhs.h

C

```
void DRV_USBHS_Tasks_ISR(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the driver's internal Interrupt state machine and implement its ISR for interrupt-driven implementations.

Remarks

This function should be called from the USB ISR. For multiple USB modules, it should be ensured that the correct Hi-Speed USB Driver system module object is passed to this function.

Preconditions

The DRV_USBHS_Initialize function must have been called for the specified Hi-Speed USB Driver instance.

Example

```
SYS_MODULE_OBJ object;     // Returned from DRV_USBHS_Initialize

while (true)
{
    DRV_USBHS_Tasks_ISR (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_USBHS_Initialize).

Function

void DRV_USBHS_Tasks_ISR(SYS_MODULE_OBJ object)

DRV_USBHS_Tasks_ISR_USBDMA Function

Maintains the driver's DMA Transfer state machine and implements its ISR.

File

drv_usbhs.h

C

```
void DRV_USBHS_Tasks_ISR_USBDMA(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the driver's internal DMA Transfer state machine and implement its ISR for interrupt-driven implementations.

Remarks

This function should be called from the USB DMA ISR. For multiple USB modules, it should be ensured that the correct Hi-Speed USB Driver system module object is passed to this function.

Preconditions

The DRV_USBHS_Initialize function must have been called for the specified Hi-Speed USB Driver instance.

Example

```
SYS_MODULE_OBJ object;     // Returned from DRV_USBHS_Initialize

while (true)
{
    DRV_USBHS_Tasks_ISR_USBDMA (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_USBHS_Initialize).

Function

void DRV_USBHS_Tasks_ISR_USBDMA(SYS_MODULE_OBJ object)

b) Client Core Functions

DRV_USBHS_ClientEventCallBackSet Function

This function sets up the event callback function that is invoked by the USB controller driver to notify the client of USB bus events.

File

drv_usbhs.h

C

```
void DRV_USBHS_ClientEventCallBackSet(DRV_HANDLE handle, uintptr_t hReferenceData, DRV_USB_EVENT_CALLBACK
myEventCallBack);
```

Returns

None.

Description

This function sets up the event callback function that is invoked by the USB controller driver to notify the client of USB bus events. The callback is disabled by either not calling this function after the DRV_USBHS_Open function has been called or by setting the myEventCallBack argument as NULL. When the callback function is called, the hReferenceData argument is returned.

Remarks

Typical usage of the Hi-Speed USB Driver requires a client to register a callback.

Preconditions

None.

Example

```
// Set the client event callback for the Device Layer.  The
// USBDeviceLayerEventHandler function is the event handler. When this
// event handler is invoked by the driver, the driver returns back the
// second argument specified in the following function (which in this case
// is the Device Layer data structure). This allows the application
// firmware to identify, as an example, the Device Layer object associated
// with this callback.

DRV_USBHS_ClientEventCallBackSet(myUSBDevice.usbDriverHandle, (uintptr_t)&myUSBDevice,
USBDeviceLayerEventHandler);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).
hReferenceData	Object (could be a pointer) that is returned with the callback.
myEventCallBack	Callback function for all USB events.

Function

void DRV_USBHS_ClientEventCallBackSet

(

DRV_HANDLE handle,

uintptr_t hReferenceData,

DRV_USBHS_EVENT_CALLBACK myEventCallBack

);

DRV_USBHS_Close Function

Closes an opened-instance of the Hi-Speed USB Driver.

File

drv_usbhs.h

C

```
void DRV_USBHS_Close(DRV_HANDLE handle);
```

Returns

None.

Description

This function closes an opened-instance of the Hi-Speed USB Driver, invalidating the handle.

Remarks

After calling this function, the handle passed in handle parameter must not be used with any of the other driver functions. A new handle must be obtained by calling DRV_USBHS_Open function before the caller may use the driver again.

Preconditions

The DRV_USBHS_Initialize function must have been called for the specified Hi-Speed USB Driver instance. DRV_USBHS_Open function must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_USBHS_Open

DRV_USBHS_Close(handle);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).

Function

void DRV_USBHS_Close(DRV_HANDLE handle)

DRV_USBHS_Open Function

Opens the specified Hi-Speed USB Driver instance and returns a handle to it.

File

drv_usbhs.h

C

```
DRV_HANDLE DRV_USBHS_Open(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

Returns

- DRV_HANDLE_INVALID - The driver could not be opened successfully.This can

happen if the driver initialization was not complete or if an internal error has occurred.

- A Valid Driver Handle - This is an arbitrary value and is returned if the function was successful. This value will never be the same as DRV_HANDLE_INVALID.

Description

This function opens the specified Hi-Speed USB Driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The intent flag should always be DRV_IO_INTENT_EXCLUSIVE|DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_NON_BLOCKING. Any other setting of the intent flag will return a invalid driver handle. A driver instance can only support one client. Trying to open a driver that has an existing client will result in an unsuccessful function call.

Remarks

The handle returned is valid until the DRV_USBHS_Close function is called. The function will typically return DRV_HANDLE_INVALID if the driver was not initialized. In such a case the client should try to open the driver again.

Preconditions

Function DRV_USBHS_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

// This code assumes that the driver has been initialized.
handle = DRV_USBHS_Open(DRV_USBHS_INDEX_0, DRV_IO_INTENT_EXCLUSIVE| DRV_IO_INTENT_READWRITE|
DRV_IO_INTENT_NON_BLOCKING);
```

```
if(DRV_HANDLE_INVALID == handle)
{
    // The application should try opening the driver again.
}
```

Parameters

Parameters	Description
drvIndex	Identifies the driver instance to be opened. As an example, this value can be set to DRV_USBHS_INDEX_0 if instance 0 of the driver has to be opened.
intent	Should always be (DRV_IO_INTENT_EXCLUSIVE|DRV_IO_INTENT_READWRITE| DRV_IO_INTENT_NON_BLOCKING).

Function

DRV_HANDLE DRV_USBHS_Open
(
const SYS_MODULE_INDEX drvIndex,
const DRV_IO_INTENT intent
)

c) Device Mode Operation Functions

DRV_USBHS_DEVICE_AddressSet Function

This function will set the USB module address that is obtained from the Host.

File

drv_usbhs.h

C

```
void DRV_USBHS_DEVICE_AddressSet(DRV_HANDLE handle, uint8_t address);
```

Returns

None.

Description

This function will set the USB module address that is obtained from the Host in a setup transaction. The address is obtained from the SET_ADDRESS command issued by the Host. The primary (first) client of the driver uses this function to set the module's USB address after decoding the setup transaction from the Host.

Remarks

None.

Preconditions

None.

Example

```
// This function should be called by the first client of the driver,
// which is typically the Device Layer. The address to set is obtained
// from the Host during enumeration.

DRV_USBHS_DEVICE_AddressSet(deviceLayer, 4);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).
address	The address of this module on the USB bus.

Function

void DRV_USBHS_DEVICE_AddressSet(DRV_HANDLE handle, uint8_t address);

DRV_USBHS_DEVICE_Attach Function

This function will enable the attach signaling resistors on the D+ and D- lines thus letting the USB Host know that a device has been attached on the bus.

File

drv_usbhs.h

C

```
void DRV_USBHS_DEVICE_Attach(DRV_HANDLE handle);
```

Returns

None.

Description

This function enables the pull-up resistors on the D+ or D- lines thus letting the USB Host know that a device has been attached on the bus . This function should be called when the driver client is ready to receive communication from the Host (typically after all initialization is complete). The USB 2.0 specification requires VBUS to be detected before the data line pull-ups are enabled. The application must ensure the same.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// Open the device driver and attach the device to the USB.
handle = DRV_USBHS_Open(DRV_USBHS_INDEX_0, DRV_IO_INTENT_EXCLUSIVE| DRV_IO_INTENT_READWRITE|
DRV_IO_INTENT_NON_BLOCKING);

// Register a callback
DRV_USBHS_ClientEventCallBackSet(handle, (uintptr_t)&myDeviceLayer, MyDeviceLayerEventCallback);

// The device can be attached when VBUS Session Valid event occurs
void MyDeviceLayerEventCallback(uintptr_t handle, DRV_USBHS_EVENT event, void * hReferenceData)
{
    switch(event)
    {
        case DRV_USBHS_EVENT_DEVICE_SESSION_VALID:
            // A valid VBUS was detected.
            DRV_USBHS_DEVICE_Attach(handle);
            break;

        case DRV_USBHS_EVENT_DEVICE_SESSION_INVALID:
            // VBUS is not valid anymore. The device can be disconnected.
            DRV_USBHS_DEVICE_Detach(handle);
            break;

        default:
            break;
        }
    }
}
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).

Function

void DRV_USBHS_DEVICE_Attach(DRV_HANDLE handle);

DRV_USBHS_DEVICE_CurrentSpeedGet Function

This function will return the USB speed at which the device is operating.

File

drv_usbhs.h

C

```
USB_SPEED DRV_USBHS_DEVICE_CurrentSpeedGet(DRV_HANDLE handle);
```

Returns

Returns a member of the USB_SPEED type.

Description

This function will return the USB speed at which the device is operating.

Remarks

None.

Preconditions

Only valid after the device is attached to the Host and Host has completed reset signaling.

Example

```
// Get the current speed.

USB_SPEED deviceSpeed;

deviceSpeed = DRV_USBHS_DEVICE_CurrentSpeedGet(deviceLayer);

if(deviceLayer == USB_SPEED_HIGH)
{
    // Possibly adjust buffers for higher throughput.
}
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).

Function

USB_SPEED DRV_USBHS_DEVICE_CurrentSpeedGet(DRV_HANDLE handle);

DRV_USBHS_DEVICE_Detach Function

This function will disable the attach signaling resistors on the D+ and D- lines thus letting the USB Host know that the device has detached from the bus.

File

drv_usbhs.h

C

```
void DRV_USBHS_DEVICE_Detach(DRV_HANDLE handle);
```

Returns

None.

Description

This function disables the pull-up resistors on the D+ or D- lines. This function should be called when the application wants to disconnect the device from the bus (typically to implement a soft detach or switch to Host mode operation). A self-powered device should be detached from the bus when the VBUS is not valid.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// Open the device driver and attach the device to the USB.
handle = DRV_USBHS_Open(DRV_USBHS_INDEX_0, DRV_IO_INTENT_EXCLUSIVE| DRV_IO_INTENT_READWRITE|
DRV_IO_INTENT_NON_BLOCKING);

// Register a callback
DRV_USBHS_ClientEventCallBackSet(handle, (uintptr_t)&myDeviceLayer, MyDeviceLayerEventCallback);

// The device can be detached when VBUS Session Invalid event occurs
void MyDeviceLayerEventCallback(uintptr_t handle, DRV_USBHS_EVENT event, void * hReferenceData)
{
    switch(event)
    {
        case DRV_USBHS_EVENT_DEVICE_SESSION_VALID:
            // A valid VBUS was detected.
            DRV_USBHS_DEVICE_Attach(handle);
            break;

        case DRV_USBHS_EVENT_DEVICE_SESSION_INVALID:
            // VBUS is not valid anymore. The device can be disconnected.
            DRV_USBHS_DEVICE_Detach(handle);
            break;

        default:
            break;
        }
    }
}
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).

Function

void DRV_USBHS_DEVICE_Detach(DRV_HANDLE handle);

DRV_USBHS_DEVICE_EndpointDisable Function

This function disables an endpoint.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_DEVICE_EndpointDisable(DRV_HANDLE handle, USB_ENDPOINT endpointAndDirection);
```

Returns

- USB_ERROR_NONE - The endpoint was successfully enabled.
- USB_ERROR_DEVICE_ENDPOINT_INVALID - The endpoint that is being accessed is not a valid endpoint (endpoint was not provisioned through the DRV_USBHS_ENDPOINTS_NUMBER configuration constant) defined for this driver instance.

Description

This function disables an endpoint. If the endpoint type is a control endpoint type, both directions are disabled. For non-control endpoints, the function disables the specified direction only. The direction to be disabled is specified by the Most Significant Bit (MSB) of the endpointAndDirection parameter.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to disable
```

```
// a control endpoint. Note that the direction parameter is ignored.
// For a control endpoint, both the directions are disabled.

USB_ENDPOINT ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 0);

DRV_USBHS_DEVICE_EndpointDisable(handle, ep );

// This code shows an example of how to disable a BULK IN
// endpoint

USB_ENDPOINT ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

DRV_USBHS_DEVICE_EndpointDisable(handle, ep );
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).
endpointAndDirection	Specifies the endpoint and direction.

Function

USB_ERROR DRV_USBHS_DEVICE_EndpointDisable
(
DRV_HANDLE handle,
USB_ENDPOINT endpointAndDirection
)

DRV_USBHS_DEVICE_EndpointDisableAll Function

This function disables all provisioned endpoints.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_DEVICE_EndpointDisableAll(DRV_HANDLE handle);
```

Returns

- USB_ERROR_NONE - The function exited successfully.
- USB_ERROR_PARAMETER_INVALID - The driver handle is invalid.

Description

This function disables all provisioned endpoints in both directions.

Remarks

This function is typically called by the USB Device Layer to disable all endpoints upon detecting a bus reset.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to disable all endpoints.

DRV_USBHS_DEVICE_EndpointDisableAll(handle);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).

Function

USB_ERROR DRV_USBHS_DEVICE_EndpointDisableAll(DRV_HANDLE handle)

DRV_USBHS_DEVICE_EndpointEnable Function

This function enables an endpoint for the specified direction and endpoint size.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_DEVICE_EndpointEnable(DRV_HANDLE handle, USB_ENDPOINT endpointAndDirection,
USB_TRANSFER_TYPE transferType, uint16_t endpointSize);
```

Returns

- USB_ERROR_NONE - The endpoint was successfully enabled.
- USB_ERROR_DEVICE_ENDPOINT_INVALID - If the endpoint that is being accessed is not a valid endpoint defined for this driver instance. The value of DRV_USBHS_ENDPOINTS_NUMBER configuration constant should be adjusted.
- USB_ERROR_PARAMETER_INVALID - The driver handle is invalid.

Description

This function enables an endpoint for the specified direction and endpoint size. The function will enable the endpoint for communication in one direction at a time. It must be called twice if the endpoint is required to communicate in both the directions, with the exception of control endpoints. If the endpoint type is a control endpoint, the endpoint is always bidirectional and the function needs to be called only once.

The size of the endpoint must match the wMaxPacketSize reported in the endpoint descriptor for this endpoint. A transfer that is scheduled over this endpoint will be scheduled in wMaxPacketSize transactions. The function does not check if the endpoint is already in use. It is the client's responsibility to make sure that a endpoint is not accidentally reused.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to enable Endpoint
// 0 for control transfers. Note that for a control endpoint, the
// direction parameter is ignored. A control endpoint is always
// bidirectional. Endpoint size is 64 bytes.

uint8_t ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 0);

DRV_USBHS_DEVICE_EndpointEnable(handle, ep, USB_TRANSFER_TYPE_CONTROL, 64);

// This code shows an example of how to set up a endpoint
// for BULK IN transfer. For an IN transfer, data moves from device
// to Host. In this example, Endpoint 1 is enabled. The maximum
// packet size is 64.

uint8_t ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

DRV_USBHS_DEVICE_EndpointEnable(handle, ep, USB_TRANSFER_TYPE_BULK, 64);

// If Endpoint 1 must also be set up for BULK OUT, the
// DRV_USBHS_DEVICE_EndpointEnable function must be called again, as shown
// here.

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_HOST_TO_DEVICE, 1);
```

```
DRV_USBHS_DEVICE_EndpointEnable(handle, ep, USB_TRANSFER_TYPE_BULK, 64);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).
endpointAndDirection	Specifies the endpoint and direction.
transferType	Should be USB_TRANSFER_TYPE_CONTROL for control endpoint, USB_TRANSFER_TYPE_BULK for bulk endpoint, USB_TRANSFER_TYPE_INTERRUPT for interrupt endpoint and USB_TRANSFER_TYPE_ISOCHRONOUS for isochronous endpoint.
endpointSize	Maximum size (in bytes) of the endpoint as reported in the endpoint descriptor.

Function

```
USB_ERROR DRV_USBHS_DEVICE_EndpointEnable
(
    DRV_HANDLE handle,
USB_ENDPOINT endpointAndDirection,
USB_TRANSFER_TYPE transferType,
uint16_t endpointSize
);
```

DRV_USBHS_DEVICE_EndpointIsEnabled Function

This function returns the enable/disable status of the specified endpoint and direction.

File

drv_usbhs.h

C

```
bool DRV_USBHS_DEVICE_EndpointIsEnabled(DRV_HANDLE client, USB_ENDPOINT endpointAndDirection);
```

Returns

- true - The endpoint is enabled.
- false - The endpoint is disabled.

Description

This function returns the enable/disable status of the specified endpoint and direction.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how the
// DRV_USBHS_DEVICE_EndpointIsEnabled function can be used to obtain the
// status of Endpoint 1 and IN direction.

USB_ENDPOINT ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

if(DRV_USBHS_ENDPOINT_STATE_DISABLED ==
            DRV_USBHS_DEVICE_EndpointIsEnabled(handle, ep))
{
    // Endpoint is disabled. Enable endpoint.

    DRV_USBHS_DEVICE_EndpointEnable(handle, ep, USB_ENDPOINT_TYPE_BULK, 64);

}
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).
endpointAndDirection	Specifies the endpoint and direction.

Function

bool DRV_USBHS_DEVICE_EndpointIsEnabled
(
DRV_HANDLE handle,
USB_ENDPOINT endpointAndDirection
)

DRV_USBHS_DEVICE_EndpointIsStalled Function

This function returns the stall status of the specified endpoint and direction.

File

drv_usbhs.h

C

```
bool DRV_USBHS_DEVICE_EndpointIsStalled(DRV_HANDLE client, USB_ENDPOINT endpoint);
```

Returns

- true - The endpoint is stalled.
- false - The endpoint is not stalled.

Description

This function returns the stall status of the specified endpoint and direction.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how the
// DRV_USBHS_DEVICE_EndpointIsStalled function can be used to obtain the
// stall status of Endpoint 1 and IN direction.

USB_ENDPOINT ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

if(true == DRV_USBHS_DEVICE_EndpointIsStalled (handle, ep))
{
    // Endpoint stall is enabled. Clear the stall.

    DRV_USBHS_DEVICE_EndpointStallClear(handle, ep);

}
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).
endpointAndDirection	Specifies the endpoint and direction.

Function

bool DRV_USBHS_DEVICE_EndpointIsStalled
(

DRV_HANDLE handle,
USB_ENDPOINT endpointAndDirection
)

DRV_USBHS_DEVICE_EndpointStall Function

This function stalls an endpoint in the specified direction.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_DEVICE_EndpointStall(DRV_HANDLE handle, USB_ENDPOINT endpointAndDirection);
```

Returns

- USB_ERROR_NONE - The endpoint was successfully enabled.
- USB_ERROR_PARAMETER_INVALID - The driver handle is not valid.
- USB_ERROR_DEVICE_ENDPOINT_INVALID - If the endpoint that is being accessed is out of the valid endpoint defined for this driver instance.
- USB_ERROR_OSAL_FUNCTION - An error with an OSAL function called in this function.

Description

This function stalls an endpoint in the specified direction.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to stall an endpoint. In
// this example, Endpoint 1 IN direction is stalled.

USB_ENDPOINT ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

DRV_USBHS_DEVICE_EndpointStall(handle, ep);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).
endpointAndDirection	Specifies the endpoint and direction.

Function

USB_ERROR DRV_USBHS_DEVICE_EndpointStall
(
DRV_HANDLE handle,
USB_ENDPOINT endpointAndDirection
)

DRV_USBHS_DEVICE_EndpointStallClear Function

This function clears the stall on an endpoint in the specified direction.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_DEVICE_EndpointStallClear(DRV_HANDLE handle, USB_ENDPOINT endpointAndDirection);
```

Returns

- USB_ERROR_NONE - The endpoint was successfully enabled.
- USB_ERROR_PARAMETER_INVALID - The driver handle is not valid.
- USB_ERROR_DEVICE_ENDPOINT_INVALID - If the endpoint that is being accessed is out of the valid endpoint defined for this driver instance.

Description

This function clears the stall on an endpoint in the specified direction.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to clear a stall. In this
// example, the stall condition on Endpoint 1 IN direction is cleared.

USB_ENDPOINT ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

DRV_USBHS_DEVICE_EndpointStallClear(handle, ep);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).
endpointAndDirection	Specifies the endpoint and direction.

Function

USB_ERROR DRV_USBHS_DEVICE_EndpointStallClear

(

DRV_HANDLE handle,

USB_ENDPOINT endpointAndDirection

)

DRV_USBHS_DEVICE_IRPCancel Function

This function cancels the specific IRP that are queued and in progress at the specified endpoint.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_DEVICE_IRPCancel(DRV_HANDLE client, USB_DEVICE_IRP * irp);
```

Returns

- USB_ERROR_NONE - The IRP have been canceled successfully.
- USB_ERROR_PARAMETER_INVALID - Invalid parameter or the IRP already has been aborted or completed
- USB_ERROR_OSAL_FUNCTION - An OSAL function called in this function did not execute successfully.

Description

This function attempts to cancel the processing of a queued IRP. An IRP that was in the queue but yet to be processed will be cancelled successfully and the IRP callback function will be called from this function with the USB_DEVICE_IRP_STATUS_ABORTED status. The application can release the data buffer memory used by the IRP when this callback occurs. If the IRP was in progress (a transaction in on the bus) when the cancel function was called, the IRP will be canceled only when an ongoing or the next transaction has completed. The IRP callback

function will then be called in an interrupt context. The application should not release the related data buffer unless the IRP callback has occurred.

Remarks

The size returned after the ABORT callback will be always 0 regardless of the amount of data that has been sent or received. The client should not assume any data transaction has happened for an canceled IRP. If the last transaction of the IRP was in progress, the IRP cancel does not have any effect. The first transaction of any ongoing IRP cannot be canceled.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to cancel IRP.  In this example the IRP
// has been scheduled from a device to the Host.

USB_ENDPOINT ep;
USB_DEVICE_IRP irp;

ep.direction = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

irp.data = myDataBufferToSend;
irp.size = 130;
irp.flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
irp.callback = MyIRPCompletionCallback;
irp.referenceData = (uintptr_t)&myDeviceLayerObj;

if (DRV_USBHS_DEVICE_IRPSubmit(handle, ep, &irp) != USB_ERROR_NONE)
{
    // This means there was an error.
}
else
{
    // Check the status of the IRP.
    if(irp.status != USB_DEVICE_IRP_STATUS_COMPLETED)
    {
        // Cancel the submitted IRP.
        if (DRV_USBHS_DEVICE_IRPCancel(handle, &irp) != USB_ERROR_NONE)
        {
            // The IRP Cancel request submission was successful.
            // IRP cancel status will be notified through the callback
            // function.
        }
        else
        {
            // The IRP may have been completed before IRP cancel operation.
            // could start. No callback notification will be generated.
        }
    }
    else
    {
        // The IRP processing must have been completed before IRP cancel was
        // submitted.
    }
}

void MyIRPCallback(USB_DEVICE_IRP * irp)
{
    // Check if the IRP callback is for a Cancel request
    if(irp->status == USB_DEVICE_IRP_STATUS_ABORTED)
    {
        // IRP cancel completed
    }
 }
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).
irp	Pointer to the IRP to cancel.

Function

USB_ERROR DRV_USBHS_DEVICE_IRPCancel
(
DRV_HANDLE client,
USB_DEVICE_IRP * irp
)

DRV_USBHS_DEVICE_IRPCancelAll Function

This function cancels all IRPs that are queued and in progress at the specified endpoint.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_DEVICE_IRPCancelAll(DRV_HANDLE client, USB_ENDPOINT endpointAndDirection);
```

Returns

- USB_ERROR_NONE - The endpoint was successfully enabled.
- USB_ERROR_DEVICE_ENDPOINT_INVALID - If the endpoint that is being accessed is out of the valid endpoint defined for this driver instance.
- USB_ERROR_PARAMETER_INVALID - The driver handle is not valid.
- USB_ERROR_OSAL_FUNCTION - An OSAL function called in this function did not execute successfully.

Description

This function cancels all IRPs that are queued and in progress at the specified endpoint.

Remarks

None.

Preconditions

The Client handle should be valid.

Example

```
// This code shows an example of how to cancel all IRPs.

void MyIRPCallback(USB_DEVICE_IRP * irp)
{
    // Check if this is setup command

    if(irp->status == USB_DEVICE_IRP_STATUS_SETUP)
    {
        if(IsSetupCommandSupported(irp->data) == false)
        {
            // This means that this setup command is not
            // supported. Stall the some related endpoint and cancel all
            // queue IRPs.

            DRV_USBHS_DEVICE_EndpointStall(handle, ep);
            DRV_USBHS_DEVICE_IRPCancelAll(handle, ep);
        }
     }
 }
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).
endpointAndDirection	Specifies the endpoint and direction.

Function

USB_ERROR DRV_USBHS_DEVICE_IRPCancelAll

```
(
    DRV_HANDLE client,
USB_ENDPOINT endpointAndDirection
);
```

DRV_USBHS_DEVICE_IRPSubmit Function

This function submits an I/O Request Packet (IRP) for processing to the Hi-Speed USB Driver.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_DEVICE_IRPSubmit(DRV_HANDLE client, USB_ENDPOINT endpointAndDirection, USB_DEVICE_IRP *
irp);
```

Returns

- USB_ERROR_NONE - if the IRP was submitted successful.
- USB_ERROR_IRP_SIZE_INVALID - if the size parameter of the IRP is not correct.
- USB_ERROR_PARAMETER_INVALID - If the client handle is not valid.
- USB_ERROR_ENDPOINT_NOT_CONFIGURED - If the endpoint is not enabled.
- USB_ERROR_DEVICE_ENDPOINT_INVALID - The specified endpoint is not valid.
- USB_ERROR_OSAL_FUNCTION - An OSAL call in the function did not complete successfully.

Description

This function submits an I/O Request Packet (IRP) for processing to the USB Driver. The IRP allows a client to send and receive data from the USB Host. The data will be sent or received through the specified endpoint. The direction of the data transfer is indicated by the direction flag in the endpointAndDirection parameter. Submitting an IRP arms the endpoint to either send data to or receive data from the Host. If an IRP is already being processed on the endpoint, the subsequent IRP submit operation will be queued. The contents of the IRP (including the application buffers) should not be changed until the IRP has been processed.

Particular attention should be paid to the size parameter of IRP. The following should be noted:

- The size parameter while sending data to the Host can be less than, greater than, equal to, or be an exact multiple of the maximum packet size for the endpoint. The maximum packet size for the endpoint determines the number of transactions required to process the IRP.
- If the size parameter, while sending data to the Host is less than the maximum packet size, the transfer will complete in one transaction.
- If the size parameter, while sending data to the Host is greater than the maximum packet size, the IRP will be processed in multiple transactions.
- If the size parameter, while sending data to the Host is equal to or an exact multiple of the maximum packet size, the client can optionally ask the driver to send a Zero Length Packet(ZLP) by specifying the USB_DEVICE_IRP_FLAG_DATA_COMPLETE flag as the flag parameter.
- The size parameter, while receiving data from the Host must be an exact multiple of the maximum packet size of the endpoint. If this is not the case, the driver will return a USB_ERROR_IRP_SIZE_INVALID result. If while processing the IRP, the driver receives less than maximum packet size or a ZLP from the Host, the driver considers the IRP as processed. The size parameter at this point contains the actual amount of data received from the Host. The IRP status is returned as USB_DEVICE_IRP_STATUS_COMPLETED_SHORT.
- If a ZLP needs to be sent to Host, the IRP size should be specified as 0 and the flag parameter should be set as USB_DEVICE_IRP_FLAG_DATA_COMPLETE.
- If the IRP size is an exact multiple of the endpoint size, the client can request the driver to not send a ZLP by setting the flag parameter to USB_DEVICE_IRP_FLAG_DATA_PENDING. This flag indicates that there is more data pending in this transfer.
- Specifying a size less than the endpoint size along with the USB_DEVICE_IRP_FLAG_DATA_PENDING flag will cause the driver to return a USB_ERROR_IRP_SIZE_INVALID.
- If the size is greater than but not a multiple of the endpoint size, and the flag is specified as USB_DEVICE_IRP_FLAG_DATA_PENDING, the driver will send multiple of endpoint size number of bytes. For example, if the IRP size is 130 and the endpoint size if 64, the number of bytes sent will 128.

Remarks

This function can be called from the ISR of the USB module to associated with the client.

Preconditions

The Client handle should be valid.

Example

```
// The following code shows an example of how to schedule a IRP to send data
// from a device to the Host. Assume that the max packet size is 64 and
```

```
// and this data needs to sent over Endpoint 1. In this example, the
// transfer is processed as three transactions of 64, 64 and 2 bytes.

USB_ENDPOINT ep;
USB_DEVICE_IRP irp;

ep.direction = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

irp.data = myDataBufferToSend;
irp.size = 130;
irp.flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
irp.callback = MyIRPCompletionCallback;
irp.referenceData = (uintptr_t)&myDeviceLayerObj;

if (DRV_USBHS_DEVICE_IRPSubmit(handle, ep, &irp) != USB_ERROR_NONE)
{
    // This means there was an error.
}
else
{
    // The status of the IRP can be checked.
    while(irp.status != USB_DEVICE_IRP_STATUS_COMPLETED)
    {
        // Wait or run a task function.
    }
}

// The following code shows how the client can request
// the driver to send a ZLP when the size is an exact multiple of
// endpoint size.

irp.data = myDataBufferToSend;
irp.size = 128;
irp.flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
irp.callback = MyIRPCompletionCallback;
irp.referenceData = (uintptr_t)&myDeviceLayerObj;

// Note that while  receiving data from the Host, the size should be an
// exact multiple of the maximum packet size of the endpoint. In the
// following example, the DRV_USBHS_DEVICE_IRPSubmit function will return a
// USB_DEVICE_IRP_SIZE_INVALID value.

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_HOST_TO_DEVICE, 1);

irp.data = myDataBufferToSend;
irp.size = 60; // THIS SIZE IS NOT CORRECT
irp.flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
irp.callback = MyIRPCompletionCallback;
irp.referenceData = (uintptr_t)&myDeviceLayerObj;
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).
endpointAndDirection	Specifies the endpoint and direction.
irp	Pointer to the IRP to be added to the queue for processing.

Function

USB_ERROR DRV_USBHS_DEVICE_IRPSubmit

(

DRV_HANDLE client,

USB_ENDPOINT endpointAndDirection,

USB_DEVICE_IRP * irp

);

DRV_USBHS_DEVICE_RemoteWakeupStart Function

This function causes the device to start Remote Wakeup Signalling on the bus.

File

drv_usbhs.h

C

```
void DRV_USBHS_DEVICE_RemoteWakeupStart(DRV_HANDLE handle);
```

Returns

None.

Description

This function causes the device to start Remote Wakeup Signalling on the bus. This function should be called when the device, presently placed in suspend mode by the Host, wants to be wakeup. Note that the device can do this only when the Host has enabled the device's Remote Wakeup capability.

Remarks

None.

Preconditions

The handle should be valid.

Example

```
DRV_HANDLE handle;

// If the Host has enabled the Remote Wakeup capability, and if the device
// is in suspend mode, then start Remote Wakeup signaling.

if(deviceIsSuspended && deviceRemoteWakeupEnabled)
{
    DRV_USBHS_DEVICE_RemoteWakeupStart(handle);
}
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).

Function

void DRV_USBHS_DEVICE_RemoteWakeupStart(DRV_HANDLE handle);

DRV_USBHS_DEVICE_RemoteWakeupStop Function

This function causes the device to stop the Remote Wakeup Signalling on the bus.

File

drv_usbhs.h

C

```
void DRV_USBHS_DEVICE_RemoteWakeupStop(DRV_HANDLE handle);
```

Returns

None.

Description

This function causes the device to stop Remote Wakeup Signalling on the bus. This function should be called after the DRV_USBHS_DEVICE_RemoteWakeupStart function was called to start the Remote Wakeup signaling on the bus.

Remarks

This function should be 1 to 15 milliseconds after the DRV_USBHS_DEVICE_RemoteWakeupStart function was called.

Preconditions

The handle should be valid. The DRV_USBHS_DEVICE_RemoteWakeupStart function was called to start the Remote Wakeup signaling on the bus.

Example

```
DRV_HANDLE handle;

// If the Host has enabled the Remote Wakeup capability, and if the device
// is in suspend mode, then start Remote Wakeup signaling. Wait for 10
// milliseconds and then stop the Remote Wakeup signaling

if(deviceIsSuspended && deviceRemoteWakeupEnabled)
{
    DRV_USBHS_DEVICE_RemoteWakeupStart(handle);
    DelayMilliSeconds(10);
    DRV_USBHS_DEVICE_RemoteWakeupStop(handle);
}
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).

Function

void DRV_USBHS_DEVICE_RemoteWakeupStop(DRV_HANDLE handle);

DRV_USBHS_DEVICE_SOFNumberGet Function

This function will return the USB SOF packet number.

File

drv_usbhs.h

C

```
uint16_t DRV_USBHS_DEVICE_SOFNumberGet(DRV_HANDLE handle);
```

Returns

The SOF packet number.

Description

This function will return the USB SOF packet number..

Remarks

None.

Preconditions

This function will return a valid value only when the device is attached to the bus. The SOF packet count will not increment if the bus is suspended.

Example

```
// This code shows how the DRV_USBHS_DEVICE_SOFNumberGet function is called
// to read the current SOF number.

DRV_HANDLE handle;
uint16_t sofNumber;

sofNumber = DRV_USBHS_DEVICE_SOFNumberGet(handle);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).

Function

uint16_t DRV_USBHS_DEVICE_SOFNumberGet(DRV_HANDLE handle);

DRV_USBHS_DEVICE_TestModeEnter Function

This function enables the specified USB 2.0 Test Mode.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_DEVICE_TestModeEnter(DRV_HANDLE handle, USB_TEST_MODE_SELECTORS testMode);
```

Returns

- USB_ERROR_NONE - The function executed successfully.
- USB_ERROR_PARAMETER_INVALID - The handle or the value of testMode parameter is not valid.

Description

This function causes the device to enter the specified USB 2.0 defined test mode. It is called in response to Set Feature command from the host. The wValue field of this command specifies the Test Mode to enter. The USB module will perform the action identified by the testMode parameter.

Remarks

This function should be called only when the USB device has attached to the Host at High speed and only in response to the Set Feature command from the Host.

Preconditions

The handle should be valid.

Example

```
DRV_HANDLE handle;

// This code shows how the DRV_USBHS_DEVICE_TestModeEnter function is
// called to make the USB device enter the Test_J test mode.

DRV_USBHS_DEVICE_TestModeEnter(handle, USB_TEST_MODE_SELECTOR_TEST_J);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).
testMode	This parameter identifies the USB 2.0 specification test mode (see table 9-7 of the USB 2.0 specification).

Function

USB_ERROR DRV_USBHS_DEVICE_TestModeEnter

(

DRV_HANDLE handle,

USB_TEST_MODE_SELECTORS testMode

);

DRV_USBHS_DEVICE_TestModeExit Function

This function disables the specified USB 2.0 Test Mode.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_DEVICE_TestModeExit(DRV_HANDLE handle, USB_TEST_MODE_SELECTORS testMode);
```

Returns

- USB_ERROR_NONE - The function executed successfully.
- USB_ERROR_PARAMETER_INVALID - The handle or the value of testMode parameter is not valid.

Description

This function causes the device to stop the specified USB 2.0 defined test mode. This function can be called after calling the DRV_USBHS_DEVICE_TestModeEnter function to stop the test mode execution.

Remarks

None.

Preconditions

The handle should be valid.

Example

```
DRV_HANDLE handle;

// This code shows how the DRV_USBHS_DEVICE_TestModeEnter function is
// called to make the USB device enter the Test_J test mode.

DRV_USBHS_DEVICE_TestModeEnter(handle, USB_TEST_MODE_SELECTOR_TEST_J);

// Now the DRV_USBHS_DEVICE_TestModeExit function is called to stop the
// Test_J test mode.

DRV_USBHS_DEVICE_TestModeExit(handle, USB_TEST_MODE_SELECTOR_TEST_J);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).
testMode	This parameter identifies the USB 2.0 specification test mode (see table 9-7 of the USB 2.0 specification).

Function

USB_ERROR DRV_USBHS_DEVICE_TestModeExit
(
    DRV_HANDLE handle,
USB_TEST_MODE_SELECTORS testMode
);

d) Host Mode Operation Functions

DRV_USBHS_HOST_EventsDisable Function

Disables Host mode events.

File

drv_usbhs.h

C

```
bool DRV_USBHS_HOST_EventsDisable(DRV_HANDLE handle);
```

Returns

- true - Driver event generation was enabled when this function was called.
- false - Driver event generation was not enabled when this function was called.

Description

This function disables the Host mode events. This function is called by the Host Layer when it wants to execute code atomically.

Remarks

None.

Preconditions

The handle should be valid.

Example

```
// This code shows how the DRV_USBHS_HOST_EventsDisable and
// DRV_USBHS_HOST_EventsEnable function can be called to disable and enable
// events.

DRV_HANDLE driverHandle;
bool eventsWereEnabled;

// Disable the driver events.
eventsWereEnabled = DRV_USBHS_HOST_EventsDisable(driverHandle);

// Code in this region will not be interrupted by driver events.

// Enable the driver events.
DRV_USBHS_HOST_EventsEnable(driverHandle, eventsWereEnabled);
```

Parameters

Parameters	Description
handle	Client's driver handle (returned from DRV_USBHS_Open function).

Function

bool DRV_USBHS_HOST_EventsDisable

(

DRV_HANDLE handle

);

DRV_USBHS_HOST_EventsEnable Function

Restores the events to the specified the original value.

File

drv_usbhs.h

C

```
void DRV_USBHS_HOST_EventsEnable(DRV_HANDLE handle, bool eventContext);
```

Returns

None.

Description

This function will restore the enable disable state of the events. The eventRestoreContext parameter should be equal to the value returned by the DRV_USBHS_HOST_EventsDisable function.

Remarks

None.

Preconditions

The handle should be valid.

Example

```
// This code shows how the DRV_USBHS_HOST_EventsDisable and
// DRV_USBHS_HOST_EventsEnable function can be called to disable and enable
// events.

DRV_HANDLE driverHandle;
```

```
bool eventsWereEnabled;

// Disable the driver events.
eventsWereEnabled = DRV_USBHS_HOST_EventsDisable(driverHandle);

// Code in this region will not be interrupted by driver events.

// Enable the driver events.
DRV_USBHS_HOST_EventsEnable(driverHandle, eventsWereEnabled);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).
eventRestoreContext	Value returned by the DRV_USBHS_HOST_EventsDisable function.

Function

void DRV_USBHS_HOST_EventsEnable
(
DRV_HANDLE handle
bool eventRestoreContext
);

DRV_USBHS_HOST_IRPCancel Function

Cancels the specified IRP.

File

drv_usbhs.h

C

```
void DRV_USBHS_HOST_IRPCancel(USB_HOST_IRP * inputIRP);
```

Returns

None.

Description

This function attempts to cancel the specified IRP. If the IRP is queued and its processing has not started, it will be cancelled successfully. If the IRP in progress, the ongoing transaction will be allowed to complete.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how a submitted IRP can be cancelled.

USB_HOST_IRP irp;
USB_ERROR result;
USB_HOST_PIPE_HANDLE controlPipe;
USB_SETUP_PACKET setup;
uint8_t controlTransferData[32];

irp.setup = setup;
irp.data = controlTransferData;
irp.size = 32;
irp.flags = USB_HOST_IRP_FLAG_NONE ;
irp.userData = &someApplicationObject;
irp.callback = IRP_Callback;

DRV_USBHS_HOST_IRPSubmit(controlPipeHandle, &irp);
```

```
// Additional application logic may come here. This logic may decide to
// cancel the submitted IRP.

DRV_USBHS_HOST_IRPCancel(&irp);
```

Parameters

Parameters	Description
inputIRP	Pointer to the IRP to cancel.

Function

void DRV_USBHS_HOST_IRPCancel(USB_HOST_IRP * inputIRP);

DRV_USBHS_HOST_IRPSubmit Function

Submits an IRP on a pipe.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_HOST_IRPSubmit(DRV_USBHS_HOST_PIPE_HANDLE hPipe, USB_HOST_IRP * pinputIRP);
```

Returns

- USB_ERROR_NONE - The IRP was submitted successfully.
- USB_ERROR_PARAMETER_INVALID - The pipe handle is not valid.
- USB_ERROR_OSAL_FUNCTION - An error occurred in an OSAL function called in this function.

Description

This function submits an IRP on the specified pipe. The IRP will be added to the queue and will be processed in turn. The data will be transferred on the bus based on the USB bus scheduling rules. When the IRP has been processed, the callback function specified in the IRP will be called. The IRP status will be updated to reflect the completion status of the IRP.

Remarks

An IRP can also be submitted in an IRP callback function.

Preconditions

The pipe handle should be valid.

Example

```
// The following code shows an example of how the host layer populates
// the IRP object and then submits it. IRP_Callback function is called when an
// IRP has completed processing. The status of the IRP at completion can be
// checked in the status flag. The size field of the irp will contain the amount
// of data transferred.

void IRP_Callback(USB_HOST_IRP * irp)
{
    // irp is pointing to the IRP for which the callback has occurred. In most
    // cases this function will execute in an interrupt context. The application
    // should not perform any hardware access or interrupt un-safe operations in
    // this function.

    switch(irp->status)
    {
        case USB_HOST_IRP_STATUS_ERROR_UNKNOWN:
            // IRP was terminated due to an unknown error
            break;

        case USB_HOST_IRP_STATUS_ABORTED:
            // IRP was terminated by the application
            break;

        case USB_HOST_IRP_STATUS_ERROR_BUS:
            // IRP was terminated due to a bus error
```

```
            break;

        case USB_HOST_IRP_STATUS_ERROR_DATA:
            // IRP was terminated due to data error
            break;

        case USB_HOST_IRP_STATUS_ERROR_NAK_TIMEOUT:
            // IRP was terminated because of a NAK timeout
            break;

        case USB_HOST_IRP_STATUS_ERROR_STALL:
            // IRP was terminated because of a device sent a STALL
            break;

        case USB_HOST_IRP_STATUS_COMPLETED:
            // IRP has been completed
            break;

        case USB_HOST_IRP_STATUS_COMPLETED_SHORT:
            // IRP has been completed but the amount of data processed was less
            // than requested.
            break;

        default:
            break;
    }
}

// In the following code snippet the a control transfer IRP is submitted to a
// control pipe. The setup parameter of the IRP points to the Setup command of
// the control transfer. The direction of the data stage is specified by the
// Setup packet.

USB_HOST_IRP irp;
USB_ERROR result;
USB_HOST_PIPE_HANDLE controlPipe;
USB_SETUP_PACKET setup;
uint8_t controlTransferData[32];

irp.setup = setup;
irp.data = controlTransferData;
irp.size = 32;
irp.flags = USB_HOST_IRP_FLAG_NONE ;
irp.userData = &someApplicationObject;
irp.callback = IRP_Callback;

result = DRV_USBHS_HOST_IRPSubmit(controlPipeHandle, &irp);
```

Parameters

Parameters	Description
hPipe	Handle to the pipe to which the IRP has to be submitted.
pInputIRP	Pointer to the IRP.

Function

```
USB_ERROR DRV_USBHS_HOST_IRPSubmit
(
    DRV_USBHS_HOST_PIPE_HANDLE  hPipe,
USB_HOST_IRP * pInputIRP
);
```

DRV_USBHS_HOST_PipeClose Function

Closes an open pipe.

File

drv_usbhs.h

C

```
void DRV_USBHS_HOST_PipeClose(DRV_USBHS_HOST_PIPE_HANDLE pipeHandle);
```

Returns

None.

Description

This function closes an open pipe. Any IRPs scheduled on the pipe will be aborted and IRP callback functions will be called with the status as DRV_USB_HOST_IRP_STATE_ABORTED. The pipe handle will become invalid and the pipe and will not accept IRPs.

Remarks

None.

Preconditions

The pipe handle should be valid.

Example

```
// This code shows how an open Host pipe can be closed.

DRV_HANDLE driverHandle;
DRV_USBHS_HOST_PIPE_HANDLE pipeHandle;

// Close the pipe.
DRV_USBHS_HOST_PipeClose(pipeHandle);
```

Parameters

Parameters	Description
pipeHandle	Handle to the pipe to close.

Function

void DRV_USBHS_HOST_PipeClose

(

DRV_USBHS_HOST_PIPE_HANDLE pipeHandle

);

DRV_USBHS_HOST_PipeSetup Function

Open a pipe with the specified attributes.

File

drv_usbhs.h

C

```
DRV_USBHS_HOST_PIPE_HANDLE DRV_USBHS_HOST_PipeSetup(DRV_HANDLE client, uint8_t deviceAddress, USB_ENDPOINT endpointAndDirection, uint8_t hubAddress, uint8_t hubPort, USB_TRANSFER_TYPE pipeType, uint8_t bInterval, uint16_t wMaxPacketSize, USB_SPEED speed);
```

Returns

- DRV_USB_HOST_PIPE_HANDLE_INVALID - The pipe could not be created.
- A valid Pipe Handle - The pipe was created successfully. This is an arbitrary value and will never be the same as DRV_USB_HOST_PIPE_HANDLE_INVALID.

Description

This function opens a communication pipe between the Host and the device endpoint. The transfer type and other attributes are specified through the function parameters. The driver does not check for available bus bandwidth, which should be done by the application (the USB Host Layer in this case)

Remarks

None.

Preconditions

The driver handle should be valid.

Example

```
// This code shows how the DRV_USBHS_HOST_PipeSetup function is called for
// create a communication pipe. In this example, Bulk pipe is created
// between the Host and a device. The Device address is 2 and the target
// endpoint on this device is 4 . The direction of the data transfer over
// this pipe is from the Host to the device. The device is connected to Port
// 1 of a Hub, whose USB address is 3. The maximum size of a transaction
// on this pipe is 64 bytes. This is a Bulk Pipe and hence the bInterval
// field is set to 0. The target device is operating at Full Speed.

DRV_HANDLE driverHandle;
DRV_USBHS_HOST_PIPE_HANDLE pipeHandle;

pipeHandle = DRV_USBHS_HOST_PipeSetup(driverHandle, 0x02, 0x14, 0x03, 0x01, USB_TRANSFER_TYPE_BULK, 0, 64,
USB_SPEED_FULL);

if(pipeHandle != DRV_USBHS_HOST_PIPE_HANDLE_INVALID)
{
    // The pipe was created successfully.
}
```

Parameters

Parameters	Description
client	Handle to the driver (returned from DRV_USBHS_Open function).
deviceAddress	USB Address of the device to connect to.
endpoint	Endpoint on the device to connect to.
hubAddress	Address of the hub to which this device is connected. If not connected to a hub, this value should be set to 0.
hubPort	Port number of the hub to which this device is connected.
pipeType	Transfer type of the pipe to open.
bInterval	Polling interval for periodic transfers. This should be specified as defined by the USB 2.0 Specification.
wMaxPacketSize	This should be set to the endpoint size reported by the device in its configuration descriptors. This defines the maximum size of the transaction in a transfer on this pipe.
speed	The speed of the pipe. This should match the speed at which the device connected to the Host.

Function

```
DRV_USBHS_HOST_PIPE_HANDLE DRV_USBHS_HOST_PipeSetup
(
    DRV_HANDLE client,
uint8_t deviceAddress,
USB_ENDPOINT endpointAndDirection,
uint8_t hubAddress,
uint8_t hubPort,
USB_TRANSFER_TYPE pipeType,
uint8_t bInterval,
uint16_t wMaxPacketSize,
USB_SPEED speed
);
```

e) Root Hub Functions

DRV_USBHS_HOST_ROOT_HUB_BusSpeedGet Function

This function returns the operating speed of the bus to which this root hub is connected.

File

drv_usbhs.h

C

```
USB_SPEED DRV_USBHS_HOST_ROOT_HUB_BusSpeedGet(DRV_HANDLE handle);
```

Returns

- USB_SPEED_HIGH - The Root hub is connected to a bus that is operating at High Speed.
- USB_SPEED_FULL - The Root hub is connected to a bus that is operating at Full Speed.

Description

This function returns the operating speed of the bus to which this root hub is connected.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how the DRV_USBHS_HOST_ROOT_HUB_BusSpeedGet function is
// called to know the operating speed of the bus to which this Root hub is
// connected.

DRV_HANDLE driverHandle;
USB_SPEED speed;

speed = DRV_USBHS_HOST_ROOT_HUB_BusSpeedGet(driverHandle);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).

Function

USB_SPEED DRV_USBHS_HOST_ROOT_HUB_BusSpeedGet(DRV_HANDLE handle);

DRV_USBHS_HOST_ROOT_HUB_Initialize Function

This function initializes the root hub driver.

File

drv_usbhs.h

C

```
void DRV_USBHS_HOST_ROOT_HUB_Initialize(DRV_HANDLE handle, USB_HOST_DEVICE_OBJ_HANDLE usbHostDeviceInfo);
```

Returns

None.

Description

This function initializes the root hub driver. It is called by the Host Layer at the time of processing the root hub devices. The Host Layer assigns a USB_HOST_DEVICE_INFO reference to this root hub driver. This identifies the relationship between the root hub and the Host Layer.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how the USB Host Layer calls the
// DRV_USBHS_HOST_ROOT_HUB_Initialize function. The usbHostDeviceInfo
// parameter is an arbitrary identifier assigned by the USB Host Layer. Its
// interpretation is opaque to the Root hub Driver.

DRV_HANDLE drvHandle;
USB_HOST_DEVICE_OBJ_HANDLE usbHostDeviceInfo = 0x10003000;

DRV_USBHS_HOST_ROOT_HUB_Initialize(drvHandle, usbHostDeviceInfo);
```

Parameters

Parameters	Description
handle	Handle to the driver.
usbHostDeviceInfo	Reference provided by the Host.

Function

void DRV_USBHS_HOST_ROOT_HUB_Initialize

(

DRV_HANDLE handle,

USB_HOST_DEVICE_OBJ_HANDLE usbHostDeviceInfo,

)

DRV_USBHS_HOST_ROOT_HUB_MaximumCurrentGet Function

Returns the maximum amount of current that this root hub can provide on the bus.

File

drv_usbhs.h

C

```
uint32_t DRV_USBHS_HOST_ROOT_HUB_MaximumCurrentGet(DRV_HANDLE handle);
```

Returns

Returns the maximum current (in milliamperes) that the root hub can supply.

Description

This function returns the maximum amount of current that this root hub can provide on the bus.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how the DRV_USBHS_HOST_ROOT_HUB_MaximumCurrentGet
// function is called to obtain the maximum VBUS current that the Root hub
// can supply.

DRV_HANDLE driverHandle;
uint32_t currentMilliAmperes;

currentMilliAmperes = DRV_USBHS_HOST_ROOT_HUB_MaximumCurrentGet(driverHandle);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).

Function

uint32_t DRV_USBHS_HOST_ROOT_HUB_MaximumCurrentGet(DRV_HANDLE);

DRV_USBHS_HOST_ROOT_HUB_OperationEnable Function

This function enables or disables root hub operation.

File

drv_usbhs.h

C

```
void DRV_USBHS_HOST_ROOT_HUB_OperationEnable(DRV_HANDLE handle, bool enable);
```

Returns

None.

Description

This function enables or disables root hub operation. When enabled, the root hub will detect devices attached to the port and will request the Host Layer to enumerate the device. This function is called by the Host Layer when it is ready to receive enumeration requests from the Host. If the operation is disabled, the root hub will not detect attached devices.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how the DRV_USBHS_HOST_ROOT_HUB_OperationEnable and the
// DRV_USBHS_HOST_ROOT_HUB_OperationIsEnabled functions are called to enable
// the Root hub operation.

DRV_HANDLE driverHandle;

// Enable Root hub operation.
DRV_USBHS_HOST_ROOT_HUB_OperationEnable(driverHandle);

// Wait till the Root hub operation is enabled.
if(DRV_USBHS_HOST_ROOT_HUB_OperationIsEnabled(driverHandle) == false)
{
    // The operation has not completed. Call the
    // DRV_USBHS_HOST_ROOT_HUB_OperationIsEnabled function again to check if
    // the operation has completed. Note that the DRV_USBHS_Tasks function
    // must be allowed to run at periodic intervals to allow the enable
    // operation to completed.
}
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).
enable	If this is set to true, root hub operation is enabled. If this is set to false, root hub operation is disabled.

Function

void DRV_USBHS_HOST_ROOT_HUB_OperationEnable

(

DRV_HANDLE handle,

bool enable

);

DRV_USBHS_HOST_ROOT_HUB_OperationIsEnabled Function

Returns the operation enabled status of the root hub.

File

drv_usbhs.h

C

```
bool DRV_USBHS_HOST_ROOT_HUB_OperationIsEnabled(DRV_HANDLE handle);
```

Returns

- true - Root hub operation is enabled.
- false - Root hub operation is not enabled.

Description

This function returns true if the DRV_USBHS_HOST_ROOT_HUB_OperationEnable function has completed enabling the Host.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how the DRV_USBHS_HOST_ROOT_HUB_OperationEnable and the
// DRV_USBHS_HOST_ROOT_HUB_OperationIsEnabled functions are called to enable
// the Root hub operation.

DRV_HANDLE driverHandle;

// Enable Root hub operation.
DRV_USBHS_HOST_ROOT_HUB_OperationEnable(driverHandle);

// Wait till the Root hub operation is enabled.
if(DRV_USBHS_HOST_ROOT_HUB_OperationIsEnabled(driverHandle) == false)
{
    // The operation has not completed. Call the
    // DRV_USBHS_HOST_ROOT_HUB_OperationIsEnabled function again to check if
    // the operation has completed. Note that the DRV_USBHS_Tasks function
    // must be allowed to run at periodic intervals to allow the enable
    // operation to completed.
}
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).

Function

bool DRV_USBHS_HOST_ROOT_HUB_OperationIsEnabled(DRV_HANDLE handle);

DRV_USBHS_HOST_ROOT_HUB_PortNumbersGet Function

Returns the number of ports this root hub contains.

File

drv_usbhs.h

C

```
uint8_t DRV_USBHS_HOST_ROOT_HUB_PortNumbersGet(DRV_HANDLE handle);
```

Returns

This function will always return 1.

Description

This function returns the number of ports that this root hub contains.

Remarks

None.

Preconditions

None.

Example

```
// This code shows how DRV_USBHS_HOST_ROOT_HUB_PortNumbersGet function can
// be called to obtain the number of Root hub ports.

DRV_HANDLE driverHandle;
uint8_t nPorts;

nPorts = DRV_USBHS_HOST_ROOT_HUB_PortNumbersGet(driverHandle);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).

Function

uint8_t DRV_USBHS_HOST_ROOT_HUB_PortNumbersGet(DRV_HANDLE handle);

DRV_USBHS_HOST_ROOT_HUB_PortReset Function

Resets the specified root hub port.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_HOST_ROOT_HUB_PortReset(DRV_HANDLE handle, uint8_t port);
```

Returns

None.

Description

This function resets the root hub port. The reset duration is defined by DRV_USBHS_ROOT_HUB_RESET_DURATION. The status of the reset signaling can be checked using the DRV_USBHS_ROOT_HUB_PortResetIsComplete function.

Remarks

The root hub on the PIC32MZ USB controller contains only one port - port 0.

Preconditions

None.

Example

```
// This code shows how the DRV_USB_HOST_ROOT_HUB_PortReset and the
// DRV_USBHS_ROOT_HUB_PortResetIsComplete functions are called to complete a
// port reset sequence.

DRV_HANDLE driverHandle;

// Reset Port 0.
DRV_USB_HOST_ROOT_HUB_PortReset(driverHandle, 0);

// Check if the Reset operation has completed.
if(DRV_USBHS_ROOT_HUB_PortResetIsComplete(driverHandle, 0) == false)
{
    // This means that the Port Reset operation has not completed yet. The
    // DRV_USBHS_ROOT_HUB_PortResetIsComplete function should be called
```

```
    // again after some time to check the status.
}
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).
port	Port to reset.

Function

void DRV_USBHS_ROOT_HUB_PortReset(DRV_HANDLE handle, uint8_t port);

DRV_USBHS_HOST_ROOT_HUB_PortResetIsComplete Function

Returns true if the root hub has completed the port reset operation.

File

drv_usbhs.h

C

```
bool DRV_USBHS_HOST_ROOT_HUB_PortResetIsComplete(DRV_HANDLE handle, uint8_t port);
```

Returns

- true - The reset signaling has completed.
- false - The reset signaling has not completed.

Description

This function returns true if the port reset operation has completed. It should be called after the DRV_USB_HOST_ROOT_HUB_PortReset function to check if the reset operation has completed.

Remarks

The root hub on this particular hardware only contains one port - port 0.

Preconditions

None.

Example

```
// This code shows how the DRV_USB_HOST_ROOT_HUB_PortReset and the
// DRV_USBHS_ROOT_HUB_PortResetIsComplete functions are called to complete a
// port reset sequence.

DRV_HANDLE driverHandle;

// Reset Port 0.
DRV_USB_HOST_ROOT_HUB_PortReset(driverHandle, 0);

// Check if the Reset operation has completed.
if(DRV_USBHS_ROOT_HUB_PortResetIsComplete(driverHandle, 0) == false)
{
    // This means that the Port Reset operation has not completed yet. The
    // DRV_USBHS_ROOT_HUB_PortResetIsComplete function should be called
    // again after some time to check the status.
}
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).
port	Port to check

Function

bool DRV_USBHS_ROOT_HUB_PortResetIsComplete

(

DRV_HANDLE handle,

uint8_t port
);

DRV_USBHS_HOST_ROOT_HUB_PortResume Function

Resumes the specified root hub port.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_HOST_ROOT_HUB_PortResume(DRV_HANDLE handle, uint8_t port);
```

Returns

- USB_ERROR_NONE - The function executed successfully.
- USB_ERROR_PARAMETER_INVALID - The driver handle is not valid or the port number does not exist.

Description

This function resumes the root hub. The resume duration is defined by DRV_USBHS_ROOT_HUB_RESUME_DURATION. The status of the resume signaling can be checked using the DRV_USBHS_ROOT_HUB_PortResumeIsComplete function.

Remarks

The root hub on this particular hardware only contains one port - port 0.

Preconditions

None.

Example

```
// This code shows how the DRV_USBHS_HOST_ROOT_HUB_PortResume function is
// called to resume the specified port.

DRV_HANDLE driverHandle;

// Resume Port 0.
DRV_USBHS_HOST_ROOT_HUB_PortResume(driverHandle, 0);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).
port	Port to resume.

Function

USB_ERROR DRV_USBHS_HOST_ROOT_HUB_PortResume
(
DRV_HANDLE handle,
uint8_t port
);

DRV_USBHS_HOST_ROOT_HUB_PortSpeedGet Function

Returns the speed of at which the port is operating.

File

drv_usbhs.h

C

```
USB_SPEED DRV_USBHS_HOST_ROOT_HUB_PortSpeedGet(DRV_HANDLE handle, uint8_t port);
```

Returns

- USB_SPEED_ERROR - This value is returned if the driver handle is not or if the speed information is not available or if the specified port is not valid.

- USB_SPEED_HIGH - A High Speed device has been connected to the port.
- USB_SPEED_FULL - A Full Speed device has been connected to the port.
- USB_SPEED_LOW - A Low Speed device has been connected to the port.

Description

This function returns the speed at which the port is operating.

Remarks

The root hub on this particular hardware only contains one port - port 0.

Preconditions

None.

Example

```
// This code shows how the DRV_USBHS_HOST_ROOT_HUB_PortSpeedGet function is
// called to know the operating speed of the port. This also indicates the
// operating speed of the device connected to this port.

DRV_HANDLE driverHandle;
USB_SPEED speed;

speed = DRV_USBHS_HOST_ROOT_HUB_PortSpeedGet(driverHandle, 0);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).
port	Port number of the port to be analyzed..

Function

USB_SPEED DRV_USBHS_HOST_ROOT_HUB_PortSpeedGet

(

DRV_HANDLE handle,

uint8_t port

);

DRV_USBHS_HOST_ROOT_HUB_PortSuspend Function

Suspends the specified root hub port.

File

drv_usbhs.h

C

```
USB_ERROR DRV_USBHS_HOST_ROOT_HUB_PortSuspend(DRV_HANDLE handle, uint8_t port);
```

Returns

- USB_ERROR_NONE - The function executed successfully.
- USB_ERROR_PARAMETER_INVALID - The driver handle is not valid or the port number does not exist.

Description

This function suspends the root hub port.

Remarks

The root hub on this particular hardware only contains one port - port 0.

Preconditions

None.

Example

```
// This code shows how the DRV_USBHS_HOST_ROOT_HUB_PortSuspend function is
// called to suspend the specified port.
```

```
DRV_HANDLE driverHandle;

// Suspend Port 0.
DRV_USBHS_HOST_ROOT_HUB_PortSuspend(driverHandle, 0);
```

Parameters

Parameters	Description
handle	Handle to the driver (returned from DRV_USBHS_Open function).
port	Port to suspend.

Function

USB_ERROR DRV_USBHS_ROOT_HUB_PortSuspend(DRV_HANDLE handle, uint8_t port);

f) Data Types and Constants

DRV_USBHS_EVENT Enumeration

Identifies the different events that the Hi-Speed USB Driver provides.

File

drv_usbhs.h

C

```
typedef enum {
  DRV_USBHS_EVENT_ERROR = DRV_USB_EVENT_ERROR,
  DRV_USBHS_EVENT_RESET_DETECT = DRV_USB_EVENT_RESET_DETECT,
  DRV_USBHS_EVENT_RESUME_DETECT = DRV_USB_EVENT_RESUME_DETECT,
  DRV_USBHS_EVENT_IDLE_DETECT = DRV_USB_EVENT_IDLE_DETECT,
  DRV_USBHS_EVENT_STALL = DRV_USB_EVENT_STALL,
  DRV_USBHS_EVENT_SOF_DETECT = DRV_USB_EVENT_SOF_DETECT,
  DRV_USBHS_EVENT_DEVICE_SESSION_VALID = DRV_USB_EVENT_DEVICE_SESSION_VALID,
  DRV_USBHS_EVENT_DEVICE_SESSION_INVALID = DRV_USB_EVENT_DEVICE_SESSION_INVALID
} DRV_USBHS_EVENT;
```

Members

Members	Description
DRV_USBHS_EVENT_ERROR = DRV_USB_EVENT_ERROR	Bus error occurred and was reported
DRV_USBHS_EVENT_RESET_DETECT = DRV_USB_EVENT_RESET_DETECT	Host has issued a device reset
DRV_USBHS_EVENT_RESUME_DETECT = DRV_USB_EVENT_RESUME_DETECT	Resume detected while USB in suspend mode
DRV_USBHS_EVENT_IDLE_DETECT = DRV_USB_EVENT_IDLE_DETECT	Idle detected
DRV_USBHS_EVENT_STALL = DRV_USB_EVENT_STALL	Stall handshake has occurred
DRV_USBHS_EVENT_SOF_DETECT = DRV_USB_EVENT_SOF_DETECT	Device received SOF operation
DRV_USBHS_EVENT_DEVICE_SESSION_VALID = DRV_USB_EVENT_DEVICE_SESSION_VALID	VBUS voltage had Session valid
DRV_USBHS_EVENT_DEVICE_SESSION_INVALID = DRV_USB_EVENT_DEVICE_SESSION_INVALID	Session Invalid

Description

Hi-Speed USB Driver Events Enumeration.

This enumeration identifies the different events that are generated by the Hi-Speed USB Driver.

Remarks

None.

DRV_USBHS_EVENT_CALLBACK Type

Type of the Hi-Speed USB Driver event callback function.

File

drv_usbhs.h

C

```
typedef void (* DRV_USBHS_EVENT_CALLBACK)(uintptr_t hClient, DRV_USBHS_EVENT eventType, void * eventData);
```

Returns

None.

Description

Type of the Hi-Speed USB Driver Event Callback Function.

Define the type of the Hi-Speed USB Driver event callback function. The client should register an event callback function of this type when it intends to receive events from the Hi-Speed USB Driver. The event callback function is registered using the DRV_USBHS_ClientEventCallBackSet function.

Remarks

None.

Parameters

Parameters	Description
hClient	Handle to the driver client that registered this callback function.
eventType	This parameter identifies the event that caused the callback function to be called.
eventData	Pointer to a data structure that is related to this event. This value will be NULL if the event has no related data.

DRV_USBHS_HOST_PIPE_HANDLE Type

Defines the Hi-Speed USB Driver Host Pipe Handle type.

File

drv_usbhs.h

C

```
typedef uintptr_t DRV_USBHS_HOST_PIPE_HANDLE;
```

Description

Hi-Speed USB Driver Host Pipe Handle.

This type definition defines the type of the Hi-Speed USB Driver Host Pipe Handle.

Remarks

None.

DRV_USBHS_INIT Structure

This type definition defines the Driver Initialization Data Structure.

File

drv_usbhs.h

C

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  USBHS_MODULE_ID usbID;
  bool stopInIdle;
  bool suspendInSleep;
  INT_SOURCE interruptSource;
  INT_SOURCE interruptSourceUSBDma;
  USB_SPEED operationSpeed;
```

```
  DRV_USBHS_OPMODES operationMode;
  void * endpointTable;
  uint32_t rootHubAvailableCurrent;
  DRV_USBHS_ROOT_HUB_PORT_POWER_ENABLE portPowerEnable;
  DRV_USBHS_ROOT_HUB_PORT_INDICATION portIndication;
  DRV_USBHS_ROOT_HUB_PORT_OVER_CURRENT_DETECT portOverCurrentDetect;
} DRV_USBHS_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System Module Initialization
USBHS_MODULE_ID usbID;	Identifies the USB peripheral to be used. This should be the USB PLIB module instance identifier.
bool stopInIdle;	This should be set to true if the USB module must stop operation in Idle mode
bool suspendInSleep;	This should be set to true if the USB module must suspend when the CPU enters Sleep mode.
INT_SOURCE interruptSource;	Specify the interrupt source for the USB module. This should be the interrupt source for the USB module instance specified in usbID.
INT_SOURCE interruptSourceUSBDma;	Specify the interrupt source for the USB module specific DMA controller. This should be the USB DMA interrupt source for the USB Module instance specified in usbID.
USB_SPEED operationSpeed;	Specify the operational speed of the USB module. This should always be set to USB_SPEED_FULL.
DRV_USBHS_OPMODES operationMode;	Specify the operation mode of the USB module. This specifies if the USB module should operate as a Device, Host, or both (Dual Role operation).
void * endpointTable;	A pointer to the endpoint descriptor table. This should be aligned at 512 byte address boundary. The size of the table is equal to the DRV_USBHS_ENDPOINT_TABLE_ENTRY_SIZE times the number of endpoints needed in the application.
uint32_t rootHubAvailableCurrent;	Root hub available current in milliamperes. This specifies the amount of current that root hub can provide to the attached device. This should be specified in mA. This is required when the driver is required to operate in Host mode.
DRV_USBHS_ROOT_HUB_PORT_POWER_ENABLE portPowerEnable;	When operating in Host mode, the application can specify a Root hub port enable function. This parameter should point to Root hub port enable function. If this parameter is NULL, it implies that the port is always enabled.
DRV_USBHS_ROOT_HUB_PORT_INDICATION portIndication;	When operating in Host mode, the application can specify a Root Port Indication. This parameter should point to the Root Port Indication function. If this parameter is NULL, it implies that Root Port Indication is not supported.
DRV_USBHS_ROOT_HUB_PORT_OVER_CURRENT_DETECT portOverCurrentDetect;	When operating is Host mode, the application can specify a Root Port Overcurrent detection. This parameter should point to the Root Port Indication function. If this parameter is NULL, it implies that Overcurrent detection is not supported.

Description

USB Device Driver Initialization Data.

This structure contains all the data necessary to initialize the Hi-Speed USB Driver. A pointer to a structure of this type, containing the desired initialization data, must be passed into the DRV_USBHS_Initialize function.

Remarks

None.

DRV_USBHS_OPMODES Enumeration

Identifies the operating modes supported by the Hi-Speed USB Driver.

File

drv_usbhs.h

C

```
typedef enum {
  DRV_USBHS_OPMODE_DUAL_ROLE = DRV_USB_OPMODE_DUAL_ROLE,
  DRV_USBHS_OPMODE_DEVICE = DRV_USB_OPMODE_DEVICE,
  DRV_USBHS_OPMODE_HOST = DRV_USB_OPMODE_HOST,
  DRV_USBHS_OPMODE_OTG = DRV_USB_OPMODE_OTG
} DRV_USBHS_OPMODES;
```

Members

Members	Description
DRV_USBHS_OPMODE_DUAL_ROLE = DRV_USB_OPMODE_DUAL_ROLE	The driver should be able to switch between Host and Device mode
DRV_USBHS_OPMODE_DEVICE = DRV_USB_OPMODE_DEVICE	The driver should support Device mode operation only
DRV_USBHS_OPMODE_HOST = DRV_USB_OPMODE_HOST	The driver should support Host mode operation only
DRV_USBHS_OPMODE_OTG = DRV_USB_OPMODE_OTG	The driver should support the OTG protocol

Description

USB Operating Modes Enumeration.

This enumeration identifies the operating modes supported by the Hi-Speed USB Driver.

Remarks

None.

DRV_USBHS_ROOT_HUB_PORT_INDICATION Type

USB Root hub Application Hooks (Port Indication).

File

drv_usbhs.h

C

```
typedef void (* DRV_USBHS_ROOT_HUB_PORT_INDICATION)(uint8_t port, USB_HUB_PORT_INDICATOR_COLOR color,
USB_HUB_PORT_INDICATOR_STATE state);
```

Description

USB Root hub Application Hooks (Port Indication).

A function of the type defined here should be provided to the driver root to implement Port Indication. The root hub driver calls this function when it needs to update the state of the port indication LEDs. The application can choose to implement the Amber and Green colors as one LED or two different LEDs. The root hub driver specifies the color and the indicator attribute (on, off or blinking) when it calls this function.

Remarks

None.

DRV_USBHS_ROOT_HUB_PORT_OVER_CURRENT_DETECT Type

USB Root hub Application Hooks (Port Overcurrent detection).

File

drv_usbhs.h

C

```
typedef bool (* DRV_USBHS_ROOT_HUB_PORT_OVER_CURRENT_DETECT)(uint8_t port);
```

Description

USB Root hub Application Hooks (Port Overcurrent detection).

A function of the type defined here should be provided to the driver root hub to check for port over current condition. This function will be called periodically by the root hub driver to check the Overcurrent status of the port. It should continue to return true while the Overcurrent condition exists on the port. It should return false when the Overcurrent condition does not exist.

Remarks

None.

DRV_USBHS_ROOT_HUB_PORT_POWER_ENABLE Type

USB Root hub Application Hooks (Port Power Enable/ Disable).

File

drv_usbhs.h

C

```
typedef void (* DRV_USBHS_ROOT_HUB_PORT_POWER_ENABLE)(uint8_t port, bool control);
```

Description

USB Root hub Application Hooks (Port Power Enable/ Disable).

A function of the type defined here should be provided to the driver root to control port power. The root hub driver will call this function when it needs to enable port power. If the application circuit contains a VBUS switch, the switch should be accessed and controlled by this function. If the enable parameter is true, the switch should be enabled and VBUS should be available on the port. If the enable parameter is false, the switch should be disabled and VBUS should not be available on the port.

Remarks

None.

DRV_USBHS_DEVICE_INTERFACE Macro

Hi-Speed USB Driver Device Mode Interface Functions.

File

drv_usbhs.h

C

```
#define DRV_USBHS_DEVICE_INTERFACE
```

Description

Hi-Speed USB Driver Device Mode Interface Functions.

The Device Controller Driver Interface member of the Device Stack Initialization data structure should be set to this value so that the Device Stack can access the Hi-Speed USB Driver Device Mode functions.

Remarks

None.

DRV_USBHS_HOST_INTERFACE Macro

Hi-Speed USB Driver Host Mode Interface Functions.

File

drv_usbhs.h

C

```
#define DRV_USBHS_HOST_INTERFACE
```

Description

Hi-Speed USB Driver Host Mode Interface Functions.

The Host Controller Driver Interface member of the Host Layer Initialization data structure should be set to this value so that the Host Layer can access the Hi-Speed USB Driver Host Mode functions.

Remarks

None.

DRV_USBHS_HOST_PIPE_HANDLE_INVALID Macro

Value of an Invalid Host Pipe Handle.

File

drv_usbhs.h

C

```
#define DRV_USBHS_HOST_PIPE_HANDLE_INVALID
```

Description

Hi-Speed USB Driver Invalid Host Pipe Handle.

This constant defines the value of an Invalid Host Pipe Handle.

Remarks

None.

DRV_USBHS_INDEX_0 Macro

Hi-Speed USB Driver Module Index 0 Definition.

File

drv_usbhs.h

C

```
#define DRV_USBHS_INDEX_0 0
```

Description

Hi-Speed USB Driver Module Index 0 Definition.

This constant defines the value of Hi-Speed USB Driver Index 0. The SYS_MODULE_INDEX parameter of the DRV_USBHS_Initialize and DRV_USBHS_Open functions should be set to this value to identify instance 0 of the driver.

Remarks

These constants should be used in place of hard-coded numeric literals and should be passed into the DRV_USBHS_Initialize and DRV_USBHS_Open functions to identify the driver instance in use. These are not indicative of the number of modules that are actually supported by the microcontroller.

Files

Files

Name	Description
drv_usbhs.h	PIC32MZ USB Module Driver Interface File
drv_usbhs_config_template.h	Hi-Speed USB (USBHS) Driver Configuration Template.

Description

drv_usbhs.h

PIC32MZ USB Module Driver Interface File

Enumerations

	Name	Description
	DRV_USBHS_EVENT	Identifies the different events that the Hi-Speed USB Driver provides.
	DRV_USBHS_OPMODES	Identifies the operating modes supported by the Hi-Speed USB Driver.

Functions

	Name	Description
	DRV_USBHS_ClientEventCallBackSet	This function sets up the event callback function that is invoked by the USB controller driver to notify the client of USB bus events.
	DRV_USBHS_Close	Closes an opened-instance of the Hi-Speed USB Driver.
	DRV_USBHS_DEVICE_AddressSet	This function will set the USB module address that is obtained from the Host.
	DRV_USBHS_DEVICE_Attach	This function will enable the attach signaling resistors on the D+ and D- lines thus letting the USB Host know that a device has been attached on the bus.
	DRV_USBHS_DEVICE_CurrentSpeedGet	This function will return the USB speed at which the device is operating.

	DRV_USBHS_DEVICE_Detach	This function will disable the attach signaling resistors on the D+ and D- lines thus letting the USB Host know that the device has detached from the bus.
	DRV_USBHS_DEVICE_EndpointDisable	This function disables an endpoint.
	DRV_USBHS_DEVICE_EndpointDisableAll	This function disables all provisioned endpoints.
	DRV_USBHS_DEVICE_EndpointEnable	This function enables an endpoint for the specified direction and endpoint size.
	DRV_USBHS_DEVICE_EndpointIsEnabled	This function returns the enable/disable status of the specified endpoint and direction.
	DRV_USBHS_DEVICE_EndpointIsStalled	This function returns the stall status of the specified endpoint and direction.
	DRV_USBHS_DEVICE_EndpointStall	This function stalls an endpoint in the specified direction.
	DRV_USBHS_DEVICE_EndpointStallClear	This function clears the stall on an endpoint in the specified direction.
	DRV_USBHS_DEVICE_IRPCancel	This function cancels the specific IRP that are queued and in progress at the specified endpoint.
	DRV_USBHS_DEVICE_IRPCancelAll	This function cancels all IRPs that are queued and in progress at the specified endpoint.
	DRV_USBHS_DEVICE_IRPSubmit	This function submits an I/O Request Packet (IRP) for processing to the Hi-Speed USB Driver.
	DRV_USBHS_DEVICE_RemoteWakeupStart	This function causes the device to start Remote Wakeup Signalling on the bus.
	DRV_USBHS_DEVICE_RemoteWakeupStop	This function causes the device to stop the Remote Wakeup Signalling on the bus.
	DRV_USBHS_DEVICE_SOFNumberGet	This function will return the USB SOF packet number.
	DRV_USBHS_DEVICE_TestModeEnter	This function enables the specified USB 2.0 Test Mode.
	DRV_USBHS_DEVICE_TestModeExit	This function disables the specified USB 2.0 Test Mode.
	DRV_USBHS_HOST_EventsDisable	Disables Host mode events.
	DRV_USBHS_HOST_EventsEnable	Restores the events to the specified the original value.
	DRV_USBHS_HOST_IRPCancel	Cancels the specified IRP.
	DRV_USBHS_HOST_IRPSubmit	Submits an IRP on a pipe.
	DRV_USBHS_HOST_PipeClose	Closes an open pipe.
	DRV_USBHS_HOST_PipeSetup	Open a pipe with the specified attributes.
	DRV_USBHS_HOST_ROOT_HUB_BusSpeedGet	This function returns the operating speed of the bus to which this root hub is connected.
	DRV_USBHS_HOST_ROOT_HUB_Initialize	This function initializes the root hub driver.
	DRV_USBHS_HOST_ROOT_HUB_MaximumCurrentGet	Returns the maximum amount of current that this root hub can provide on the bus.
	DRV_USBHS_HOST_ROOT_HUB_OperationEnable	This function enables or disables root hub operation.
	DRV_USBHS_HOST_ROOT_HUB_OperationIsEnabled	Returns the operation enabled status of the root hub.
	DRV_USBHS_HOST_ROOT_HUB_PortNumbersGet	Returns the number of ports this root hub contains.
	DRV_USBHS_HOST_ROOT_HUB_PortReset	Resets the specified root hub port.
	DRV_USBHS_HOST_ROOT_HUB_PortResetIsComplete	Returns true if the root hub has completed the port reset operation.
	DRV_USBHS_HOST_ROOT_HUB_PortResume	Resumes the specified root hub port.
	DRV_USBHS_HOST_ROOT_HUB_PortSpeedGet	Returns the speed of at which the port is operating.
	DRV_USBHS_HOST_ROOT_HUB_PortSuspend	Suspends the specified root hub port.
	DRV_USBHS_Initialize	Initializes the Hi-Speed USB Driver.
	DRV_USBHS_Open	Opens the specified Hi-Speed USB Driver instance and returns a handle to it.
	DRV_USBHS_Status	Provides the current status of the Hi-Speed USB Driver module.
	DRV_USBHS_Tasks	Maintains the driver's state machine when the driver is configured for Polled mode.
	DRV_USBHS_Tasks_ISR	Maintains the driver's Interrupt state machine and implements its ISR.
	DRV_USBHS_Tasks_ISR_USBDMA	Maintains the driver's DMA Transfer state machine and implements its ISR.

Macros

	Name	Description
	DRV_USBHS_DEVICE_INTERFACE	Hi-Speed USB Driver Device Mode Interface Functions.
	DRV_USBHS_HOST_INTERFACE	Hi-Speed USB Driver Host Mode Interface Functions.

	DRV_USBHS_HOST_PIPE_HANDLE_INVALID	Value of an Invalid Host Pipe Handle.
	DRV_USBHS_INDEX_0	Hi-Speed USB Driver Module Index 0 Definition.

Structures

	Name	Description
	DRV_USBHS_INIT	This type definition defines the Driver Initialization Data Structure.

Types

	Name	Description
	DRV_USBHS_EVENT_CALLBACK	Type of the Hi-Speed USB Driver event callback function.
	DRV_USBHS_HOST_PIPE_HANDLE	Defines the Hi-Speed USB Driver Host Pipe Handle type.
	DRV_USBHS_ROOT_HUB_PORT_INDICATION	USB Root hub Application Hooks (Port Indication).
	DRV_USBHS_ROOT_HUB_PORT_OVER_CURRENT_DETECT	USB Root hub Application Hooks (Port Overcurrent detection).
	DRV_USBHS_ROOT_HUB_PORT_POWER_ENABLE	USB Root hub Application Hooks (Port Power Enable/ Disable).

Description

PIC32MZ USB Module Driver Interface Header File

The PIC32MZ Hi-Speed USB Module driver provides a simple interface to manage the "USB" peripheral on the PIC32MZ microcontroller. This file defines the interface definitions and prototypes for the Hi-Speed USB Driver. The driver interface meets the requirements of the MPLAB Harmony USB Host and Device Layer.

File Name

drv_usbhs.h

Company

Microchip Technology Inc.

drv_usbhs_config_template.h

Hi-Speed USB (USBHS) Driver Configuration Template.

Macros

	Name	Description
	DRV_USBHS_DEVICE_SUPPORT	Determines if the USB Device Functionality should be enabled.
	DRV_USBHS_ENDPOINTS_NUMBER	Configures the number of endpoints to be provisioned in the driver.
	DRV_USBHS_HOST_ATTACH_DEBOUNCE_DURATION	Configures the time duration (in milliseconds) that the driver will wait to reconfirm a device attach.
	DRV_USBHS_HOST_NAK_LIMIT	Configures the NAK Limit for Host Mode Control Transfers.
	DRV_USBHS_HOST_PIPES_NUMBER	Configures the maximum number of pipes that are can be opened when the driver is operating in Host mode.
	DRV_USBHS_HOST_RESET_DURATION	Configures the time duration (in milliseconds) of the Reset Signal.
	DRV_USBHS_HOST_SUPPORT	Determines if the USB Host Functionality should be enabled.
	DRV_USBHS_INSTANCES_NUMBER	Specifies the number of driver instances to be enabled in the application.
	DRV_USBHS_INTERRUPT_MODE	Configures the driver for interrupt or polling mode operation.

Description

Hi-Speed USB Driver Configuration Template

This file lists all the configurations constants that affect the operation of the USBHS Driver.

File Name

drv_usbhs_config_template.h

Company

Microchip Technology Inc.

USART Driver Library

This section describes the USART Driver Library.

Introduction

This section introduces the MPLAB Harmony USART Driver.

Description

The MPLAB Harmony USART Driver (also referred to as the USART Driver) provides a high-level interface to the USART and UART peripherals on Microchip's PIC32 microcontrollers. This driver provides application ready routines to read and write data to the UART using common data transfer models, which eliminates the need for the application to implement this code. The USART driver features the following:

- Provides byte transfer, read/write, and buffer queue data transfer models
- Supports Interrupt and Polled modes of operation
- Supports point-to-point data communication
- Supports multi-client and multi-instance operation
- Provides data transfer events
- Supports blocking and non-blocking operation with the read/write data transfer model
- Features thread-safe functions for use in RTOS applications
- Supports DMA transfers
- Supports high baud rate setting
- Major features are implemented in separate source code files and can be included only if needed. This helps optimize overall application code size.

Using the Library

This topic describes the basic architecture of the USART Driver Library and provides information and examples on its use.

Description

Interface Header File: `drv_usart.h`

The interface to the USART library is defined in the `drv_usart.h` header file.

Please refer to the What is MPLAB Harmony? section for how the driver interacts with the framework.

Abstraction Model

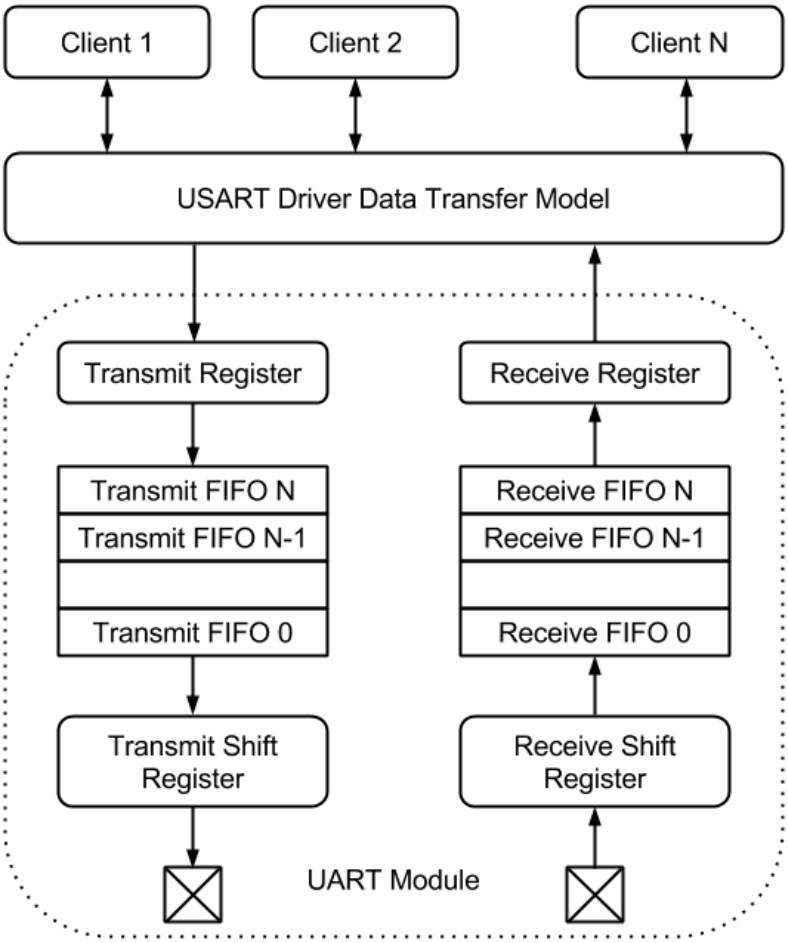
This section describes how the USART Driver abstracts the USART peripheral features.

Description

The USART driver features routines to perform the following functions:

- Driver initialization
- Transfer data
- Manage communication properties of the module

The Driver initialization routines allow the system to initialize the driver. The driver must be initialized before it can be opened by a client. The data transfer routines allow the application to receive and transmit data through the USART. The driver also provides routines to change the communication properties such as USART baud or line control settings.



As seen in the previous figure, the USART driver clients transfer data through the USART Driver Data Transfer model. The driver abstracts out the hardware details of the UART module FIFO mechanism and shift registers, and provides a low overhead data transfer mechanism to the application. The USART driver provides three different data transfer models for transferring data.

- The Byte Transfer Model
- The File I/O Type Read/Write Transfer Model
- Buffer Queue Transfer Model

Byte Transfer Model:

The byte transfer model allows the application to transfer data through USART driver one byte at a time. With this model, the driver reads one byte from the receive FIFO or writes one byte to the transmit FIFO. The application must check if data has been received before reading the data. Similarly, it must check if the transmit FIFO is not full before writing to the FIFO. The byte transfer model places the responsibility of maintaining the USART peripheral on the Application. The driver cannot support other data transfer models if support for this data transfer model is enabled. The byte transfer model is only recommended for simple data transfer applications.

To use the byte transfer model, the `drv_usart_byte_model.c` file must be included in the project and the DRV_USART_BYTE_MODEL_SUPPORT configuration macro should be set to true.

File I/O Type Read/Write Transfer Model:

This data transfer model is similar to file read and write API model in a UNIX operating system application. The application calls the USART driver read and write routines to transfer data through the USART. Unlike the byte transfer model, the read/write data model can process a block of data. Depending on the mode (blocking or non-blocking) in which the client opened the driver, the driver will either block until all of the data is transferred or will immediately return with the number of bytes transferred. The application does not have to check the FIFO status while using this mode. The application can instead use the return status (number of bytes transferred) to maintain its logic and complete the data transfer. The read/write model can be used with the non-DMA buffer queue model. It cannot be used with the byte transfer model and the DMA-enabled buffer queue model in the same application.

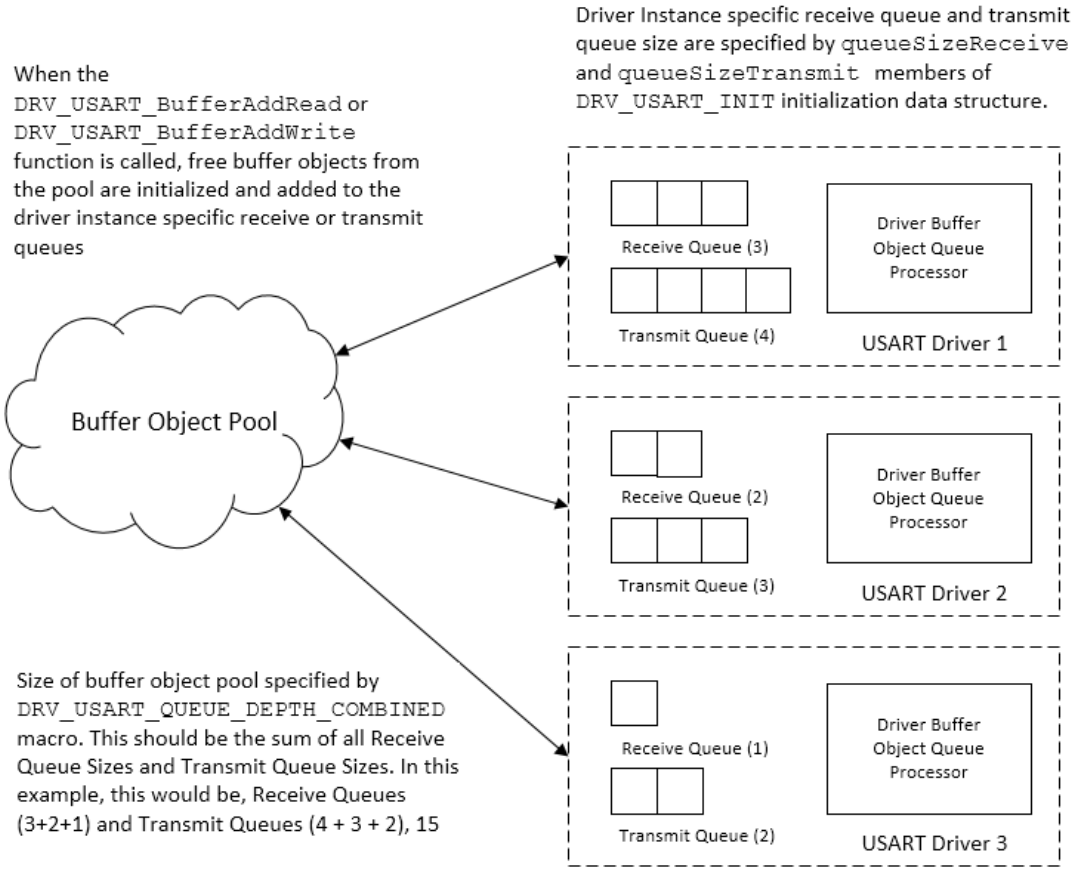
To use the file I/O type read/write data transfer model, the `drv_usart_read_write.c` file must be included in the project and the DRV_USART_READ_WRITE_MODEL_SUPPORT configuration macro should be set to true.

See File I/O Type Read/Write Data Transfer Model for additional information.

Buffer Queue Transfer Model:

The buffer queue data transfer model allows clients to queue data transfers for processing. This data transfer model is always non-blocking. The USART driver returns a buffer handle for a queued request. The clients can track the completion of a buffer through events and API. If the USART driver is busy processing a data transfer, other data transfer requests are queued. This allows the clients to optimize their application logic and increase throughput. To optimize memory usage, the USART driver implements a shared buffer object pool concept to add a data transfer request to the queue. The following figure shows a conceptual representation of the buffer queue model.

Buffer Queue Transfer Model



As shown in the previous figure, each USART driver hardware instance has a read and write queue. The system designer must configure the sizes of these read and write queues. The USART driver additionally employs a global pool of buffer queue objects. This pool is common to all USART Driver hardware instances and its size is defined by the DRV_USART_QUEUE_DEPTH_COMBINED configuration macro. When a client places a request to add a data transfer, the driver performs the following actions:

- It checks if a buffer object is free in the global pool. If not, the driver rejects the request.
- It then checks if the hardware instance specific queue is full. If not, the buffer object from the global pool is added to the hardware instance specific queue. If the queue is full, the driver rejects the request.

The buffer queue model can be used along with the file I/O type read/write data transfer model.

To use the Buffer Queue Data Transfer model, the `drv_usart_buffer_queue.c` file must be included in the project and DRV_USART_BUFFER_QUEUE_SUPPORT configuration macro should be set to true.

The USART Driver DMA feature is only available while using the Buffer Queue Model. If enabled, the USART Driver uses the DMA module channels to transfer data directly from application memory to USART transmit or receive registers. This reduces CPU resource consumption and improves system performance. To use the buffer queue model with DMA, the `drv_usart_buffer_queue_dma.c` file should be included in the project instead of `drv_usart_buffer_queue.c`.

See Buffer Queue Transfer Model for additional information.

Communication Management

The USART Driver API contains functions to control the USART Driver communication properties. These functions allow the client to change the parity, stop bits, number of data bits and the communication baud rate. A change in the communication setting affects all ongoing communication and all driver clients.

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the USART Driver Library.

Library Interface Section	Description
System Routines	These routines are accessed by the MPLAB Harmony system module. They allow the driver to be initialized, deinitialized and maintained.
Core Client Routines	These routines allow the application client to open and close the driver.
Communication Management Client Routines	These routines allow the client to change the properties of the communication channel (such as baud, parity, etc.).
Buffer Queue Read/Write Client Routines	These routines allow the client to use the buffer queue data transfer model.
File I/O Type Read/Write Routines	These routines allow the client to use the file I/O type read/write routines.
Byte Transfer Routines	These routines allow the client to use the byte data transfer model.

The USART driver must be first initialized. One or more application clients can then open the USART Driver in Blocking or non-Blocking mode. The Open function returns a handle which allows the client to access the driver client functionality. The Driver tasks routines should be invoked regularly from the SYS_Tasks routine in case of Polled mode operation or from USART Driver Interrupt Service Routine (ISR), in case of Interrupt mode.

The driver implementation is split across multiple files to optimize the application project code size. The application project must include the `drv_usart.c` file if the USART driver is needed in the application. If DMA-enabled data transfers are required, the `drv_usart_dma.c` file should be included into the project instead of the `drv_usart.c` file. These files implement the system and core Client routines. Other driver files can be included based on the required driver features.

The USART Driver API, unless otherwise specified, should not be called from an interrupt context. That is, they should not be called from an ISR or from event handlers that are executing within an ISR context.

How the Library Works

This section describes how to use the USART Driver.

Description

Prior to using the USART Driver, the application must decide on which USART data transfer models are required. The application project should then include the USART Driver files, required to support the data transfer model into the application project. Additionally, the application design must consider the need for USART Driver to be opened in blocking or non blocking modes. This will also affect the application flow.

Initializing the USART Driver

Describes how to initialize the USART Driver.

Description

The USART Driver must be configured and initialized for clients to be able to open the driver. The driver build time configuration is defined by the configuration macros. Refer to the Building the Library section for the location of and more information on the various configuration macros and how these macros should be designed. The driver initialization is configured through the DRV_USART_INIT data structure that is passed to the DRV_USART_Initialize function. The initialization parameters include the USART baud, the USART Peripheral, USART interrupts and read queue and write queue sizes (which are applicable only when buffer queue data transfer is used). The following code shows an example of initializing the USART driver for 300 bps and uses USART2. If the driver is configured for Interrupt mode of operation, the priority of the USART interrupts needs to be specified.

```
/* The following code shows an example of designing the
 * DRV_USART_INIT data structure. It also shows how an example
 * usage of the DRV_USART_Initialize() function and how Interrupt
 * System Service routines are used to set USART Interrupt
 * priority. */

DRV_USART_INIT usartInit;
SYS_MODULE_OBJ usartModule1;

/* Set the baud to 300 */
usartInit.baud = 300;

/* Auto Baud detection or Stop Idle is not needed */
usartInit.flags = DRV_USART_INIT_FLAG_NONE;

/* Handshaking is not needed */
usartInit.handshake = DRV_USART_HANDSHAKE_NONE;
```

```
/* USART Error Interrupt source for this USART
 * driver instance. Note that INT_SOURCE_USART_2_ERROR
 * value is defined by the Interrupt System Service and
 * is the error interrupt for USART 2*/
usartInit.interruptError = INT_SOURCE_USART_2_ERROR;

/* USART Receive Interrupt source for this USART
 * driver instance. Note that INT_SOURCE_USART_2_RECEIVE
 * value is defined by the Interrupt System Service and
 * is the error interrupt for USART 2 */
usartInit.interruptReceive = INT_SOURCE_USART_2_RECEIVE;

/* USART Transmit Interrupt source for this USART
 * driver instance. Note that INT_SOURCE_USART_2_TRANSMIT
 * value is defined by the Interrupt System Service and
 * is the error interrupt for USART 2 */
usartInit.interruptTransmit = INT_SOURCE_USART_2_TRANSMIT;

/* Line control mode */
usartInit.lineControl = DRV_USART_LINE_CONTROL_8NONE1;

/* Operation mode is normal. Loopback or addressed is not
 * needed */
usartInit.mode = DRV_USART_OPERATION_MODE_NORMAL;

/* Peripheral Bus clock frequency at which the USART is
 * operating */
usartInit.brgClock = 80000000;

/* System module power setting. Typically set to
 * SYS_MODULE_POWER_RUN_FULL */
usartInit.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;

/* Receive buffer queue size. In this case a maximum of 2
 * receive buffers can be queued. Only applicable if the
 * Buffer Queue Data Transfer Model is included in the
 * application. */
usartInit.queueSizeReceive = 2;

/* Transmit buffer queue size. In this case a maximum of 3
 * transmit buffers can be queued. Only applicable if the
 * Buffer Queue Data Transfer Model is included in the
 * application. */
usartInit.queueSizeTransmit = 3;

/* The USART peripheral instance index associated with this
 * driver instance. Note that this value is defined by the
 * USART Peripheral Library */
usartInit.usartID = USART_ID_2;

/* Initialize USART Driver Instance 0 */
usartModule1 = DRV_USART_Initialize(DRV_USART_0, (SYS_MODULE_INIT*)&usartInit);

/* The result of the driver initialization can be checked */
if(SYS_MODULE_OBJ_INVALID == usartModule1)
{
    /* There was an error in initialization. */
}

/* If the USART driver is configured for interrupt mode of
 * operation, the interrupt priorities should be configured.
 * Here the Interrupt System Service is used to set the
 * priority to level 4 */

/* Initialize the interrupt system service */
SYS_INT_Initialize();

/* Set the USART 2 module interrupt priority to 4*/
SYS_INT_VectorPrioritySet(INT_VECTOR_UART2, INT_PRIORITY_LEVEL4);
```

```
    /* Set the USART 2 module interrupt sub-priority to 0*/
    SYS_INT_VectorSubprioritySet(INT_VECTOR_UART2, INT_SUBPRIORITY_LEVEL0);

    /* Enable global interrupt */
    SYS_INT_Enable();
```

The USART Driver can be configured to transfer data through the DMA. In such a case, the DMA channels to be used for USART transmit and receive need to be specified. The USART Driver depends on the DMA System Service to access the DMA module. The DMA channels to be used for transmit and receive transfers should be specified in the DRV_USART_INIT data structure.

The following code shows an example of using the USART Driver initialization to use DMA for transferring data. The code also shows example initialization of the DMA System Service.

```
  /* The following code shows an example of designing the
   * DRV_USART_INIT data structure. It also shows how an example
   * usage of the DRV_USART_Initialize() function and how Interrupt
   * System Service routines are used to set USART Interrupt
   * priority. */

    DRV_USART_INIT usartInit;
    SYS_DMA_INIT dmaInit;
    SYS_MODULE_OBJ usartModule1;
    SYS_MODULE_OBJ dmaModule;

    /* Set the baud to 300 */
    usartInit.baud = 300;

    /* Auto Baud detection or Stop Idle is not needed */
    usartInit.flags = DRV_USART_INIT_FLAG_NONE;

    /* Handshaking is not needed */
    usartInit.handshake = DRV_USART_HANDSHAKE_NONE;

    /* USART Error Interrupt source for this USART
     * driver instance. Note that INT_SOURCE_USART_2_ERROR
     * value is defined by the Interrupt System Service and
     * is the error interrupt for USART2*/
    usartInit.interruptError = INT_SOURCE_USART_2_ERROR;

    /* USART Receive Interrupt source for this USART
     * driver instance. Note that INT_SOURCE_USART_2_RECEIVE
     * value is defined by the Interrupt System Service and
     * is the receive interrupt for USART2 */
    usartInit.interruptReceive = INT_SOURCE_USART_2_RECEIVE;

    /* USART Transmit Interrupt source for this USART
     * driver instance. Note that INT_SOURCE_USART_2_TRANSMIT
     * value is defined by the Interrupt System Service and
     * is the transmit interrupt for USART2 */
    usartInit.interruptTransmit = INT_SOURCE_USART_2_TRANSMIT;

    /* Line control mode */
    usartInit.lineControl = DRV_USART_LINE_CONTROL_8NONE1;

    /* Operation mode is normal. Loopback or addressed is not
     * needed */
    usartInit.mode = DRV_USART_OPERATION_MODE_NORMAL;

    /* Peripheral Bus clock frequency at which the USART is
     * operating */
    usartInit.brgClock = 80000000;

    /* System module power setting. Typically set to
     * SYS_MODULE_POWER_RUN_FULL */
    usartInit.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;

    /* Receive buffer queue size. In this case a maximum of 2
     * receive buffers can be queued. Only applicable if the
     * Buffer Queue Data Transfer Model is included in the
     * application. */
```

```
usartInit.queueSizeReceive = 2;

/* Transmit buffer queue size. In this case a maximum of 3
 * transmit buffers can be queued. Only applicable if the
 * Buffer Queue Data Transfer Model is included in the
 * application. */
usartInit.queueSizeTransmit = 3;

/* The USART peripheral instance index associated with this
 * driver instance. Note that this value is defined by the
 * USART Peripheral Library */
usartInit.usartID = USART_ID_2;

/* Use DMA channel 1 for transmit. If transmit via DMA is
 * not required, set this to DMA_CHANNEL_NONE. These values
 * are defined by the DMA System Service. */
usartInit.dmaChannelTransmit = DMA_CHANNEL_1;

/* Use DMA channel 2 for receive. If receive via DMA is
 * not required, set this to DMA_CHANNEL_NONE. These values
 * are defined by the DMA System Service. */
usartInit.dmaChannelReceive = DMA_CHANNEL_2;

/* Set the interrupt source for the Transmit DMA channel.
 * This parameter is ignored if the dmaChannelTransmit
 * parameter is set to DMA_CHANNEL_NONE. */
usartInit.dmaInterruptTransmit = INT_SOURCE_DMA_1;

/* Set the interrupt source for the Receive DMA channel.
 * This parameter is ignored if the dmaChannelReceive
 * parameter is set to DMA_CHANNEL_NONE. */
usartInit.dmaInterruptReceive = INT_SOURCE_DMA_2;

/********* End of DRV_USART_INIT Initialization *************/


/* If the USART driver is configured for interrupt mode of
 * operation, the interrupt priorities should be configured.
 * Here the Interrupt System Service is used to set the
 * priority to level 4 */

/* Initialize the interrupt system service */
SYS_INT_Initialize();

/* Set the USART 2 module interrupt priority to 4*/
SYS_INT_VectorPrioritySet(INT_VECTOR_UART2, INT_PRIORITY_LEVEL4);

/* Set the USART 2 module interrupt sub-priority to 0*/
SYS_INT_VectorSubprioritySet(INT_VECTOR_UART2, INT_SUBPRIORITY_LEVEL0);

/* Set the DMA 1 channel interrupt priority to 4*/
SYS_INT_VectorPrioritySet(INT_VECTOR_DMA1, INT_PRIORITY_LEVEL4);

/* Set the DMA 1 channel interrupt sub-priority to 0*/
SYS_INT_VectorSubprioritySet(INT_VECTOR_DMA1, INT_SUBPRIORITY_LEVEL0);

/* Set the DMA 2 channel interrupt priority to 4*/
SYS_INT_VectorPrioritySet(INT_VECTOR_DMA2, INT_PRIORITY_LEVEL4);

/* Set the DMA 2 channel interrupt sub-priority to 0*/
SYS_INT_VectorSubprioritySet(INT_VECTOR_DMA2, INT_SUBPRIORITY_LEVEL0);

/* Enable global interrupt */
SYS_INT_Enable();

/* This is the DMA System Service Initialization */
dmaInit.sidl = SYS_DMA_SIDL_DISABLE;
dmaModule = SYS_DMA_Initialize((SYS_MODULE_INIT*)&dmaInit);
```

```
    /* The result of the DMA System Service initialization can be checked */
    if(SYS_MODULE_OBJ_INVALID == dmaModule)
    {
        /* DMA System Service initialization was not successful */
    }

    /* Initialize USART Driver Instance 0 */
    usartModule1 = DRV_USART_Initialize(DRV_USART_0, (SYS_MODULE_INIT*)&usartInit);

    /* The result of the driver initialization can be checked */
    if(SYS_MODULE_OBJ_INVALID == usartModule1)
    {
        /* There was an error in initialization. */
    }
```

Opening the USART Driver

Describes how to open the USART Driver.

Description

To use the USART driver, the application must open the driver. This is done by calling the DRV_USART_Open function. Calling this function with DRV_IO_INTENT_NONBLOCKING will cause the driver to be opened in non blocking mode. The DRV_USART_Read and DRV_USART_Write functions when called by this client will be non blocking. . Calling this function with DRV_IO_INTENT_BLOCKING will cause the driver to be opened in blocking mode. The DRV_USART_Read and DRV_USART_Write functions when called by this client will be blocking.

If successful, the DRV_USART_Open function will return a valid handle to the driver. This handle records the association between the client and the driver instance that was opened. The DRV_USART_Open function may return DRV_HANDLE_INVALID in the situation where the driver is not ready to be opened. When this occurs, the application can try opening the driver again. Note that the open function may return an invalid handle in other (error) cases as well.

The following code shows an example of the driver being opened in different modes.

```
DRV_HANDLE usartHandle1, usartHandle2;

/* Client 1 opens the USART driver in non blocking mode */
usartHandle1 = DRV_USART_Open(DRV_USART_0, DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_NONBLOCKING);

/* Check if the handle is valid */
if(DRV_HANDLE_INVALID == usartHandle1)
{
    /* The driver was not opened successfully. The client
     * can try opening it again */
}

/* Client 2 opens the USART driver in blocking mode */
usartHandle2 = DRV_USART_Open(DRV_USART_0, DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_BLOCKING);

/* Check if the handle is valid */
if(DRV_HANDLE_INVALID == usartHandle2)
{
    /* The driver was not opened successfully. The client
     * can try opening it again */
}

/* The client can also open the USART driver in read only mode
 * (DRV_IO_INTENT_READ), write only mode (DRV_IO_INTENT_WRITE)
 * and exclusive mode (DRV_IO_INTENT_EXCLUSIVE). If the driver
 * has been opened exclusively by a client, it cannot be opened
 * again by another client */
```

Byte Transfer Model

Describes the USART Driver byte transfer model.

Description

To use the byte transfer model, the DRV_USART_BYTE_MODEL_SUPPORT configuration macro should be true. The `drv_usart_byte_model.c` function should be included in the application project. The application cannot support the read/write and buffer queue data transfer model when the byte model is enabled.

The following code shows an example of how the DRV_USART_WriteByte function and the DRV_USART_ReadByte function are used.

```
 /* Client uses the a byte model API to write a byte*/
if(!DRV_USART_TransmitBufferIsFull(usartHandle1))
{
    byte = '1';
    DRV_USART_WriteByte(usartHandle1,byte);
}

/* Client waits until data is available and then reads
 * byte */
 while(DRV_USART_ReceiverBufferIsEmpty(usartHandle1));
        byte = DRV_USART_ReadByte(usartHandle1);
```

File I/O Type Read/Write Data Transfer Model

This topic describes the file I/O type read/write data transfer model.

Description

To use the file I/O type read/write data transfer model, the DRV_USART_READ_WRITE_MODEL_SUPPORT configuration macro should be 'true'. The file `drv_usart_read_write.c` file should be included in the application project. The driver can support the non-DMA buffer queue data transfer model along with the file I/O type read/write data transfer model. The byte transfer model and DMA buffer queue model cannot be enabled if the file I/O type read/write data transfer model is enabled.

The DRV_USART_Read and DRV_USART_Write function represent the file I/O type read/write data transfer model. The functional behavior of these API is affected by the mode in which the client opened the driver. If the client opened the driver in blocking mode these API will block. In blocking mode, the DRV_USART_Read and DRV_USART_Write functions will not return until the requested number of bytes have been read or written. When operating in a RTOS application, the application thread that has opened driver in blocking mode, will enter a blocked state when it calls DRV_USART_Write or DRV_USART_Read function. This will allow the RTOS scheduler to schedule other threads which are ready to run. If the client opened the driver in non-blocking mode these API will not block. In non-blocking mode, the DRV_USART_Read and DRV_USART_Write functions will return immediately with the amount of data that could be read or written.

Note: Do not open the driver in Blocking mode when the driver is configured for polling operation (DRV_USART_INTERRUPT_MODE is false) in a bare-metal (non-RTOS) application. This will cause the system to enter an infinite loop condition when the DRV_USART_Read or DRV_USART_Write function is called.

The following code shows an example of file I/O type read/write data transfer model usage when the driver is opened in Blocking mode.

```
/* This code shows the functionality of the DRV_USART_Write and
 * DRV_USART_Read function when the driver is opened in blocking mode */

DRV_HANDLE usartHandle1;
uint8_t myData[10];
size_t bytesProcessed;

/* The driver is opened in blocking mode */
usartHandle1 = DRV_USART_Open(DRV_USART_0, DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_BLOCKING);

/* Check if the driver was opened successfully */
if(DRV_HANDLE_INVALID == usartHandle1)
{
    /* The driver could not be opened successfully */
}

/* Transmit 10 bytes from the myData array. Function will not return until 10 bytes
 * have been accepted by the driver. This is because the client opened the driver
 * in blocking mode. */

bytesProcessed = DRV_USART_Write(usartHandle1, myData, 10);

/* Read 10 bytes from the myData array. Function will not return until all 10 bytes
 * have been received by the driver. This is because the client opened the driver
 * in blocking mode. */

bytesProcessed = DRV_USART_Read(usartHandle1, myData, 10);
```

In non-Blocking mode, the driver uses the internal USART hardware FIFO as storage. The DRV_USART_Read function checks if bytes are available in USART receive hardware FIFO. If bytes are available, these are read and the number of bytes read is returned. The DRV_USART_Write function checks if USART transmit hardware FIFO has empty location. If locations are empty, the bytes to be transmitted are queued up in the FIFO and the number of queued bytes is returned. In either case, the number of bytes read or written may be less than the number requested by the client. The client can, in such a case, call the DRV_USART_Read and/or the DRV_USART_Write functions again to process the pending bytes. The following code shows how this can be done.

```
/* This code shows the functionality of the DRV_USART_Write and
 * DRV_USART_Read functions when the driver is opened in non-blocking mode */

DRV_HANDLE usartHandle1;
uint8_t myData[10];
size_t bytesProcessed;

/* The driver is opened in non-blocking mode */
usartHandle1 = DRV_USART_Open(DRV_USART_0,
        DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_NONBLOCKING);

/* Check if the driver was opened successfully */
if(DRV_HANDLE_INVALID == usartHandle1)
{
    /* The driver could not be opened successfully */
}

/* The following code call the DRV_USART_Write function
 * multiple times to write 10 bytes completely. Note how the
 * function return value is used to update the location of
 * user source data. */

bytesProcessed = 0;
do
{
    /* Write data to the USART and use the return value to
     * update the source data pointer and pending bytes number. */
    bytesProcessed += DRV_USART_Write(usartHandle1,
            myData + bytesProcessed, (10 - bytesProcessed));

} while(bytesProcessed < 10);

/* The following code calls the DRV_USART_Read function multiple times to read
 * 10 bytes completely. Note how the function return value is used to update the
 * location of user destination array. */

bytesProcessed = 0;
do
{
    /* Read data from the USART and use the return value to update the
     * destination pointer and pending bytes number. */
    bytesProcessed += DRV_USART_Read(usartHandle1,
            myData + bytesProcessed, (10 - bytesProcessed));

}while (bytesProcessed < 10);
```

Buffer Queue Transfer Model

This topic describes the buffer queue data transfer model.

Description

To use the buffer queue data transfer model, the DRV_USART_BUFFER_QUEUE_SUPPORT configuration macro should be true. The file, `drv_usart_buffer_queue.c`, should be included in the application project. If the DMA-enabled buffer queue model is required, the `drv_usart_buffer_queue_dma.c` file (*and not* the `drv_usart_buffer_queue.c`) should be included in the application project. The DMA and non-DMA buffer queue model API is the same. The driver can support the non-DMA buffer queue data transfer model along with the file I/O type read/write data transfer model. The byte transfer model cannot be enabled if the buffer queue data transfer model is enabled.

The DRV_USART_BufferAddRead and DRV_USART_BufferAddWrite functions represent the buffer queue data transfer model. These functions are always non-blocking. The Buffer Queue Data Transfer Model employs queuing of read and write request. Each driver instance contains a read and write queue. The size of the read queue is determined by the queueSizeRead member of the DRV_USART_INIT data structure. The size of the write queue is determined by the queueSizeWrite member of the DRV_USART_INIT data structure. The driver provides driver events (DRV_USART_BUFFER_EVENT) that indicates termination of the buffer requests.

When the driver is configured for Interrupt mode operation (that is defined and registered by the driver client), the buffer event handler executes in an interrupt context. Calling computationally intensive or hardware polling routines within the event handlers is not recommended. Calling interrupt unsafe functions in the event handler when the driver is configured for Interrupt mode could result in unpredictable system behavior.

When the driver adds request to the queue, it returns a buffer handle. This unique handle allows the client to track the request as it progresses through the queue. The buffer handle is returned with the buffer event and expires when the event associated with the buffer has been generated and the event handler returns. The following code shows an example of using the buffer queue data transfer model.

```
/* This code shows an example of using the
 * Buffer Queue Data Transfer Model. */
DRV_HANDLE usartHandle1;
uint8_t myData1[10], myData2[10];
uint8_t myData3[10], myData4[10];
size_t bytesProcessed;
DRV_USART_BUFFER_HANDLE bufferHandle1, bufferHandle2;
DRV_USART_BUFFER_HANDLE bufferHandle3, bufferHandle4;


/* The driver is opened in non blocking mode */
usartHandle1 = DRV_USART_Open(DRV_USART_0,
        DRV_IO_INTENT_READWRITE|DRV_IO_INTENT_NONBLOCKING);

/* Check if the driver was opened successfully */
if(DRV_HANDLE_INVALID == usartHandle1)
{
    /* The driver could not be opened successfully */
}

/* Register a Buffer Event Handler with USART driver.
 * This event handler function will be called whenever
 * there is a buffer event.  An application defined
 * context can also be specified. This is returned when
 * the event handler is called.
 * */
DRV_USART_BufferEventHandlerSet(usartHandle1,
        APP_USARTBufferEventHandler, NULL);


/* Queue up two buffers for transmit */
DRV_USART_BufferAddWrite(usartHandle1, &bufferHandle1, myData1, 10);
DRV_USART_BufferAddWrite(usartHandle1, &bufferHandle2, myData2, 10);

/* Queue up two buffers for receive */
DRV_USART_BufferAddRead(usartHandle1, &bufferHandle3, myData3, 10);
DRV_USART_BufferAddRead(usartHandle1, &bufferHandle4, myData4, 10);

/* This is application USART Driver Buffer Event Handler */

void APP_USARTBufferEventHandler(DRV_USART_BUFFER_EVENT event,
        DRV_USART_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    switch(event)
    {
        case DRV_USART_BUFFER_EVENT_COMPLETE:
            /* This means the data was transferred */
            break;
        case DRV_USART_BUFFER_EVENT_ERROR:
            /* Error handling here. */
            break;
        default:
            break;
    }
}
```

Driver Tasks Routine

This topic describes the Driver "Task" routines.

Description

The USART driver contains three task routines, DRV_USART_TasksTransmit, DRV_USART_TasksReceive and DRV_USART_TasksError. These task routines implement the USART Driver state machines for transmit, receive and error related operations. If the driver is configured for polling operation, the required task routine should be called in SYS_Tasks routine of the system. If the driver is configured for interrupt mode of operation, the task routine should be called from the ISR. The following code shows an example of both.

```
/* The following code shows an example of
 * USART2 Interrupt Service Routine. This function
 * will be called when a USART2 interrupt occurs
```

```
 * and the driver is configured for interrupt mode
 * operation */

void __ISR ( _UART_2_VECTOR,ipl4 ) _InterruptHandler_USART ( void )
{
    /* usartModule1 is the System Module Object
     * that was returned by the DRV_USART_Initialize
     * function. */

    DRV_USART_TasksTransmit(usartModule1);
    DRV_USART_TasksReceive(usartModule1);
    DRV_USART_TasksError(usartModule1);

}

/* In case of Polled mode, the tasks routines are
 * invoked from the SYS_Tasks() routine. */

void SYS_Tasks(void)
{
    DRV_USART_TasksTransmit(usartModule1);
    DRV_USART_TasksReceive(usartModule1);
    DRV_USART_TasksError(usartModule1);
}

/* The SYS_Tasks routine is invoked from the main
 * application while(1) loop. */

while(1)
{
    SYS_Tasks();
}
```

Using the USART Driver with DMA

This topic provides information on using the USART Driver with DMA.

Description

To use the USART Driver with DMA, the following should be noted:

- Include `drv_usart_dma.c` in the project. Do not include `drv_usart.c`.
- Include `drv_usart_buffer_queue_dma.c` in the project. Do not include `drv_usart_buffer_queue.c`.
- Initialize the driver to use DMA. Refer to Initializing the USART Driver for details.
- Refer to the DMA System Service section for details on initializing and using the DMA system service in Polling or Interrupt mode
- The DRV_USART_INTERRUPT_MODE configuration macro should be set to 'true'
- Do not directly invoke the DRV_USART_TasksTransmit and DRV_USART_TasksReceive functions

Configuring the Library

Macros

	Name	Description
	DRV_USART_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_USART_INDEX	USART Static Index selection.
	DRV_USART_INTERRUPT_MODE	Macro controls interrupt based operation of the driver.
	DRV_USART_INTERRUPT_SOURCE_ERROR	Defines the error interrupt source for the static driver.
	DRV_USART_PERIPHERAL_ID	Configures the USART PLIB Module ID.
	DRV_USART_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_USART_BUFFER_QUEUE_SUPPORT	Specifies if the Buffer Queue support should be enabled.
	DRV_USART_BYTE_MODEL_SUPPORT	Specifies if the Byte Model support should be enabled.
	DRV_USART_INTERRUPT_SOURCE_RECEIVE	Defines the Receive interrupt source for the static driver.
	DRV_USART_INTERRUPT_SOURCE_RECEIVE_DMA	Defines the Receive DMA Channel interrupt source for the static driver.

	DRV_USART_INTERRUPT_SOURCE_TRANSMIT	Defines the Transmit interrupt source for the static driver.
	DRV_USART_INTERRUPT_SOURCE_TRANSMIT_DMA	Defines the Transmit DMA Channel interrupt source for the static driver.
	DRV_USART_QUEUE_DEPTH_COMBINED	Defines the number of entries of all queues in all instances of the driver.
	DRV_USART_READ_WRITE_MODEL_SUPPORT	Specifies if Read/Write Model support should be enabled.
	DRV_USART_RECEIVE_DMA	Defines the USART Driver Receive DMA Channel for the static driver.
	DRV_USART_TRANSMIT_DMA	Defines the USART Driver Transmit DMA Channel in case of static driver.
	DRV_USART_BAUD_RATE_IDXn	Specifies the USART Baud at which the USART driver is initialized.
	DRV_USART_BYTE_MODEL_BLOCKING	Enables or Disables DRV_USART_ByteWrite function blocking behavior.
	DRV_USART_BYTE_MODEL_CALLBACK	Enables or Disables Callback Feature of the Byte Transfer Model.
	DRV_USART_RCV_QUEUE_SIZE_IDXn	Sets the USART Driver Receive Queue Size while using the Buffer Queue Data Transfer Model.
	DRV_USART_XMIT_QUEUE_SIZE_IDXn	Sets the USART Driver Transmit Queue Size while using the Buffer Queue Data Transfer Model.

Description

The USART Driver requires the specification of compile-time configuration macros. These macros define resource usage, feature availability, and dynamic behavior of the driver. These configuration macros should be defined in the `system_config.h` file.

This header can be placed anywhere in the application specific folders and the path of this header needs to be presented to the include search for a successful build. Refer to the Applications Help section for more details.

Note: Initialization overrides are not supported in this version.

```
/* In this configuration example, the USART driver
 * must manage only on USART peripheral instance.
 * This macro can be greater than one if more
 * USART peripherals are needed. Not defining this
 * macro will cause the driver to be built in
 * static mode */
#define DRV_USART_INSTANCES_NUMBER 1

/* There will be 3 different client that use the
 * one instance of the USART peripheral. Note that
 * this macro configures the total (combined) number of clients
 * across all instance of the USART driver. Not defining
 * this macro will cause the driver to be configured
 * for single client operation */
#define DRV_USART_CLIENTS_NUMBER 3

/* USART Driver should be built for interrupt mode.
 * Set this to false for Polled mode operation */
#define DRV_USART_INTERRUPT_MODE true

/* Combined buffer queue depth is 5. Refer to the
 * description of the Buffer Queue data transfer model
 * and the DRV_USART_QUEUE_DEPTH_COMBINED macro
 * for more details on how this is configured. */
#define DRV_USART_QUEUE_DEPTH_COMBINED 5

/* Set this macro to true is Buffer Queue data
 * transfer model is to be enabled. */
#define DRV_USART_BUFFER_QUEUE_SUPPORT true

/* Set this macro to true if Byte by Byte data
 * transfer model is to be enabled. */
#define DRV_USART_BYTE_MODEL_SUPPORT false

/* Set this macro to true File IO type Read Write
 * data transfer model is to be enabled */
#define DRV_USART_READ_WRITE_MODEL_SUPPORT false
```

DRV_USART_CLIENTS_NUMBER Macro

Sets up the maximum number of clients that can be connected to any hardware instance.

File

drv_usart_config_template.h

C

```
#define DRV_USART_CLIENTS_NUMBER 4
```

Description

USART Client Count Configuration

This macro sets up the maximum number of clients that can be connected to any hardware instance. This value represents the total number of clients to be supported across all hardware instances. Therefore, if USART1 will be accessed by two clients and USART2 will accessed by three clients, this number should be 5. It is recommended that this value be set exactly equal to the number of expected clients, as client support consumes RAM memory space. If this macro is not defined and the DRV_USART_INSTANCES_NUMBER macro is not defined, the driver will be built for static - single client operation. If this macro is defined and the DRV_USART_INSTANCES_NUMBER macro is not defined, the driver will be built for static - multi client operation.

Remarks

None.

DRV_USART_INDEX Macro

USART Static Index selection.

File

drv_usart_config_template.h

C

```
#define DRV_USART_INDEX DRV_USART_INDEX_2
```

Description

Index - Used for static drivers

USART Static Index selection for the driver object reference. This macro defines the driver index for static and static multi-client builds. For example, if this macro is set to DRV_USART_INDEX_2, the static driver APIs would be DRV_USART2_Initialize, DRV_USART2_Open, etc. When building static drivers, this macro should be different for each static build of the USART driver that needs to be included in the project.

Remarks

This index is required to make a reference to the driver object

DRV_USART_INTERRUPT_MODE Macro

Macro controls interrupt based operation of the driver.

File

drv_usart_config_template.h

C

```
#define DRV_USART_INTERRUPT_MODE true
```

Description

USART Interrupt Mode Operation Control

This macro controls the interrupt based operation of the driver. The possible values are:

- true - Enables the interrupt mode
- false - Enables the polling mode

If the macro value is true, the Interrupt Service Routine (ISR) for the interrupt should be defined in the system. The DRV_USART_Tasks routine should be called in the ISR. While using the USART driver with DMA, this flag should always be true.

Remarks

None.

DRV_USART_INTERRUPT_SOURCE_ERROR Macro

Defines the error interrupt source for the static driver.

File

drv_usart_config_template.h

C

```
#define DRV_USART_INTERRUPT_SOURCE_ERROR INT_SOURCE_USART_2_ERROR
```

Description

Error Interrupt Source

This macro defines the Error interrupt source for the static driver. The interrupt source defined by this macro will override the errorInterruptSource member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the USART module error interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None.

DRV_USART_PERIPHERAL_ID Macro

Configures the USART PLIB Module ID.

File

drv_usart_config_template.h

C

```
#define DRV_USART_PERIPHERAL_ID USART_ID_2
```

Description

USART Peripheral Library Module ID

This macro configures the PLIB ID if the driver is built statically. This value will override the usartID member of the DRV_USART_INIT initialization data structure. In that when the driver is built statically, the usartID member of the DRV_USART_INIT data structure will be ignored by the driver initialization routine and this macro will be considered. This should be set to the PLIB ID of USART module (USART_ID_1, USART_ID_2, and so on).

Remarks

None.

DRV_USART_INSTANCES_NUMBER Macro

Sets up the maximum number of hardware instances that can be supported.

File

drv_usart_config_template.h

C

```
#define DRV_USART_INSTANCES_NUMBER 2
```

Description

USART driver objects configuration

This macro sets up the maximum number of hardware instances that can be supported. It is recommended that this number be set exactly equal to the number of USART modules that are needed by the application, as hardware Instance support consumes RAM memory space. If this macro is not defined, the driver will be built statically.

Remarks

None

DRV_USART_BUFFER_QUEUE_SUPPORT Macro

Specifies if the Buffer Queue support should be enabled.

File

drv_usart_config_template.h

C

```
#define DRV_USART_BUFFER_QUEUE_SUPPORT true
```

Description

USART Driver Buffer Queue Support

This macro defines whether or not Buffer Queue support should be enabled. Setting this macro to true will enable buffer queue support and all buffer related driver function. The driver should be built along with the drv_usart_buffer_queue.c file, which contains the functional implementation for buffer queues. If buffer queue operation is enabled, the DRV_USART_BYTE_MODEL_SUPPORT function should not be true. If this macro is set to false, the behavior of the USART Driver Buffer Queue API is not defined. While using the USART driver with DMA, the driver supports Buffer Queue Data transfer model regardless of the value of this configuration macro.

Remarks

None.

DRV_USART_BYTE_MODEL_SUPPORT Macro

Specifies if the Byte Model support should be enabled.

File

drv_usart_config_template.h

C

```
#define DRV_USART_BYTE_MODEL_SUPPORT false
```

Description

USART Driver Byte Model Support

This macro defines whether or Byte Model support should be enabled. Setting this macro to true will enable byte model support and all byte operation related driver functions. The driver should be built along with the drv_usart_byte_model.c file, which contains the functional implementation for byte model operation. If byte model operation is enabled, the driver will not support buffer queue and read write models. The behavior of the byte mode API when this macro is set to false is not defined.

Remarks

None.

DRV_USART_INTERRUPT_SOURCE_RECEIVE Macro

Defines the Receive interrupt source for the static driver.

File

drv_usart_config_template.h

C

```
#define DRV_USART_INTERRUPT_SOURCE_RECEIVE INT_SOURCE_USART_2_RECEIVE
```

Description

Receive Interrupt Source

This macro defines the Receive interrupt source for the static driver. The interrupt source defined by this macro will override the rxInterruptSource member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the USART module receive interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None.

DRV_USART_INTERRUPT_SOURCE_RECEIVE_DMA Macro

Defines the Receive DMA Channel interrupt source for the static driver.

File

drv_usart_config_template.h

C

```
#define DRV_USART_INTERRUPT_SOURCE_RECEIVE_DMA
```

Description

Receive DMA Channel Interrupt Source

This macro defines the Receive DMA Channel interrupt source for the static driver. The interrupt source defined by this macro will override the dmaInterruptReceive member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None.

DRV_USART_INTERRUPT_SOURCE_TRANSMIT Macro

Defines the Transmit interrupt source for the static driver.

File

drv_usart_config_template.h

C

```
#define DRV_USART_INTERRUPT_SOURCE_TRANSMIT INT_SOURCE_USART_2_TRANSMIT
```

Description

Transmit Interrupt Source

This macro defines the TX interrupt source for the static driver. The interrupt source defined by this macro will override the txInterruptSource member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the USART module transmit interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None.

DRV_USART_INTERRUPT_SOURCE_TRANSMIT_DMA Macro

Defines the Transmit DMA Channel interrupt source for the static driver.

File

drv_usart_config_template.h

C

```
#define DRV_USART_INTERRUPT_SOURCE_TRANSMIT_DMA
```

Description

Transmit DMA Channel Interrupt Source

This macro defines the TX DMA Channel interrupt source for the static driver. The interrupt source defined by this macro will override the dmaInterruptTransmit member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel interrupt enumeration in the Interrupt PLIB for the microcontroller.

Remarks

None.

DRV_USART_QUEUE_DEPTH_COMBINED Macro

Defines the number of entries of all queues in all instances of the driver.

File

drv_usart_config_template.h

C

```
#define DRV_USART_QUEUE_DEPTH_COMBINED 16
```

Description

USART Driver Instance combined queue depth.

This macro defines the number of entries of all queues in all instances of the driver.

Each hardware instance supports a buffer queue for transmit and receive operations. The size of queue is specified either in driver initialization (for dynamic build) or by macros (for static build). The hardware instance transmit buffer queue will queue transmit buffers submitted by the DRV_USART_BufferAddWrite function. The hardware instance receive buffer queue will queue receive buffers submitted by the DRV_USART_BufferAddRead function.

A buffer queue will contain buffer queue entries, with each related to a BufferAdd request. This configuration macro defines the total number of buffer entries that will be available for use between all USART driver hardware instances. The buffer queue entries are allocated to individual hardware instances as requested by hardware instances. Once the request is processed, the buffer queue entry is free for use by other hardware instances.

The total number of buffer entries in the system determines the ability of the driver to service non blocking read and write requests. If a free buffer entry is not available, the driver will not add the request and will return an invalid buffer handle. The greater the number of buffer entries, the greater the ability of the driver to service and add requests to its queue. A hardware instance additionally can queue up as many buffer entries as specified by its transmit and receive buffer queue size.

For example, consider the case of static single client driver application where full duplex non blocking operation is desired without queuing, the minimum transmit queue depth and minimum receive queue depth should be 1. Therefore, the total number of buffer entries should be 2.

As another example, consider the case of a dynamic driver (i.e., two instances) where instance 1 will queue up to three write requests and up to two read requests, and instance 2 will queue up to two write requests and up to six read requests, the value of this macro should be: 13 (2 + 3 + 2 + 6).

Remarks

None.

DRV_USART_READ_WRITE_MODEL_SUPPORT Macro

Specifies if Read/Write Model support should be enabled.

File

drv_usart_config_template.h

C

```
#define DRV_USART_READ_WRITE_MODEL_SUPPORT true
```

Description

USART Driver Read Write Model Support

This macro defines whether or not Read Write Model support should be enabled. Setting this macro to true will enable read write model support and all read/write related driver functions. The driver should be built along with the drv_usart_read_write.c file, which contains the functional implementation for byte model operation. If read/write model operation is enabled, the DRV_USART_BYTE_MODEL_SUPPORT macro should not be true. The behavior of the Read Write Model API when this macro is set to false is not defined.

Remarks

None.

DRV_USART_RECEIVE_DMA Macro

Defines the USART Driver Receive DMA Channel for the static driver.

File

drv_usart_config_template.h

C

```
#define DRV_USART_RECEIVE_DMA
```

Description

USART Driver Receive DMA Channel

This macro defines the USART Receive DMA Channel for the static driver. The DMA channel defined by this macro will override the dmaReceive member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel in the

DMA PLIB for the microcontroller.

Remarks

None.

DRV_USART_TRANSMIT_DMA Macro

Defines the USART Driver Transmit DMA Channel in case of static driver.

File

drv_usart_config_template.h

C

```
#define DRV_USART_TRANSMIT_DMA
```

Description

USART Driver Transmit DMA Channel

This macro defines the USART Transmit DMA Channel for the static driver. The DMA channel defined by this macro will override the dmaTransmit member of the DRV_USB_INIT initialization data structure in the driver initialization routine. This value should be set to the DMA channel in the DMA PLIB for the microcontroller.

Remarks

None.

DRV_USART_BAUD_RATE_IDXn Macro

Specifies the USART Baud at which the USART driver is initialized.

File

drv_usart_config_template.h

C

```
#define DRV_USART_BAUD_RATE_IDXn
```

Description

USART Driver Baud Selection.

This configuration constant specifies the baud rate at which the USART Driver is initialized. This is the baud rate at which the USART module will operate when the driver initialization has completed. The driver client can call the DRV_USART_BaudSet function after opening the driver to change the USART baud rate after initialization has completed.

Remarks

This constant is automatically generated by MHC and its value is set to the value specified in USART Driver Baud Selection field.

DRV_USART_BYTE_MODEL_BLOCKING Macro

Enables or Disables DRV_USART_ByteWrite function blocking behavior.

File

drv_usart_config_template.h

C

```
#define DRV_USART_BYTE_MODEL_BLOCKING
```

Description

USART Driver Byte Write Blocking Behavior

This USART Driver MHC option controls the blocking behavior of the DRV_USART_ByteWrite function and is only applicable when the USART Driver Byte Transfer model is selected. Selecting this option will cause the DRV_USART_ByteWrite function to block until the byte has been written to the USART Transmit FIFO. Blocking behavior is enabled by default (to enable backward compatibility with previous versions of the driver). This option can be used for simple applications where interoperability with other MPLAB Harmony modules is not a design concern.

If the application uses several other MPLAB Harmony modules (Middleware, File System, etc.), it is recommended to disable this option and use the non-blocking DRV_USART_ByteWrite function. This requires the application to call the DRV_USART_TransmitBufferIsFull function to check if the byte can be written to the USART, as shown in the following code example.

```
if(!DRV_USART_TransmitBufferIsFull(usartHandle1))
{
    byte = '1';
    DRV_USART_WriteByte(usartHandle1,byte);
}
```

Using the non-blocking implementation results in improved application interoperability with other MPLAB Harmony modules.

Remarks

The DRV_USART_BYTE_MODEL_BLOCKING constant is specified for documentation purposes only. It does not affect the configuration of the driver.

DRV_USART_BYTE_MODEL_CALLBACK Macro

Enables or Disables Callback Feature of the Byte Transfer Model.

File

drv_usart_config_template.h

C

```
#define DRV_USART_BYTE_MODEL_CALLBACK
```

Description

USART Driver Byte Model Callback Feature.

This USART Driver MHC option controls the Callback feature of the Byte Transfer model. Selecting this option allows an application to register Byte Transfer Event Callback functions with the driver. These callback functions are invoked on the occurrence of Byte Transfer events. Callback functions can be registered to Byte Transmit, Byte Receive, and USART Error events, as shown in the following code example.

```
// This code shows how a callback function is
// registered for the Byte Receive event.
DRV_USART_ByteReceiveCallbackSet(DRV_USART_INDEX_0, APP_USARTReceiveEventHandler);

// Event Processing Technique. Event is received when
// a byte is received.

void APP_USARTReceiveEventHandler(const SYS_MODULE_INDEX index)
{
    // Byte has been Received. Handle the event.
    // Read byte using DRV_USART_ReadByte.
}
```

When operating in Interrupt mode, the callback functions are invoked in an interrupt context. If this option is not selected, the application must use the DRV_USART_TransmitBufferIsFull, DRV_USART_ReceiverBufferIsEmpty, and DRV_USART_ErrorGet functions to check the status of Byte transmit or receive.

Remarks

The DRV_USART_BYTE_MODEL_CALLBACK constant is specified for documentation purposes only. It does not affect the configuration of the driver.

DRV_USART_RCV_QUEUE_SIZE_IDXn Macro

Sets the USART Driver Receive Queue Size while using the Buffer Queue Data Transfer Model.

File

drv_usart_config_template.h

C

```
#define DRV_USART_RCV_QUEUE_SIZE_IDXn
```

Description

USART Driver Receive Queue Size Selection.

This constant sets the USART Driver Receive queue size when using the Buffer Queue Data Transfer Model. It affects the queuing capacity of the DRV_USART_BufferAddRead function for the selected driver instance. For example, if this option is set to 5 for USART Driver 0, USART Driver 0 can then queue up to a maximum of five driver client receive buffer requests from any driver clients.

Therefore, if USART Driver 0 has two clients and if client 1 has queued up three buffers for receive, client 2 can only queue up to two buffers. If the client attempts to queue up more buffers, DRV_USART_BufferAddRead will not accept the request and will generate an invalid buffer handle (DRV_USART_BUFFER_HANDLE_INVALID).

Remarks

This constant is automatically generated by MHC and its value is set to the value specified in USART Driver Receive Queue Size field.

DRV_USART_XMIT_QUEUE_SIZE_IDXn Macro

Sets the USART Driver Transmit Queue Size while using the Buffer Queue Data Transfer Model.

File

drv_usart_config_template.h

C

```
#define DRV_USART_XMIT_QUEUE_SIZE_IDXn
```

Description

USART Driver Transmit Queue Size Selection.

This constant sets the USART Driver Transmit queue size when using the Buffer Queue Data Transfer Model. It affects the queuing capacity of the DRV_USART_BufferAddWrite function, for the selected driver instance. For example, if this option is set to 5 for USART Driver 0, USART Driver 0 can then queue up to a maximum of five driver client transmit buffer requests from any driver clients.

Therefore if USART Driver 0 has two clients and if client 1 has queued up three buffers for transmit, client 2 can only queue up to two buffers. If the client attempts to queue up more buffers, DRV_USART_BufferAddWrite will not accept the request and will generate an invalid buffer handle (DRV_USART_BUFFER_HANDLE_INVALID).

Remarks

This constant is automatically generated by MHC and its value is set to the value specified in USART Driver Transmit Queue Size field.

Building the Library

This section lists the files that are available in the USART Driver Library.

Description

This section list the files that are available in the \src folder of the USART Driver. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/usart`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`/drv_usart.h`	This file should be included by any .c file which accesses the USART Driver API. This one file contains the prototypes for all driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`/src/dynamic/drv_usart.c`	This file should always be included in the project when using the USART Driver.
`/src/dynamic/drv_usart_dma.c`	This file should always be included in the project when using the USART driver with DMA.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
`/src/dynamic/drv_usart_byte_model.c`	This file should be included in the project if the USART Driver Byte Model API is required.
`/src/dynamic/drv_usart_buffer_queue.c`	This file should be included in the project if the USART Driver Buffer Queue Model API (without DMA) is required.

/src/dynamic/drv_usart_read_write.c	This file should be included in the project if the USART Driver Read Write Model API is required.
/src/dynamic/drv_usart_buffer_queue_dma.c	This file should be included in the project if the USART Driver Buffer Queue Model API with DMA is required.

Module Dependencies

The USART Driver Library depends on the following modules:

- Interrupt System Service Library
- DMA System Service Library (if USART Driver is configured to use DMA)

Library Interface

a) System Functions

	Name	Description
	DRV_USART_Initialize	Initializes the USART instance for the specified driver index. **Implementation:** Static/Dynamic
	DRV_USART_Deinitialize	Deinitializes the specified instance of the USART driver module. **Implementation:** Static/Dynamic
	DRV_USART_Status	Gets the current status of the USART driver module. **Implementation:** Static/Dynamic
	DRV_USART_TasksReceive	Maintains the driver's receive state machine and implements its ISR. **Implementation:** Static/Dynamic
	DRV_USART_TasksTransmit	Maintains the driver's transmit state machine and implements its ISR. **Implementation:** Static/Dynamic
	DRV_USART_TasksError	Maintains the driver's error state machine and implements its ISR. **Implementation:** Static/Dynamic

b) Core Client Functions

	Name	Description
	DRV_USART_Open	Opens the specified USART driver instance and returns a handle to it. **Implementation:** Static/Dynamic
	DRV_USART_Close	Closes an opened-instance of the USART driver. **Implementation:** Static/Dynamic
	DRV_USART_ClientStatus	Gets the current client-specific status the USART driver. **Implementation:** Static/Dynamic
	DRV_USART_ErrorGet	This function returns the error(if any) associated with the last client request. **Implementation:** Static/Dynamic

c) Communication Management Client Functions

	Name	Description
	DRV_USART_BaudSet	This function changes the USART module baud to the specified value. **Implementation:** Static/Dynamic
	DRV_USART_LineControlSet	This function changes the USART module line control to the specified value. **Implementation:** Static/Dynamic

d) Buffer Queue Read/Write Client Functions

	Name	Description
	DRV_USART_BufferAddRead	Schedule a non-blocking driver read operation. **Implementation:** Static/Dynamic
	DRV_USART_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Static/Dynamic
	DRV_USART_BufferEventHandlerSet	Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Static/Dynamic
	DRV_USART_BufferProcessedSizeGet	This API will be deprecated and not recommended to use. Use DRV_USART_BufferCompletedBytesGet to get the number of bytes processed for the specified buffer.

	DRV_USART_AddressedBufferAddWrite	Schedule a non-blocking addressed driver write operation. **Implementation:** Dynamic
	DRV_USART_BufferCompletedBytesGet	Returns the number of bytes that have been processed for the specified buffer. **Implementation:** Static/Dynamic
	DRV_USART_BufferRemove	Removes a requested buffer from the queue. **Implementation:** Static/Dynamic

e) File I/O Type Read/Write Functions

	Name	Description
	DRV_USART_Read	Reads data from the USART. **Implementation:** Static/Dynamic
	DRV_USART_Write	Writes data to the USART. **Implementation:** Static/Dynamic

f) Byte Transfer Functions

	Name	Description
	DRV_USART_ReadByte	Reads a byte of data from the USART. **Implementation:** Static/Dynamic
	DRV_USART_WriteByte	Writes a byte of data to the USART. **Implementation:** Static/Dynamic
	DRV_USART_TransmitBufferSizeGet	Returns the size of the transmit buffer. **Implementation:** Static/Dynamic
	DRV_USART_ReceiverBufferSizeGet	Returns the size of the receive buffer. **Implementation:** Static/Dynamic
	DRV_USART_TransferStatus	Returns the transmitter and receiver transfer status. **Implementation:** Static/Dynamic
	DRV_USART_TransmitBufferIsFull	Provides the status of the driver's transmit buffer. **Implementation:** Static/Dynamic
	DRV_USART_ReceiverBufferIsEmpty	Provides the status of the driver's receive buffer. **Implementation:** Static/Dynamic
	DRV_USART_ByteErrorCallbackSet	Registers callback to handle for byte error events.
	DRV_USART_ByteReceiveCallbackSet	Registers receive callback function for byte receive event.
	DRV_USART_ByteTransmitCallbackSet	Registers a callback function for byte transmit event.

Description

This section describes the functions of the USART Driver Library.

Refer to each section for a detailed description.

a) System Functions

DRV_USART_Initialize Function

Initializes the USART instance for the specified driver index.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
SYS_MODULE_OBJ DRV_USART_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, returns SYS_MODULE_OBJ_INVALID.

Description

This routine initializes the USART driver instance for the specified driver index, making it ready for clients to open and use it. The initialization data is specified by the init parameter. The initialization may fail if the number of driver objects allocated are insufficient or if the specified driver instance is already initialized. The driver instance index is independent of the USART module ID. For example, driver instance 0 can be assigned to

USART2. If the driver is built statically, then some of the initialization parameters are overridden by configuration macros. Refer to the description of the DRV_USART_INIT data structure for more details on which members on this data structure are overridden.

Remarks

This routine must be called before any other USART routine is called.

This routine should only be called once during system initialization unless DRV_USART_Deinitialize is called to deinitialize the driver instance. This routine will NEVER block for hardware access.

Preconditions

None.

Example

```
// The following code snippet shows an example USART driver initialization.
// The driver is initialized for normal mode and a baud of 300. The
// receive queue size is set to 2 and transmit queue size is set to 3.

DRV_USART_INIT              usartInit;
SYS_MODULE_OBJ              objectHandle;

usartInit.baud  = 300;
usartInit.mode  = DRV_USART_OPERATION_MODE_NORMAL;
usartInit.flags = DRV_USART_INIT_FLAG_NONE;
usartInit.usartID   = USART_ID_2;
usartInit.brgClock  = 80000000;
usartInit.handshake = DRV_USART_HANDSHAKE_NONE;
usartInit.lineControl       = DRV_USART_LINE_CONTROL_8NONE1;
usartInit.interruptError    = INT_SOURCE_USART_2_ERROR;
usartInit.interruptReceive  = INT_SOURCE_USART_2_RECEIVE;
usartInit.queueSizeReceive  = 2;
usartInit.queueSizeTransmit = 3;
usartInit.interruptTransmit = INT_SOURCE_USART_2_TRANSMIT;
usartInit.moduleInit.value  = SYS_MODULE_POWER_RUN_FULL;

objectHandle = DRV_USART_Initialize(DRV_USART_INDEX_1, (SYS_MODULE_INIT*)&usartInitData);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Identifier for the instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the driver.

Function

SYS_MODULE_OBJ DRV_USART_Initialize

(

const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init

)

DRV_USART_Deinitialize Function

Deinitializes the specified instance of the USART driver module.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
void DRV_USART_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

Deinitializes the specified instance of the USART driver module, disabling its operation (and any hardware). Invalidates all the internal data.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function DRV_USART_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     //  Returned from DRV_USART_Initialize
SYS_STATUS          status;

DRV_USART_Deinitialize(object);

status = DRV_USART_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_USART_Initialize routine

Function

void DRV_USART_Deinitialize(SYS_MODULE_OBJ object)

DRV_USART_Status Function

Gets the current status of the USART driver module.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
SYS_STATUS DRV_USART_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the driver is busy with a previous system level operation and cannot start another

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

Description

This routine provides the current status of the USART driver module.

Remarks

A driver can opened only when its status is SYS_STATUS_READY.

Preconditions

Function DRV_USART_Initialize should have been called before calling this function.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_USART_Initialize
SYS_STATUS          usartStatus;

usartStatus = DRV_USART _Status(object);
```

```
if (SYS_STATUS_READY == usartStatus)
{
    // This means the driver can be opened using the
    // DRV_USART_Open() function.
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from the DRV_USART_Initialize routine

Function

SYS_STATUS DRV_USART_Status(SYS_MODULE_OBJ object)

DRV_USART_TasksReceive Function

Maintains the driver's receive state machine and implements its ISR.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
void DRV_USART_TasksReceive(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal receive state machine and implement its receive ISR for interrupt-driven implementations. In polling mode, this function should be called from the SYS_Tasks function. In interrupt mode, this function should be called in the receive interrupt service routine of the USART that is associated with this USART driver hardware instance.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_USART_Initialize

while (true)
{
    DRV_USART_TasksReceive (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_USART_Initialize)

Function

void DRV_USART_TasksReceive (SYS_MODULE_OBJ object);

DRV_USART_TasksTransmit Function

Maintains the driver's transmit state machine and implements its ISR.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
void DRV_USART_TasksTransmit(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal transmit state machine and implement its transmit ISR for interrupt-driven implementations. In polling mode, this function should be called from the SYS_Tasks function. In interrupt mode, this function should be called in the transmit interrupt service routine of the USART that is associated with this USART driver hardware instance.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_USART_Initialize

while (true)
{
    DRV_USART_TasksTransmit (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_USART_Initialize)

Function

void DRV_USART_TasksTransmit (SYS_MODULE_OBJ object);

DRV_USART_TasksError Function

Maintains the driver's error state machine and implements its ISR.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
void DRV_USART_TasksError(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This routine is used to maintain the driver's internal error state machine and implement its error ISR for interrupt-driven implementations. In polling mode, this function should be called from the SYS_Tasks function. In interrupt mode, this function should be called in the error interrupt service routine of the USART that is associated with this USART driver hardware instance.

Remarks

This routine is normally not called directly by an application. It is called by the system's Tasks routine (SYS_Tasks) or by the appropriate raw ISR.

This routine may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

Example

```
SYS_MODULE_OBJ      object;     // Returned from DRV_USART_Initialize

while (true)
{
    DRV_USART_TasksError (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from DRV_USART_Initialize)

Function

void DRV_USART_TasksError (SYS_MODULE_OBJ object);

b) Core Client Functions

DRV_USART_Open Function

Opens the specified USART driver instance and returns a handle to it.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
DRV_HANDLE DRV_USART_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT ioIntent);
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is DRV_HANDLE_INVALID. Error can occur

- if the number of client objects allocated via DRV_USART_CLIENTS_NUMBER is insufficient.
- if the client is trying to open the driver but driver has been opened exclusively by another client.
- if the driver hardware instance being opened is not initialized or is invalid.
- if the client is trying to open the driver exclusively, but has already been opened in a non exclusive mode by another client.
- if the driver is not ready to be opened, typically when the initialize routine has not completed execution.

Description

This routine opens the specified USART driver instance and provides a handle that must be provided to all other client-level operations to identify the caller and the instance of the driver. The ioIntent parameter defines how the client interacts with this driver instance.

The DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING ioIntent options additionally affect the behavior of the DRV_USART_Read and DRV_USART_Write functions. If the ioIntent is DRV_IO_INTENT_NONBLOCKING, then these function will not block even if the required amount of data could not be processed. If the ioIntent is DRV_IO_INTENT_BLOCKING, these functions will block until the required amount of data is processed. If the driver is configured for polling and bare-metal operation, it will not support DRV_IO_INTENT_BLOCKING. The driver will operation will always be non-blocking.

If ioIntent is DRV_IO_INTENT_READ, the client will only be able to read from the driver. If ioIntent is DRV_IO_INTENT_WRITE, the client will only be able to write to the driver. If the ioIntent is DRV_IO_INTENT_READWRITE, the client will be able to do both, read and write.

Specifying a DRV_IO_INTENT_EXCLUSIVE will cause the driver to provide exclusive access to this client. The driver cannot be opened by any other client.

Remarks

The handle returned is valid until the DRV_USART_Close routine is called. This routine will NEVER block waiting for hardware.If the requested intent flags are not supported, the routine will return DRV_HANDLE_INVALID. This function is thread safe in a RTOS application.

Preconditions

Function DRV_USART_Initialize must have been called before calling this function.

Example

```
DRV_HANDLE handle;

handle = DRV_USART_Open(DRV_USART_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
if (DRV_HANDLE_INVALID == handle)
{
    // Unable to open the driver
    // May be the driver is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
intent	Zero or more of the values from the enumeration DRV_IO_INTENT "ORed" together to indicate the intended use of the driver. See function description for details.

Function

DRV_HANDLE DRV_USART_Open
(
const SYS_MODULE_INDEX index,
const DRV_IO_INTENT ioIntent
)

DRV_USART_Close Function

Closes an opened-instance of the USART driver.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
void DRV_USART_Close(const DRV_HANDLE handle);
```

Returns

None.

Description

This routine closes an opened-instance of the USART driver, invalidating the handle. Any buffers in the driver queue that were submitted by this client will be removed. After calling this routine, the handle passed in "handle" must not be used with any of the remaining driver routines (with one possible exception described in the "Remarks" section). A new handle must be obtained by calling DRV_USART_Open before the caller may use the driver again

Remarks

Usually there is no need for the client to verify that the Close operation has completed. The driver will abort any ongoing operations when this routine is called. However, if it requires additional time to do so in a non-blocking environment, it will still return from the Close operation but the handle is now a zombie handle. The client can only call the DRV_USART_ClientStatus on a zombie handle to track the completion of the Close operation. The DRV_USART_ClientStatus routine will return DRV_CLIENT_STATUS_CLOSED when the close operation has completed.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE handle;  // Returned from DRV_USART_Open

DRV_USART_Close(handle);
```

```
// After this point, the handle cannot be used with any other function
// except the DRV_USART_ClientStatus function, which can be used to query
// the success status of the DRV_USART_Close function.

while(DRV_USART_CLIENT_STATUS_CLOSED != DRV_USART_ClientStatus(handle));
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

void DRV_USART_Close(DRV_Handle handle)

DRV_USART_ClientStatus Function

Gets the current client-specific status the USART driver.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
DRV_USART_CLIENT_STATUS DRV_USART_ClientStatus(DRV_HANDLE handle);
```

Returns

A DRV_USART_CLIENT_STATUS value describing the current status of the driver.

Description

This function gets the client-specific status of the USART driver associated with the given handle. This function can be used to check the status of client after the DRV_USART_Close() function has been called.

Remarks

This function will not block for hardware access and will immediately return the current status. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_USART_Initialize function must have been called.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE              handle;  // Returned from DRV_USART_Open
DRV_USART_CLIENT_STATUS   status;

status = DRV_USART_ClientStatus(handle);
if( DRV_USART_CLIENT_STATUS_CLOSED != status )
{
    // The client had not closed.
}
```

Parameters

Parameters	Description
handle	Handle returned from the driver's open function.

Function

DRV_USART_CLIENT_STATUS DRV_USART_ClientStatus(DRV_HANDLE handle)

DRV_USART_ErrorGet Function

This function returns the error(if any) associated with the last client request.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
DRV_USART_ERROR DRV_USART_ErrorGet(const DRV_HANDLE client);
```

Returns

A DRV_USART_ERROR type indicating last known error status.

Description

This function returns the error(if any) associated with the last client request. DRV_USART_Read and DRV_USART_Write will update the client error status when these functions return DRV_USART_TRANSFER_ERROR. If the driver send a DRV_USART_BUFFER_EVENT_ERROR to the client, the client can call this function to know the error cause. The error status will be updated on every operation and should be read frequently (ideally immediately after the driver operation has completed) to know the relevant error status.

Remarks

It is the client's responsibility to make sure that the error status is obtained frequently. The driver will update the client error status regardless of whether this has been examined by the client. This function is thread safe when used in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_USART_BUFFER_HANDLE bufferHandle;

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

// Client registers an event handler with driver. This is done once.

DRV_USART_BufferEventHandlerSet( myUSARTHandle, APP_USARTBufferEventHandle,
                                 (uintptr_t)&myAppObj );

bufferHandle = DRV_USART_BufferAddRead( myUSARThandle,
                                        myBuffer, MY_BUFFER_SIZE );

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_USARTBufferEventHandler( DRV_USART_BUFFER_EVENT event,
        DRV_USART_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle )
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;

    switch(event)
    {
        case DRV_USART_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_USART_BUFFER_EVENT_ERROR:
```

```
            // Error handling here.
            // We can find out how many bytes were processed in this
            // buffer before the error occurred. We can also find
            // the error cause.

            processedBytes = DRV_USART_BufferCompletedBytesGet(bufferHandle);
            if(DRV_USART_ERROR_RECEIVE_OVERRUN == DRV_USART_ErrorGet(myUSARTHandle))
            {
                // There was an receive over flow error.
                // Do error handling here.
            }

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
bufferhandle	Handle of the buffer of which the processed number of bytes to be obtained.

Function

DRV_USART_ERROR DRV_USART_ErrorGet(DRV_HANDLE client);

c) Communication Management Client Functions

DRV_USART_BaudSet Function

This function changes the USART module baud to the specified value.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
DRV_USART_BAUD_SET_RESULT DRV_USART_BaudSet(const DRV_HANDLE client, uint32_t baud);
```

Returns

None.

Description

This function changes the USART module baud to the specified value. Any queued buffer requests will be processed at the updated baud. The USART driver operates at the baud specified in DRV_USART_Initialize function unless the DRV_USART_BaudSet function is called to change the baud.

Remarks

The implementation of this function, in this release of the driver, changes the baud immediately. This may interrupt on-going data transfer. It is recommended that the driver be opened exclusively if this function is to be called. This function is thread safe when used in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

DRV_USART_BaudSet(myUSARTHandle, 9600);
```

Parameters

Parameters	Description
handle	client handle returned by DRV_USART_Open function.
baud	desired baud.

Function

void DRV_USART_BaudSet(DRV_HANDLE client, uint32_t baud);

DRV_USART_LineControlSet Function

This function changes the USART module line control to the specified value.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
DRV_USART_LINE_CONTROL_SET_RESULT DRV_USART_LineControlSet(const DRV_HANDLE client, const
DRV_USART_LINE_CONTROL lineControl);
```

Returns

DRV_USART_LINE_CONTROL_SET_SUCCESS if the function was successful. Returns DRV_HANDLE_INVALID if the client handle is not valid.

Description

This function changes the USART module line control parameters to the specified value. Any queued buffer requests will be processed at the updated line control parameters. The USART driver operates at the line control parameters specified in DRV_USART_Initialize function unless the DRV_USART_LineControlSet function is called to change the line control parameters.

Remarks

The implementation of this function, in this release of the driver, changes the line control immediately. This may interrupt on-going data transfer. It is recommended that the driver be opened exclusively if this function is to be called. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

DRV_USART_LineControlSet(myUSARTHandle, DRV_USART_LINE_CONTROL_8NONE1);
```

Parameters

Parameters	Description
handle	client handle returned by DRV_USART_Open function.
lineControl	line control parameters.

Function

void DRV_USART_LineControlSet

(

DRV_HANDLE client,

DRV_USART_LINE_CONTROL lineControl

);

d) Buffer Queue Read/Write Client Functions

DRV_USART_BufferAddRead Function

Schedule a non-blocking driver read operation.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
void DRV_USART_BufferAddRead(const DRV_HANDLE handle, DRV_USART_BUFFER_HANDLE * const bufferHandle, void *
buffer, const size_t size);
```

Returns

The buffer handle is returned in the bufferHandle argument. This is DRV_USART_BUFFER_HANDLE_INVALID if the request was not successful.

Description

This function schedules a non-blocking read operation. The function returns with a valid buffer handle in the bufferHandle argument if the read request was scheduled successfully. The function adds the request to the hardware instance receive queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. The function returns DRV_USART_BUFFER_HANDLE_INVALID in the bufferHandle argument:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the buffer size is 0
- if the read queue size is full or queue depth is insufficient.
- if the driver handle is invalid

If the requesting client registered an event callback with the driver, the driver will issue a DRV_USART_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully of DRV_USART_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the USART Driver Buffer Event Handler that is registered by the client. It should not be called in the event handler associated with another USART driver instance. It should not be called directly in an ISR.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART device instance and the DRV_USART_Status must have returned SYS_STATUS_READY.

DRV_USART_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_USART_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_USART_BUFFER_HANDLE bufferHandle;

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

// Client registers an event handler with driver

DRV_USART_BufferEventHandlerSet(myUSARTHandle,
                APP_USARTBufferEventHandler, (uintptr_t)&myAppObj);

DRV_USART_BufferAddRead(myUSARThandle, &bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_USARTBufferEventHandler(DRV_USART_BUFFER_EVENT event,
```

```
        DRV_USART_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_USART_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_USART_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the communication channel as returned by the DRV_USART_Open function.
buffer	Buffer where the received data will be stored.
size	Buffer size in bytes.

Function

void DRV_USART_BufferAddRead

(

const DRV_HANDLE handle,

DRV_USART_BUFFER_HANDLE * bufferHandle,

void * buffer,

const size_t size

)

DRV_USART_BufferAddWrite Function

Schedule a non-blocking driver write operation.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
void DRV_USART_BufferAddWrite(const DRV_HANDLE handle, DRV_USART_BUFFER_HANDLE * bufferHandle, void *
buffer, const size_t size);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_USART_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking write operation. The function returns with a valid buffer handle in the bufferHandle argument if the write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. On returning, the bufferHandle parameter may be DRV_USART_BUFFER_HANDLE_INVALID for the following reasons:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read-only
- if the buffer size is 0

- if the transmit queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_USART_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully or a DRV_USART_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the USART Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another USART driver instance. It should not otherwise be called directly in an ISR.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART device instance and the DRV_USART_Status must have returned SYS_STATUS_READY.

DRV_USART_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_USART_Open call.

Example

```
MY_APP_OBJ myAppObj;
uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_USART_BUFFER_HANDLE bufferHandle;

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

// Client registers an event handler with driver

DRV_USART_BufferEventHandlerSet(myUSARTHandle,
                APP_USARTBufferEventHandler, (uintptr_t)&myAppObj);

DRV_USART_BufferAddWrite(myUSARThandle, &bufferHandle,
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_USARTBufferEventHandler(DRV_USART_BUFFER_EVENT event,
        DRV_USART_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_USART_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_USART_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the communication channel as return by the DRV_USART_Open function.
bufferHandle	Pointer to an argument that will contain the return buffer handle.

buffer	Data to be transmitted.
size	Buffer size in bytes.

Function

void DRV_USART_BufferAddWrite
(
const DRV_HANDLE handle,
DRV_USART_BUFFER_HANDLE * bufferHandle,
void * buffer,
size_t size
);

DRV_USART_BufferEventHandlerSet Function

Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
void DRV_USART_BufferEventHandlerSet(const DRV_HANDLE handle, const DRV_USART_BUFFER_EVENT_HANDLER
eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. When a client calls either the DRV_USART_BufferAddRead or DRV_USART_BufferAddWrite function, it is provided with a handle identifying the buffer that was added to the driver's buffer queue. The driver will pass this handle back to the client by calling "eventHandler" function when the buffer transfer has completed.

The event handler should be set before the client performs any "buffer add" operations that could generate events. The event handler once set, persists until the client closes the driver or sets another event handler (which could be a "NULL" pointer to indicate no callback).

Remarks

If the client does not want to be notified when the queued buffer transfer has completed, it does not need to register a callback. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_USART_BUFFER_HANDLE bufferHandle;

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

// Client registers an event handler with driver. This is done once

DRV_USART_BufferEventHandlerSet( myUSARTHandle, APP_USARTBufferEventHandle,
                                 (uintptr_t)&myAppObj );

DRV_USART_BufferAddRead(myUSARThandle, &bufferHandle
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
```

```
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_USARTBufferEventHandler(DRV_USART_BUFFER_EVENT event,
        DRV_USART_BUFFER_HANDLE handle, uintptr_t context)
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) context;

    switch(event)
    {
        case DRV_USART_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_USART_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
eventHandler	Pointer to the event handler function.
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

void DRV_USART_BufferEventHandlerSet

(

const DRV_HANDLE handle,

const DRV_USART_BUFFER_EVENT_HANDLER eventHandler,

const uintptr_t context

)

DRV_USART_BufferProcessedSizeGet Function

This API will be deprecated and not recommended to use. Use DRV_USART_BufferCompletedBytesGet to get the number of bytes processed for the specified buffer.

File

drv_usart.h

C

```
size_t DRV_USART_BufferProcessedSizeGet(DRV_USART_BUFFER_HANDLE bufferHandle);
```

Returns

None.

Description

None.

Remarks

None.

Preconditions

None.

Example

None.

Function

size_t DRV_USART_BufferProcessedSizeGet

(

DRV_USART_BUFFER_HANDLE bufferHandle

);

DRV_USART_AddressedBufferAddWrite Function

Schedule a non-blocking addressed driver write operation.

Implementation: Dynamic

File

drv_usart.h

C

```
void DRV_USART_AddressedBufferAddWrite(const DRV_HANDLE hClient, DRV_USART_BUFFER_HANDLE * bufferHandle,
uint8_t address, void * source, size_t nWords);
```

Returns

The bufferHandle parameter will contain the return buffer handle. This will be DRV_USART_BUFFER_HANDLE_INVALID if the function was not successful.

Description

This function schedules a non-blocking addressed write operation. The function returns with a valid buffer handle in the bufferHandle argument if the addressed write request was scheduled successfully. The function adds the request to the hardware instance transmit queue and returns immediately. While the request is in the queue, the application buffer is owned by the driver and should not be modified. On returning, the bufferHandle parameter may be DRV_USART_BUFFER_HANDLE_INVALID for the following reasons:

- if a buffer could not be allocated to the request
- if the input buffer pointer is NULL
- if the client opened the driver for read-only
- if the buffer size is 0
- if the transmit queue is full or the queue depth is insufficient

If the requesting client registered an event callback with the driver, the driver will issue a DRV_USART_BUFFER_EVENT_COMPLETE event if the buffer was processed successfully or a DRV_USART_BUFFER_EVENT_ERROR event if the buffer was not processed successfully.

Remarks

This function is thread safe in a RTOS application. It can be called from within the USART Driver Buffer Event Handler that is registered by this client. It should not be called in the event handler associated with another USART driver instance. It should not otherwise be called directly in an ISR.

The source buffer should be a 16-bit word aligned buffer. The 9th bit of the higher byte 16-bit buffer is used to indicate data/address.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART device instance and the DRV_USART_Status must have returned SYS_STATUS_READY.

DRV_USART_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_USART_Open call.

The operation mode of the driver must be DRV_USART_OPERATION_MODE_ADDRESSED.

Example

```
MY_APP_OBJ myAppObj;
uint16_t mybuffer[MY_BUFFER_SIZE];
DRV_USART_BUFFER_HANDLE bufferHandle;
```

```
uint8_t clientAddress;

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

// Client registers an event handler with driver

clientAddress = 0x60;
DRV_USART_BufferEventHandlerSet(myUSARTHandle,
                APP_USARTBufferEventHandler, (uintptr_t)&myAppObj);

DRV_USART_AddressedBufferAddWrite(myUSARThandle, &bufferHandle, clientAddress
                                    myBuffer, MY_BUFFER_SIZE);

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event is received when
// the buffer is processed.

void APP_USARTBufferEventHandler(DRV_USART_BUFFER_EVENT event,
        DRV_USART_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle)
{
    // contextHandle points to myAppObj.

    switch(event)
    {
        case DRV_USART_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_USART_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
hClient	Handle of the communication channel as return by the DRV_USART_Open function.
bufferHandle	Pointer to an argument that will contain the return buffer handle.
address	Address of the receiver client
source	Data to be transmitted.
size	Buffer size in 16-bit words.

Function

void DRV_USART_AddressedBufferAddWrite
(
const DRV_HANDLE hClient,
DRV_USART_BUFFER_HANDLE * bufferHandle,
uint8_t address,
void * source,
size_t nWords
);

DRV_USART_BufferCompletedBytesGet Function

Returns the number of bytes that have been processed for the specified buffer.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
size_t DRV_USART_BufferCompletedBytesGet(DRV_USART_BUFFER_HANDLE bufferHandle);
```

Returns

Returns the number of bytes that have been processed for this buffer.

Returns DRV_USART_BUFFER_HANDLE_INVALID for an invalid or an expired buffer handle.

Description

This function returns number of bytes that have been processed for the specified buffer. The client can use this function, in a case where the buffer has terminated due to an error, to obtain the number of bytes that have been processed. Or in any other use case. This function can be used for non-DMA buffer transfers only. It cannot be used when the USART driver is configured to use DMA.

Remarks

This function is thread safe when used in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Either the DRV_USART_BufferAddRead or DRV_USART_BufferAddWrite function must have been called and a valid buffer handle returned.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_USART_BUFFER_HANDLE bufferHandle;

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

// Client registers an event handler with driver. This is done once

DRV_USART_BufferEventHandlerSet( myUSARTHandle, APP_USARTBufferEventHandle,
                                 (uintptr_t)&myAppObj );

bufferHandle = DRV_USART_BufferAddRead( myUSARThandle,
                                        myBuffer, MY_BUFFER_SIZE );

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_USARTBufferEventHandler( DRV_USART_BUFFER_EVENT event,
        DRV_USART_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle )
{
    // The context handle was set to an application specific
    // object. It is now retrievable easily in the event handler.
    MY_APP_OBJ myAppObj = (MY_APP_OBJ *) contextHandle;
    size_t processedBytes;

    switch(event)
    {
```

```
            case DRV_USART_BUFFER_EVENT_COMPLETE:

                // This means the data was transferred.
                break;

            case DRV_USART_BUFFER_EVENT_ERROR:

                // Error handling here.
                // We can find out how many bytes were processed in this
                // buffer before the error occurred.

                processedBytes = DRV_USART_BufferCompletedBytesGet(bufferHandle);

                break;

            default:
                break;
        }
    }
```

Parameters

Parameters	Description
bufferhandle	Handle for the buffer of which the processed number of bytes to be obtained.

Function

size_t DRV_USART_BufferCompletedBytesGet
(
DRV_USART_BUFFER_HANDLE bufferHandle
);

DRV_USART_BufferRemove Function

Removes a requested buffer from the queue.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
DRV_USART_BUFFER_RESULT DRV_USART_BufferRemove(DRV_USART_BUFFER_HANDLE bufferHandle);
```

Returns

DRV_USART_BUFFER_RESULT_HANDLE_INVALID - Buffer handle is invalid.

DRV_USART_BUFFER_RESULT_HANDLE_EXPIRED - Buffer handle is expired.

DRV_USART_BUFFER_RESULT_REMOVED_SUCCESFULLY - Buffer is removed from the queue successfully.

DRV_USART_BUFFER_RESULT_REMOVAL_FAILED - Failed to remove buffer from the queue because of mutex timeout in RTOS environment.

Description

This function removes a specified buffer from the queue. The client can use this function to delete

1. An unwated stalled buffer.
2. Queued buffers on timeout.

or in any other use case.

Remarks

This function is thread safe when used in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Either the DRV_USART_BufferAddRead or DRV_USART_BufferAddWrite function must have been called and a valid buffer handle returned.

Example

```
// myAppObj is an application specific object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];
DRV_USART_BUFFER_HANDLE bufferHandle;

// myUSARTHandle is the handle returned
// by the DRV_USART_Open function.

// Client registers an event handler with driver. This is done once

DRV_USART_BufferEventHandlerSet( myUSARTHandle, APP_USARTBufferEventHandle,
                                 (uintptr_t)&myAppObj );

bufferHandle = DRV_USART_BufferAddRead( myUSARThandle,
                                        myBuffer, MY_BUFFER_SIZE );

if(DRV_USART_BUFFER_HANDLE_INVALID == bufferHandle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the buffer is processed.

void APP_USARTBufferEventHandler( DRV_USART_BUFFER_EVENT event,
        DRV_USART_BUFFER_HANDLE bufferHandle, uintptr_t contextHandle )
{
    switch(event)
    {
        case DRV_USART_BUFFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case DRV_USART_BUFFER_EVENT_ERROR:

            // Error handling here.

            break;

        default:
            break;
    }
}

// Timeout function, where remove queued buffer if it still exists.
void APP_TimeOut(void)
{
    DRV_USART_BUFFER_RESULT bufferResult;
    bufferResult = DRV_USART_BufferRemove(bufferHandle);

    if(DRV_USART_BUFFER_RESULT_REMOVED_SUCCESFULLY == bufferResult)
    {
        //Buffer removed succesfully from the queue
    }
    else
    {
        //Either buffer is invalid or expired.
        //Or not able to acquire mutex in RTOS mode.
    }
}
```

Parameters

Parameters	Description
bufferhandle	Handle of the buffer to delete.

Function

DRV_USART_BUFFER_RESULT DRV_USART_BufferRemove(DRV_USART_BUFFER_HANDLE bufferHandle)

e) File I/O Type Read/Write Functions

DRV_USART_Read Function

Reads data from the USART.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
size_t DRV_USART_Read(const DRV_HANDLE handle, void * buffer, const size_t numbytes);
```

Returns

Number of bytes actually copied into the caller's buffer. Returns DRV_USART_READ_ERROR in case of an error.

Description

This routine reads data from the USART. This function is blocking if the driver was opened by the client for blocking operation. This function will not block if the driver was opened by the client for non blocking operation. If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_BLOCKING, this function will only return when (or will block until) numbytes of bytes have been received or if an error occurred. If there are buffers queued for receiving data, these buffers will be serviced first. The function will not return until the requested number of bytes have been read.

If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_NON_BLOCKING, this function will return with the number of bytes that were actually read. The function will not wait until numBytes of bytes have been read. If there are buffer queued for reading data, then the function will not block and will return immediately with 0 bytes read.

Remarks

This function is thread safe in a RTOS application. If the driver is configured for polled operation, this it will not support blocking operation in a bare metal (non-RTOS) application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_READ or DRV_IO_INTENT_READWRITE must have been specified in the DRV_USART_Open call.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
char            myBuffer[MY_BUFFER_SIZE];
unsigned int    count;
unsigned int    total;

total = 0;
do
{
    count  = DRV_USART_Read(myUSARTHandle, &myBuffer[total], MY_BUFFER_SIZE - total);
    if(count == DRV_USART_READ_ERROR)
    {
       // There was an error. The DRV_USART_ErrorGet() function
       // can be called to find the exact error.
    }
    total += count;

    // Do something else...

} while( total < MY_BUFFER_SIZE );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
buffer	Buffer into which the data read from the USART instance will be placed.
numbytes	Total number of bytes that need to be read from the module instance (must be equal to or less than the size of the buffer)

Function

size_t DRV_USART_Read
(
const DRV_HANDLE handle,
void * buffer,
const size_t numbytes
)

DRV_USART_Write Function

Writes data to the USART.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
size_t DRV_USART_Write(const DRV_HANDLE handle, void * buffer, const size_t numbytes);
```

Returns

Number of bytes actually written to the driver. Return DRV_USART_WRITE_ERROR in case of an error.

Description

This routine writes data to the USART. This function is blocking if the driver was opened by the client for blocking operation. This function will not block if the driver was opened by the client for non blocking operation. If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_BLOCKING, this function will only return when (or will block until) numbytes of bytes have been transmitted or if an error occurred. If there are buffers queued for writing, the function will wait until all the preceding buffers are completed. Ongoing buffer transmit operations will not be affected.

If the ioIntent parameter at the time of opening the driver was DRV_IO_INTENT_NON_BLOCKING, this function will return with the number of bytes that were actually accepted for transmission. The function will not wait until numBytes of bytes have been transmitted. If there a buffers queued for transmit, the function will not wait and will return immediately with 0 bytes.

Remarks

This function is thread safe in a RTOS application. This function is thread safe in a RTOS application. If the driver is configured for polled operation, this it will not support blocking operation in a bare metal (non-RTOS) application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

DRV_IO_INTENT_WRITE or DRV_IO_INTENT_READWRITE must have been specified in the DRV_USART_Open call.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
char            myBuffer[MY_BUFFER_SIZE];
int    count;
unsigned int    total;

total = 0;
do
{
    count  = DRV_USART_Write(myUSARTHandle, &myBuffer[total],
                                            MY_BUFFER_SIZE - total);
    total += count;
```

```
    // Do something else...

} while( total < MY_BUFFER_SIZE );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
buffer	Buffer containing the data to written.
numbytes	size of the buffer

Function

size_t DRV_USART_Write
(
const DRV_HANDLE handle,
void * buffer,
const size_t numbytes
)

f) Byte Transfer Functions

DRV_USART_ReadByte Function

Reads a byte of data from the USART.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
uint8_t DRV_USART_ReadByte(const DRV_HANDLE handle);
```

Returns

A data byte received by the driver.

Description

This routine reads a byte of data from the USART.

Remarks

This function is thread safe when called in a RTOS application. Note that DRV_USART_WriteByte and DRV_USART_ReadByte function cannot co-exist with DRV_USART_BufferAddRead, DRV_USART_BufferAddWrite, DRV_USART_Read and DRV_USART_Write functions in a application. Calling the DRV_USART_ReadByte and DRV_USART_WriteByte functions will disrupt the processing of any queued buffers.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

The transfer status should be checked to see if the receiver is not empty before calling this function.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
char            myBuffer[MY_BUFFER_SIZE];
unsigned int    numBytes;

numBytes = 0;
do
{
    if( DRV_USART_TRANSFER_STATUS_RECEIVER_DATA_PRESENT & DRV_USART_TransferStatus(myUSARTHandle) )
    {
        myBuffer[numBytes++] = DRV_USART_ReadByte(myUSARTHandle);
    }

    // Do something else...
```

```
} while( numBytes < MY_BUFFER_SIZE);
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

uint8_t DRV_USART_ReadByte(const DRV_HANDLE handle)

DRV_USART_WriteByte Function

Writes a byte of data to the USART.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
void DRV_USART_WriteByte(const DRV_HANDLE handle, const uint8_t byte);
```

Returns

None.

Description

This routine writes a byte of data to the USART.

Remarks

This function is thread safe when called in a RTOS application. Note that DRV_USART_WriteByte and DRV_USART_ReadByte function cannot co-exist with DRV_USART_BufferAddRead, DRV_USART_BufferAddWrite, DRV_USART_Read and DRV_USART_Write functions in a application. Calling the DRV_USART_ReadByte and DRV_USART_WriteByte function will disrupt the processing of any queued buffers.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

The transfer status should be checked to see if transmitter is not full before calling this function.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
char            myBuffer[MY_BUFFER_SIZE];
unsigned int    numBytes;

// Preinitialize myBuffer with MY_BUFFER_SIZE bytes of valid data.

numBytes = 0;
while( numBytes < MY_BUFFER_SIZE );
{
    if( !(DRV_USART_TRANSFER_STATUS_TRANSMIT_FULL & DRV_USART_TransferStatus(myUSARTHandle)) )
    {
        DRV_USART_WriteByte(myUSARTHandle, myBuffer[numBytes++]);
    }

    // Do something else...
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine
byte	Data byte to write to the USART

Function

void DRV_USART_WriteByte(const DRV_HANDLE handle, const uint8_t byte)

DRV_USART_TransmitBufferSizeGet Function

Returns the size of the transmit buffer.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
unsigned int DRV_USART_TransmitBufferSizeGet(const DRV_HANDLE handle);
```

Returns

Size of the driver's transmit buffer, in bytes.

Description

This routine returns the size of the transmit buffer and can be used by the application to determine the number of bytes to write with the DRV_USART_WriteByte function.

Remarks

Does not account for client queued buffers. This function is thread safe when used in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
const uint8_t   writeBuffer[5];
unsigned int    size, numBytes = 0;
unsigned int    writeBufferLen = sizeof(writeBuffer);

size     = DRV_USART_TransmitBufferSizeGet (myUSARTHandle);

// Do something based on the transmitter buffer size
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

unsigned int DRV_USART_TransmitBufferSizeGet (const DRV_HANDLE handle)

DRV_USART_ReceiverBufferSizeGet Function

Returns the size of the receive buffer.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
unsigned int DRV_USART_ReceiverBufferSizeGet(const DRV_HANDLE handle);
```

Returns

Size of the driver's receive buffer, in bytes.

Description

This routine returns the size of the receive buffer.

Remarks

Does not account for client queued buffers. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
const uint8_t   readBuffer[5];
unsigned int    size, numBytes = 0;
unsigned int    readbufferLen = sizeof(readBuffer);

size     = DRV_USART_ReceiverBufferSizeGet(myUSARTHandle);

// Do something based on the receiver buffer size
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

unsigned int DRV_USART_ReceiverBufferSizeGet(const DRV_HANDLE handle)

DRV_USART_TransferStatus Function

Returns the transmitter and receiver transfer status.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
DRV_USART_TRANSFER_STATUS DRV_USART_TransferStatus(const DRV_HANDLE handle);
```

Returns

A DRV_USART_TRANSFER_STATUS value describing the current status of the transfer.

Description

This returns the transmitter and receiver transfer status.

Remarks

The returned status may contain a value with more than one of the bits specified in the DRV_USART_TRANSFER_STATUS enumeration set. The caller should perform an "AND" with the bit of interest and verify if the result is non-zero (as shown in the example) to verify the desired status bit. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open

if (DRV_USART_TRANSFER_STATUS_RECEIVER_DATA_PRESENT & DRV_USART_TransferStatus(myUSARTHandle))
{
    // Data has been received that can be read
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

DRV_USART_TRANSFER_STATUS DRV_USART_TransferStatus(const DRV_HANDLE handle)

DRV_USART_TransmitBufferIsFull Function

Provides the status of the driver's transmit buffer.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
bool DRV_USART_TransmitBufferIsFull(const DRV_HANDLE handle);
```

Returns

true - if the transmit buffer is full

false - if the transmit buffer is not full

Description

This routine identifies if the driver's transmit buffer is full or not. This function can be used in conjunction with the DRV_USART_Write and DRV_USART_WriteByte functions.

Remarks

Does not account for client queued buffers. This function is thread safe when called in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
unsigned int    numBytes;
int             bytesToWrite;
const uint8_t   writeBuffer[35] = "1234567890ABCDEFGHIJKLMNOPn" ;
int             writebufferLen = strlen((char *)writeBuffer);

numBytes = 0;
while( numBytes < writebufferLen )
{
    if (DRV_USART_TransmitBufferisFull())
    {
        // Do something else until there is some room in the driver's Transmit buffer.
    }
    else
    {
        DRV_USART_WriteByte(myUSARTHandle, writeBuffer[numBytes++]);
    }
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

bool DRV_USART_TransmitBufferIsFull(const DRV_HANDLE handle)

DRV_USART_ReceiverBufferIsEmpty Function

Provides the status of the driver's receive buffer.

Implementation: Static/Dynamic

File

drv_usart.h

C

```
bool DRV_USART_ReceiverBufferIsEmpty(const DRV_HANDLE handle);
```

Returns

true - if the driver's receive buffer is empty

false - if the driver's receive buffer is not empty

Description

This routine indicates if the driver's receiver buffer is empty. This function can be used in conjunction with the DRV_USART_Read and DRV_USART_ReadByte functions.

Remarks

Does not account for client queued buffers. This function is safe thread safe when used in a RTOS application.

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

DRV_USART_Open must have been called to obtain a valid opened device handle.

Example

```
DRV_HANDLE               myUSARTHandle;    // Returned from DRV_USART_Open
char                     myBuffer[MY_BUFFER_SIZE];
unsigned int             numBytes;

numBytes = 0;
while( numBytes < MY_BUFFER_SIZE );
{
    if ( !DRV_USART_ReceiverBufferIsEmpty(myUSARTHandle) )
    {
        if( numBytes < MY_BUFFER_SIZE )
        {
            myBuffer[numBytes++] = DRV_USART_ReadByte (myUSARTHandle);
        }
        else
        {
            break;
        }
    }

    // Do something else while more data is received.
}
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from the driver's open routine

Function

bool DRV_USART_ReceiverBufferIsEmpty(const DRV_HANDLE handle)

DRV_USART_ByteErrorCallbackSet Function

Registers callback to handle for byte error events.

File

drv_usart.h

C

```
void DRV_USART_ByteErrorCallbackSet(const SYS_MODULE_INDEX index, const DRV_USART_BYTE_EVENT_HANDLER
eventHandler);
```

Returns

None.

Description

This function allows a callback function to be registered with the driver to handle the error events occurring in the transmit/receive path during byte transfers.

The callback function should be registered as part of the initialization. The callback functionality is available only in the interrupt mode of operation. The driver clears the interrupt after invoking the callback function.

Remarks

None

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];

// myUSARTHandle is the handle returned by the DRV_USART_Open function.
myUSARTHandle = DRV_USART_Open(DRV_USART_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
                                (uintptr_t)&myAppObj );

// Register an event handler with driver. This is done once
DRV_USART_ByteErrorCallbackSet (DRV_USART_INDEX_0, APP_USARTErrorEventHandler);

// Event Processing Technique.
void APP_USARTErrorEventHandler(const SYS_MODULE_INDEX index)
{
    // Error has occurred. Handle the event.
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
eventHandler	Pointer to the event handler function.

Function

void DRV_USART_ByteErrorCallbackSet

(

const SYS_MODULE_INDEX index,

const DRV_USART_BYTE_EVENT_HANDLER eventHandler

)

DRV_USART_ByteReceiveCallbackSet Function

Registers receive callback function for byte receive event.

File

drv_usart.h

C

```
void DRV_USART_ByteReceiveCallbackSet(const SYS_MODULE_INDEX index, const DRV_USART_BYTE_EVENT_HANDLER
eventHandler);
```

Returns

None.

Description

This function allows a receive callback function to be registered with the driver. The callback function is invoked when a byte has been received. The received byte can then be read using DRV_USART_ReadByte() function.

The callback function should be registered with the driver as part of the initialization. The callback functionality is available only in the interrupt mode of operation. The driver clears the interrupt after invoking the callback function.

Remarks

None

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];

// myUSARTHandle is the handle returned by the DRV_USART_Open function.
myUSARTHandle = DRV_USART_Open(DRV_USART_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
                                (uintptr_t)&myAppObj );

// Register an event handler with driver. This is done once
DRV_USART_ByteReceiveCallbackSet(DRV_USART_INDEX_0, APP_USARTReceiveEventHandler);

// Event Processing Technique. Event is received when
// a byte is received.

void APP_USARTReceiveEventHandler(const SYS_MODULE_INDEX index)
{
    // Byte has been Received. Handle the event.
    // Read byte using DRV_USART_ReadByte ()
    // DRV_USART_ReceiverBufferIsEmpty() function can be used to
    // check if the receiver buffer is empty.
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
eventHandler	Pointer to the event handler function.

Function

void DRV_USART_ByteReceiveCallbackSet

(

const SYS_MODULE_INDEX index,

const DRV_USART_BYTE_EVENT_HANDLER eventHandler

)

DRV_USART_ByteTransmitCallbackSet Function

Registers a callback function for byte transmit event.

File

drv_usart.h

C

```
void DRV_USART_ByteTransmitCallbackSet(const SYS_MODULE_INDEX index, const DRV_USART_BYTE_EVENT_HANDLER
eventHandler);
```

Returns

None.

Description

This function allows a transmit callback function to be registered with the driver. The callback function is invoked when a byte has been transmitted using DRV_USART_WriteByte () function.

The callback function should be registered with the driver prior to any writes to the driver. The callback functionality is available only in the interrupt mode of operation. The driver clears the interrupt after invoking the callback function.

Remarks

None

Preconditions

The DRV_USART_Initialize routine must have been called for the specified USART driver instance.

Example

```
// myAppObj is an application specific state data object.
MY_APP_OBJ myAppObj;

uint8_t mybuffer[MY_BUFFER_SIZE];

// myUSARTHandle is the handle returned by the DRV_USART_Open function.
myUSARTHandle = DRV_USART_Open(DRV_USART_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);
                                (uintptr_t)&myAppObj );

// Register an event handler with driver. This is done once
DRV_USART_ByteTransmitCallbackSet (DRV_USART_INDEX_0, APP_USARTTransmitEventHandler);

DRV_USART_WriteByte (myUSARThandle, myBuffer[0]);

// Event Processing Technique. Event is received when
// the byte is transmitted.

void APP_USARTTransmitEventHandler (const SYS_MODULE_INDEX index)
{
    // Byte has been transmitted. Handle the event.
}
```

Parameters

Parameters	Description
index	Identifier for the object instance to be opened
eventHandler	Pointer to the event handler function.

Function

void DRV_USART_ByteTransmitCallbackSet

(

const SYS_MODULE_INDEX index,

const DRV_USART_BYTE_EVENT_HANDLER eventHandler

)

Files

Files

Name	Description
drv_usart.h	USART Driver Interface Header File
drv_usart_config_template.h	USART Driver Configuration Template.

Description

This section lists the source and header files used by the USART Driver Library.

drv_usart.h

USART Driver Interface Header File

Functions

	Name	Description
	DRV_USART_AddressedBufferAddWrite	Schedule a non-blocking addressed driver write operation. **Implementation:** Dynamic

	DRV_USART_BaudSet	This function changes the USART module baud to the specified value. **Implementation:** Static/Dynamic
	DRV_USART_BufferAddRead	Schedule a non-blocking driver read operation. **Implementation:** Static/Dynamic
	DRV_USART_BufferAddWrite	Schedule a non-blocking driver write operation. **Implementation:** Static/Dynamic
	DRV_USART_BufferCompletedBytesGet	Returns the number of bytes that have been processed for the specified buffer. **Implementation:** Static/Dynamic
	DRV_USART_BufferEventHandlerSet	Allows a client to identify a buffer event handling function for the driver to call back when queued buffer transfers have finished. **Implementation:** Static/Dynamic
	DRV_USART_BufferProcessedSizeGet	This API will be deprecated and not recommended to use. Use DRV_USART_BufferCompletedBytesGet to get the number of bytes processed for the specified buffer.
	DRV_USART_BufferRemove	Removes a requested buffer from the queue. **Implementation:** Static/Dynamic
	DRV_USART_ByteErrorCallbackSet	Registers callback to handle for byte error events.
	DRV_USART_ByteReceiveCallbackSet	Registers receive callback function for byte receive event.
	DRV_USART_ByteTransmitCallbackSet	Registers a callback function for byte transmit event.
	DRV_USART_ClientStatus	Gets the current client-specific status the USART driver. **Implementation:** Static/Dynamic
	DRV_USART_Close	Closes an opened-instance of the USART driver. **Implementation:** Static/Dynamic
	DRV_USART_Deinitialize	Deinitializes the specified instance of the USART driver module. **Implementation:** Static/Dynamic
	DRV_USART_ErrorGet	This function returns the error(if any) associated with the last client request. **Implementation:** Static/Dynamic
	DRV_USART_Initialize	Initializes the USART instance for the specified driver index. **Implementation:** Static/Dynamic
	DRV_USART_LineControlSet	This function changes the USART module line control to the specified value. **Implementation:** Static/Dynamic
	DRV_USART_Open	Opens the specified USART driver instance and returns a handle to it. **Implementation:** Static/Dynamic
	DRV_USART_Read	Reads data from the USART. **Implementation:** Static/Dynamic
	DRV_USART_ReadByte	Reads a byte of data from the USART. **Implementation:** Static/Dynamic
	DRV_USART_ReceiverBufferIsEmpty	Provides the status of the driver's receive buffer. **Implementation:** Static/Dynamic
	DRV_USART_ReceiverBufferSizeGet	Returns the size of the receive buffer. **Implementation:** Static/Dynamic
	DRV_USART_Status	Gets the current status of the USART driver module. **Implementation:** Static/Dynamic
	DRV_USART_TasksError	Maintains the driver's error state machine and implements its ISR. **Implementation:** Static/Dynamic
	DRV_USART_TasksReceive	Maintains the driver's receive state machine and implements its ISR. **Implementation:** Static/Dynamic
	DRV_USART_TasksTransmit	Maintains the driver's transmit state machine and implements its ISR. **Implementation:** Static/Dynamic
	DRV_USART_TransferStatus	Returns the transmitter and receiver transfer status. **Implementation:** Static/Dynamic
	DRV_USART_TransmitBufferIsFull	Provides the status of the driver's transmit buffer. **Implementation:** Static/Dynamic
	DRV_USART_TransmitBufferSizeGet	Returns the size of the transmit buffer. **Implementation:** Static/Dynamic
	DRV_USART_Write	Writes data to the USART. **Implementation:** Static/Dynamic
	DRV_USART_WriteByte	Writes a byte of data to the USART. **Implementation:** Static/Dynamic

Description

USART Driver Interface Header File

The USART device driver provides a simple interface to manage the USART or UART modules on Microchip microcontrollers. This file provides the interface definition for the USART driver.

File Name

drv_usart.h

Company

Microchip Technology Inc.

drv_usart_config_template.h

USART Driver Configuration Template.

Macros

	Name	Description
	DRV_USART_BAUD_RATE_IDXn	Specifies the USART Baud at which the USART driver is initialized.
	DRV_USART_BUFFER_QUEUE_SUPPORT	Specifies if the Buffer Queue support should be enabled.
	DRV_USART_BYTE_MODEL_BLOCKING	Enables or Disables DRV_USART_ByteWrite function blocking behavior.
	DRV_USART_BYTE_MODEL_CALLBACK	Enables or Disables Callback Feature of the Byte Transfer Model.
	DRV_USART_BYTE_MODEL_SUPPORT	Specifies if the Byte Model support should be enabled.
	DRV_USART_CLIENTS_NUMBER	Sets up the maximum number of clients that can be connected to any hardware instance.
	DRV_USART_INDEX	USART Static Index selection.
	DRV_USART_INSTANCES_NUMBER	Sets up the maximum number of hardware instances that can be supported.
	DRV_USART_INTERRUPT_MODE	Macro controls interrupt based operation of the driver.
	DRV_USART_INTERRUPT_SOURCE_ERROR	Defines the error interrupt source for the static driver.
	DRV_USART_INTERRUPT_SOURCE_RECEIVE	Defines the Receive interrupt source for the static driver.
	DRV_USART_INTERRUPT_SOURCE_RECEIVE_DMA	Defines the Receive DMA Channel interrupt source for the static driver.
	DRV_USART_INTERRUPT_SOURCE_TRANSMIT	Defines the Transmit interrupt source for the static driver.
	DRV_USART_INTERRUPT_SOURCE_TRANSMIT_DMA	Defines the Transmit DMA Channel interrupt source for the static driver.
	DRV_USART_PERIPHERAL_ID	Configures the USART PLIB Module ID.
	DRV_USART_QUEUE_DEPTH_COMBINED	Defines the number of entries of all queues in all instances of the driver.
	DRV_USART_RCV_QUEUE_SIZE_IDXn	Sets the USART Driver Receive Queue Size while using the Buffer Queue Data Transfer Model.
	DRV_USART_READ_WRITE_MODEL_SUPPORT	Specifies if Read/Write Model support should be enabled.
	DRV_USART_RECEIVE_DMA	Defines the USART Driver Receive DMA Channel for the static driver.
	DRV_USART_TRANSMIT_DMA	Defines the USART Driver Transmit DMA Channel in case of static driver.
	DRV_USART_XMIT_QUEUE_SIZE_IDXn	Sets the USART Driver Transmit Queue Size while using the Buffer Queue Data Transfer Model.

Description

USART Driver Configuration Template

These file provides the list of all the configurations that can be used with the driver. This file should not be included in the driver.

File Name

drv_usart_config_template.h

Company

Microchip Technology Inc.

Wi-Fi Driver Libraries

This section describes the Wi-Fi Driver Libraries available in MPLAB Harmony.

Description

MRF24WN0MA Wi-Fi PICtail/PICtail Plus Daughter Board: Part number - AC164153
http://www.microchip.com/Developmenttools/ProductDetails.aspx?PartNO=AC164153

The following table lists the library files available for the Wi-Fi Drivers.

Wi-Fi Library File Matrix

Wi-Fi Device	PIC32MX795F512L	PIC32MZ2048ECH144	PIC32MZ2048EFM144
MRF24WN	wdrvext_mx.a	wdrvext_mz_ec.a	wdrvext_mz_ef.a

MRF24WN Wi-Fi Driver Library

This topic describes the MRF24WN Wi-Fi Driver Library.

Description

The following table lists the library files available for the MRF24WN Wi-Fi Driver.

Wi-Fi Library File Matrix	Target MCU Device		
	PIC32MX795F512L	PIC32MZ2048ECH144	PIC32MZ2048EFM144
Wi-Fi Device MRF24WN	wdrvext_mx.a	wdrvext_mz_ec.a	wdrvext_mz_ef.a

Introduction

This library provides a low-level abstraction of the MRF24WN Wi-Fi Driver Library that is available on the Microchip family of microcontrollers with a convenient C language interface. It can be used to simplify low-level access to the module without the necessity of interacting directly with the module's registers, there by hiding differences from one microcontroller variant to another.

Description

The MRF24WN Wi-Fi Driver Library, in conjunction with the MRF24WN module, allows an application to:

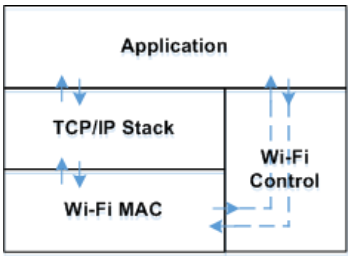
- Join an existing 802.11 Wi-Fi Infrastructure network
- Create a 802.11 Wi-Fi Ad Hoc or Soft AP network

The following application services are provided by the Wi-Fi library:

- Configuring Wi-Fi connection (SSID, security mode, channel list, etc.)
- Join an existing Wi-Fi Infrastructure network
- Create a Wi-Fi Ad Hoc or Soft AP network
- Scan for Wi-Fi Access Point (AP) or Soft AP
- Getting Wi-Fi network status
- Wi-Fi power control
- Wi-Fi console commands

The MAC layer services are not directly accessible to the application; this portion of the code resides under the TCP/IP Stack MAC module software layers and is used by stack services to transmit and receive data over a Wi-Fi network. The following diagram shows the interaction of the primary software blocks in a Wi-Fi application.

Wi-Fi Software Block Diagram



The following table provides information that includes network mode and security mode support by MRF24WN Wi-Fi Driver.

MRF24WN Network Connection Matrix	Network Mode	
	Infrastructure	Soft AP
Security Mode		
Open	YES	YES
WEP40	YES	YES
WEP104	YES	YES
WPA-PSK	YES	YES
WPA2-PSK	YES	YES
WPS Push Button	YES	NA
WPS PIN	YES	NA

Using the Library

This topic describes the basic architecture of the MRF24WN Wi-Fi Driver Library and provides information and examples on its use.

Description

Interface Header Files: `wdrv_mrf24wn_common.h` and `wdrv_mrf24wn_api.h`

The interface to the MRF24WN Wi-Fi Driver Library is defined in the `wdrv_mrf24wn_common.h` and wdrv_mrf24wn_api.h header files.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the MRF24WN Wi-Fi module with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The MRF24WN Wi-Fi Library provides the following functionality:

- Wi-Fi library initialization
- Wi-Fi network configuration
- Wi-Fi network connection
- Scanning for existing Wi-Fi networks
- Wi-Fi event processing
- Wi-Fi status
- Wi-FI console commands

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The Library Interface functions are divided into various sub-sections, which address one of the blocks or the overall operation of the Wi-Fi module.

Library Interface Section	Description
Wi-Fi Initialization Functions	This section provides functions that initialize the Wi-Fi library and allow its API to be used.
Wi-Fi Status Functions	This section provides functions that retrieve the Wi-Fi connection status.
Wi-Fi External Functions	This section provides public functions accessible to TCP/IP applications.
Other Functions	This section provides additional miscellaneous functions for configuring the Wi-Fi connection.

How the Library Works

This section describes how the MRF24WN Wi-Fi Driver Library operates.

Description

Before the driver is ready for use, its should be configured (compile time configuration).

There are few run-time configuration items that are done during initialization of the driver instance, and a few that are client-specific and are done using dedicated functions.

To use the MRF24WN Wi-Fi Driver, initialization and client functions should be invoked in a specific sequence to ensure correct operation.

System Initialization

This section describes initialization and reinitialization features.

Description

Wi-Fi initialization configures the MRF24WN module and then directs it to join (or create) a Wi-Fi network. The MRF24WN module defaults to open security and scans all channels in the domain. Therefore, to initialize and connect with the minimum function call overhead in an open security network, the following functions can be used:

```
WDRV_EXT_CmdSSIDSet("MySsidName",strlen("MySsidName");
WDRV_EXT_CmdConnect();        // start the connection process
```

Alternatively, the following functions could be used to achieve the same effect:

```
WDRV_EXT_CmdNetModeBSSSet();
WDRV_EXT_CmdSecNoneSet();
WDRV_EXT_CmdSSIDSet("MySsidName",strlen("MySsidName");
WDRV_EXT_CmdConnect();
```

Client Functionality

This section describes core operation.

Description

From the client perspective, once Wi-Fi initialization is complete and the connection process has started, the client responds to Wi-Fi events. The client is notified of events by the callback function WDRV_ProcessEvent. The parameters into that function are `event` and `eventInfo`, where `event` is the event code and `eventInfo` is additional information about the event.

Wi-Fi Connection Events

```
/*No Wi-Fi connection exists*/
WDRV_CSTATE_NOT_CONNECTED = 1,

/*Wi-Fi connection in progress*/
WDRV_CSTATE_CONNECTION_IN_PROGRESS = 2,

/*Wi-Fi connected in infrastructure mode*/
WDRV_CSTATE_CONNECTED_INFRASTRUCTURE = 3,

/*Wi-Fi connected in adHoc mode*/
WDRV_CSTATE_CONNECTED_ADHOC = 4,

/*Wi-Fi in process of reconnecting*/
WDRV_CSTATE_RECONNECTION_IN_PROGRESS = 5,

/*Wi-Fi connection temporarily lost*/
WDRV_CSTATE_CONNECTION_TEMPORARY_LOST = 6,

/*Wi-Fi connection permanently lost*/
WDRV_CSTATE_CONNECTION_PERMANENTLY_LOST = 7
```

Scan Events

```
WDRV_SOFTAP_EVENT_CONNECTED = 0,
WDRV_SOFTAP_EVENT_DISCONNECTED = 1
```

Key Events

```
WDRV_SOFTAP_EVENT_LINK_LOST = 0,
WDRV_SOFTAP_EVENT_RECEIVED_DEAUTH = 1
```

Disconnect Events

```
WDRV_DISCONNECT_REASON_NO_NETWORK_AVAIL = 0x01,
WDRV_DISCONNECT_REASON_LOST_LINK = 0x02,
WDRV_DISCONNECT_REASON_DISCONNECT_CMD = 0x03,
```

```
WDRV_DISCONNECT_REASON_BSS_DISCONNECTED = 0x04,
WDRV_DISCONNECT_REASON_AUTH_FAILED = 0x05,
WDRV_DISCONNECT_REASON_ASSOC_FAILED = 0x06,
WDRV_DISCONNECT_REASON_NO_RESOURCES_AVAIL = 0x07,
WDRV_DISCONNECT_REASON_CONNECTION_DENIED = 0x08,
WDRV_DISCONNECT_REASON_INVALID_PROFILE = 0x0A,
WDRV_DISCONNECT_REASON_PROFILE_MISMATCH = 0x0C,
WDRV_DISCONNECT_REASON_CONNECTION_EVICTED = 0x0d
```

Configuring the Library

The configuration of the MRF24WN Wi-Fi Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the Wi-Fi Driver. Based on the selections made, the MRF24WN Wi-Fi Driver may support the selected features. These configuration settings will apply to all instances of the MRF24WN Wi-Fi Driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Sample Functionality

The following code provides an example of Wi-Fi Driver configuration.

```
/*** Wi-Fi Driver Configuration ***/

#define WIFI_USE_RTOS

#define WDRV_EXT_INIT_TASK_STACK_SIZE 512u
#define WDRV_EXT_INIT_TASK_PRIO 6u
#define WDRV_EXT_MAIN_TASK_STACK_SIZE 2048u
#define WDRV_EXT_MAIN_TASK_PRIO 7u

#define WDRV_ASSERT(condition, msg) WDRV_Assert(condition, msg, __FILE__, __LINE__)

#define DRV_WIFI_SPI_INDEX 0
#define DRV_WIFI_SPI_INSTANCE sysObj.spiObjectIdx0

#define DRV_WIFI_NVM_SPACE_ENABLE
#define DRV_WIFI_NVM_SPACE_ADDR (48*1024)

#define MRF_INT_SOURCE INT_SOURCE_EXTERNAL_1
#define MRF_INT_VECTOR INT_VECTOR_INT1

// IO mapping for general control pins, including CS, RESET and HIBERNATE
// MRF24W in SPI 1 slot
#define WF_CS_PORT_CHANNEL PORT_CHANNEL_E
#define WF_CS_BIT_POS      9

#define WF_RESET_PORT_CHANNEL PORT_CHANNEL_F
#define WF_RESET_BIT_POS      0

#define WF_HIBERNATE_PORT_CHANNEL PORT_CHANNEL_F
#define WF_HIBERNATE_BIT_POS      1

#define WF_INT_PRIORITY     3
#define WF_INT_SUBPRIORITY  1
#define WF_INT_PORT_CHANNEL PORT_CHANNEL_E
#define WF_INT_BIT_POS      8

#define WDRV_DEFAULT_NETWORK_TYPE WDRV_NETWORK_TYPE_INFRASTRUCTURE
#define WDRV_DEFAULT_SSID_NAME "MicrochipDemoApp"

#define WDRV_DEFAULT_WIFI_SECURITY_MODE WDRV_SECURITY_OPEN
#define WDRV_DEFAULT_WEP_KEYS_40 "5AFB6C8E77" // default WEP40 key
#define WDRV_DEFAULT_WEP_KEYS_104 "90E96780C739409DA50034FCAA" // default WEP104 key
#define WDRV_DEFAULT_PSK_PHRASE "Microchip 802.11 Secret PSK Password" // default WPA-PSK or WPA2-PSK
passphrase
#define WDRV_DEFAULT_WPS_PIN "12390212" // default WPS PIN
```

```
#define WDRV_DEFAULT_CHANNEL 6
#define WDRV_DEFAULT_POWER_SAVE WDRV_FUNC_DISABLED
```

Building the Library

This section lists the files that are available in the MRF24WN Wi-Fi Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/wifi/mrf24wn`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
`wdrv_mrf24wn_common.h`	Contains all data types, define constants for the MRF24WN Wi-Fi Driver.
`wdrv_mrf24wn_api.h`	Contains function prototypes for interfacing to the MRF24WN Wi-Fi Driver.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
`wdrv_mrf24wn_cli.c`	Provides access to MRF24WN Wi-Fi Driver controller.
`wdrv_mrf24wn_config_data.c`	Stores and retrieves MRF24WN Wi-Fi Driver configuration information in Non-volatile Memory (NVM).
`wdrv_mrf24wn_connmgr.c`	Provides access to MRF24WN Wi-Fi Driver controller for connection manager.
`wdrv_mrf24wn_events.c`	Provides access to MRF24WN Wi-Fi Driver controller for MAC events.
`wdrv_mrf24wn_iwpriv.c`	Provides functions to configure optional (private) parameters of the MRF24WN Wi-Fi Driver.
`wdrv_mrf24wn_main.c`	Module for Microchip TCP/IP Stack PIC32 implementation for multiple Wi-Fi MAC support.
`wdrv_mrf24wn_misc.c`	Miscellaneous support functions and data types for the MRF24WN Wi-Fi Driver.
`wdrv_mrf24wn_osal.c`	RTOS wrapper functions for the MRF24WN Wi-Fi Driver.
`wdrv_mrf24wn_scan_helper.c`	Provides helper functions to access scan results.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	The MRF24WN Wi-Fi Driver controller has no optional files.

Module Dependencies

The MRF24WN Wi-Fi Driver Library depends on the following modules:

- SPI Driver Library
- NVM Driver Library
- UART Driver Library
- USB Driver Library
- Operating System Abstraction Layer (OSAL) Library Help
- Clock System Service Library
- System Service Library Introduction
- Console System Service Library
- File System Service Library
- Interrupt System Service Library
- Timer System Service Library
- Debug System Service Library
- Ports System Service Library

- FreeRTOS Library Help
- Crypto Library
- Peripheral Libraries

Console Commands

This section describes the console commands available for the MRF24WN Wi-Fi Driver.

Description

Both the Web Server and the EasyConfig demonstrations support the followings commands, which enable control over the Wi-Fi settings.

Command: deleteconf

Parameters	Description
None.	Wi-Fi console command to erase saved Wi-Fi configuration in memory.

Command: iwconfig

Parameters	Description
[ssid <name>]	name: Specifies the name of the SSID (1-32 ASCII characters).
[mode <idle | managed>]	idle: Disconnected from the current configuration. managed: Connects in infrastructure mode to the currently set SSID.
[power <enable | disable>]	enable: Enables all Power-Saving features (PS_POLL). Will wake up to check for all types of traffic (unicast, multicast, and broadcast). disable: Disables any Power-Saving features. Will always be in an active power state.

[security <mode>]	mode: open/wep40/wep104/wpa/wpa2/pin/pbc. For example:
	`iwconfig security open`
	`iwconfig security wep40 <key>`
	`iwconfig security wep104 <key>`
	`iwconfig security wpa <key>`
	`iwconfig security wpa2 <key>`
	`iwconfig security pin <pin>`
	`iwconfig security pbc`

[scan]	Starts a Wi-Fi scan.
[scanget <scan_index>]	scan_index: Retrieves the scan result after the scan completes (1 - n).

Command: mac

Parameters	Description
None.	Wi-Fi console command to retrieve the MAC address of the MRF24WN module.

Command: readconf

Parameters	Description
None.	Wi-Fi console command to read saved Wi-Fi configuration in memory.

Command: saveconf

Parameters	Description
None.	Wi-Fi console command to save Wi-Fi configuration to memory.

Library Interface

a) Wi-Fi Initialization Functions

	Name	Description
	WDRV_SPI_In	Receives data from the module through the SPI bus. **Implementation:** Dynamic

	WDRV_SPI_Out	Sends data out to the module through the SPI bus. **Implementation:** Dynamic
	WDRV_GPIO_Init	Initializes the GPIO objects for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_GPIO_PowerOff	Powers off the MRF24WN module. **Implementation:** Dynamic
	WDRV_GPIO_PowerOn	Powers on the MRF24WN module. **Implementation:** Dynamic
	WDRV_IsPowerOff	Checks if MRF24WN is turned off. **Implementation:** Dynamic
	WDRV_MRF24WN_ISR	Wi-Fi driver (MRF24WN-specific) interrupt service routine. **Implementation:** Dynamic

b) Wi-Fi Status Functions

	Name	Description
	WDRV_EXT_CmdConnectContextChannelGet	Gets the AP channel **Implementation:** Dynamic
	WDRV_EXT_CmdPowerSaveGet	Retrieves current power save status. **Implementation:** Dynamic
	WDRV_EXT_ScanResultGet	Reads the selected scan results back from the MRF24WN module. **Implementation:** Dynamic

c) External Functions

	Name	Description
	WDRV_EXT_CmdNetModeIBSSSet	Sets the Wi-Fi network type to Adhoc. **Implementation:** Dynamic
	WDRV_EXT_CmdSecWPA2Set	Sets Wi-Fi security to WPA2. **Implementation:** Dynamic
	WDRV_EXT_Initialize	Initializes the MRF24WN Wi-Fi driver. **Implementation:** Dynamic
	WDRV_EXT_Initialize	Initializes the WINC1500 Wi-Fi driver. **Implementation:** Dynamic
	WDRV_EXT_PrivConfig	Configures g_wdrvext_priv parameter. **Implementation:** Dynamic

e) Private Configuration Functions

	Name	Description
	iwpriv_config_write	Writes to the Wi-Fi context configuration which is currently used by Wi-Fi driver. **Implementation:** Dynamic
	iwpriv_connstatus_get	Gets the Wi-Fi connection status. **Implementation:** Dynamic
	iwpriv_devinfo_get	Gets the device information. **Implementation:** Dynamic
	iwpriv_initialconn_set	Sets the initial connection status of Wi-Fi driver. **Implementation:** Dynamic
	iwpriv_initstatus_get	Gets the initialization status of Wi-Fi driver. **Implementation:** Dynamic
	iwpriv_is_servermode	Checks if the passed Wi-Fi context configuration is operating in server mode. **Implementation:** Dynamic
	iwpriv_leftclient_get	Gets the left client's information. **Implementation:** Dynamic
	iwpriv_mcastfilter_set	Adds a MAC address to the multi-cast filter. **Implementation:** Dynamic
	iwpriv_nettype_get	Gets the current network type. **Implementation:** Dynamic
	iwpriv_nettype_set	Sets the current network type. **Implementation:** Dynamic

	iwpriv_numberofscanresults_get	Gets the number of scan results. **Implementation:** Dynamic
	iwpriv_powersave_config	Enables or disables Power Save mode in Wi-Fi driver. **Implementation:** Dynamic
	iwpriv_prescan_start	Starts prescan. **Implementation:** Dynamic
	iwpriv_scan_start	Starts scan. **Implementation:** Dynamic
	iwpriv_scanstatus_get	Gets the prescan status. **Implementation:** Dynamic
	iwpriv_ssid_get	Gets the current SSID. **Implementation:** Dynamic
	iwpriv_ssid_set	Sets the current SSID. **Implementation:** Dynamic
	iwpriv_execute	This is function iwpriv_execute.
	iwpriv_get	This is function iwpriv_get.
	iwpriv_prescan_isfinished	Checks if the prescan is complete. **Implementation:** Dynamic
	iwpriv_prescan_option_get	To see if prescan will run before next connection. **Implementation:** Dynamic
	iwpriv_prescan_option_set	To run prescan or not. **Implementation:** Dynamic
	iwpriv_set	This is function iwpriv_set.
	iwpriv_adhocctx_set	Sets the Ad hoc network context information. **Implementation:** Dynamic
	iwpriv_config_read	Reads the Wi-Fi context configuration. **Implementation:** Dynamic

f) Data Types and Constants

	Name	Description
	IWPRIV_CONN_STATUS	This is type IWPRIV_CONN_STATUS.
	IWPRIV_STATUS	This is type IWPRIV_STATUS.
	IWPRIV_CMD	This is type IWPRIV_CMD.
	IWPRIV_EXECUTE_PARAM	This is type IWPRIV_EXECUTE_PARAM.
	IWPRIV_GET_PARAM	This is type IWPRIV_GET_PARAM.
	IWPRIV_PARAM_CLIENTINFO	This is type IWPRIV_PARAM_CLIENTINFO.
	IWPRIV_PARAM_CONTEXT	This is type IWPRIV_PARAM_CONTEXT.
	IWPRIV_PARAM_DEVICEINFO	This is type IWPRIV_PARAM_DEVICEINFO.
	IWPRIV_SCAN_STATUS	This is type IWPRIV_SCAN_STATUS.
	IWPRIV_SET_PARAM	This is type IWPRIV_SET_PARAM.
	IWPRIV_PARAM_CONFIG	This is type IWPRIV_PARAM_CONFIG.
	IWPRIV_PARAM_CONNECT	This is type IWPRIV_PARAM_CONNECT.
	IWPRIV_PARAM_DRIVERSTATUS	This is type IWPRIV_PARAM_DRIVERSTATUS.
	IWPRIV_PARAM_FWUPGRADE	This is type IWPRIV_PARAM_FWUPGRADE.
	IWPRIV_PARAM_MULTICASTFILTER	This is type IWPRIV_PARAM_MULTICASTFILTER.
	IWPRIV_PARAM_NETWORKTYPE	This is type IWPRIV_PARAM_NETWORKTYPE.
	IWPRIV_PARAM_OPERATIONMODE	This is type IWPRIV_PARAM_OPERATIONMODE.
	IWPRIV_PARAM_POWERSAVE	This is type IWPRIV_PARAM_POWERSAVE.
	IWPRIV_PARAM_SCAN	This is type IWPRIV_PARAM_SCAN.
	IWPRIV_PARAM_SSID	This is type IWPRIV_PARAM_SSID.

Description

This section describes the Application Programming Interface (API) functions of the MRF24WN Wi-Fi Driver.

Refer to each section for a detailed description.

a) Wi-Fi Initialization Functions

WDRV_SPI_In Function

Receives data from the module through the SPI bus.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_SPI_In(uint8_t *const OutBuf, uint16_t OutSize, uint8_t *const InBuf, uint16_t InSize);
```

Returns

None.

Description

This function receives data from the module through the SPI bus.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Parameters

Parameters	Description
bufOut	buffer pointer of output command
OutSize	the command size
InBuf	buffer pointer of input data
InSize	the input data size

Function

void WDRV_SPI_In(uint8_t const *const OutBuf, uint16_t OutSize,
uint8_t *const InBuf, uint16_t InSize)

WDRV_SPI_Out Function

Sends data out to the module through the SPI bus.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_SPI_Out(uint8_t *const bufOut, uint16_t OutSize);
```

Returns

None.

Description

This function sends data out to the module through the SPI bus.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Parameters

Parameters	Description
bufOut	buffer pointer of output data

OutSize	the data size

Function

void WDRV_SPI_Out(uint8_t const *const bufOut, uint16_t OutSize)

WDRV_GPIO_Init Function

Initializes the GPIO objects for the Wi-Fi driver.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_GPIO_Init();
```

Returns

None.

Description

This function initializes the GPIO objects for the Wi-Fi driver.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_GPIO_Init(void)

WDRV_GPIO_PowerOff Function

Powers off the MRF24WN module.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_GPIO_PowerOff();
```

Returns

None.

Description

This function powers off the MRF24WN module.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_GPIO_PowerOff(void)

WDRV_GPIO_PowerOn Function

Powers on the MRF24WN module.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_GPIO_PowerOn();
```

Returns

None.

Description

This function powers on the MRF24WN module.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_GPIO_PowerOn(void)

WDRV_IsPowerOff Function

Checks if MRF24WN is turned off.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
bool WDRV_IsPowerOff();
```

Returns

- 0 - Indicates that MRF24WN is turned off
- Non-zero value - Indicates that MRF24WN is on

Description

This function checks if MRF24WN is turned off.

Remarks

None.

Function

bool WDRV_IsPowerOff(void)

WDRV_MRF24WN_ISR Function

Wi-Fi driver (MRF24WN-specific) interrupt service routine.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_MRF24WN_ISR();
```

Returns

None.

Description

This function is the Wi-Fi driver (MRF24WN-specific) interrupt service routine.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void WDRV_MRF24WN_ISR(void)

b) Wi-Fi Status Functions

WDRV_EXT_CmdConnectContextChannelGet Function

Gets the AP channel

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
uint32_t WDRV_EXT_CmdConnectContextChannelGet(uint16_t * bssChannel);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function gets the current AP channel.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
bssChannel	pointer where the current AP channel will be written

Function

uint32_t WDRV_EXT_CmdConnectContextChannelGet(uint16_t *bssChannel)

WDRV_EXT_CmdPowerSaveGet Function

Retrieves current power save status.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
uint32_t WDRV_EXT_CmdPowerSaveGet(bool * enabled);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function retrieves the current power save status.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
enabled	pointer where the current power save status will be written

Function

uint32_t WDRV_EXT_CmdPowerSaveGet(bool *enabled)

WDRV_EXT_ScanResultGet Function

Reads the selected scan results back from the MRF24WN module.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_EXT_ScanResultGet(uint16_t idx, WDRV_SCAN_RESULT * p_scanResult);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

After a scan has completed this function is used to read one scan result at a time from the MRF24WN module.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
listIndex	index (0 based list) of the scan entry to retrieve
p_scanResult	pointer to where scan result is written

Function

void WDRV_EXT_ScanResultGet(uint8_t listIndex, WDRV_SCAN_RESULT *p_scanResult)

c) External Functions

WDRV_EXT_CmdNetModeIBSSSet Function

Sets the Wi-Fi network type to Adhoc.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
uint32_t WDRV_EXT_CmdNetModeIBSSSet();
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function sets the Wi-Fi network type to Adhoc.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

uint32_t WDRV_EXT_CmdNetModeIBSSSet(void)

WDRV_EXT_CmdSecWPA2Set Function

Sets Wi-Fi security to WPA2.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
uint32_t WDRV_EXT_CmdSecWPA2Set(uint8_t * key, uint16_t len);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function sets the Wi-Fi security to WPA2. One can only connect to an AP that is running the same WPA2 mode.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete and in an unconnected state.

Parameters

Parameters	Description
key	pointer to the WPA2 key buffer
len	WPA2 key length

Function

uint32_t WDRV_EXT_CmdSecWPA2Set(uint8_t *key, uint16_t len)

WDRV_EXT_Initialize Function

Initializes the MRF24WN Wi-Fi driver.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
int32_t WDRV_EXT_Initialize(const WDRV_CALLBACKS *const CB);
```

Returns

- 0 - Indicates success

- non-zero value - Indicates failure

Description

This function initializes the MRF24WN Wi-Fi driver, making it ready for clients to use.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
CB	pointer to callback functions

Function

int32_t WDRV_EXT_Initialize(const WDRV_CALLBACKS *const CB)

WDRV_EXT_Initialize Function

Initializes the WINC1500 Wi-Fi driver.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_Initialize(const WDRV_HOOKS *const ehooks, bool initWait);
```

Returns

None.

Description

This function initializes the WINC1500 Wi-Fi driver, making it ready for clients to use.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
ehooks	pointer to WDRV layer hooks
initWait	true will put WDRV in wait during initialization

Function

void WDRV_EXT_Initialize(const WDRV_HOOKS *const ehooks, bool initWait)

WDRV_EXT_PrivConfig Function

Configures g_wdrvext_priv parameter.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_EXT_PrivConfig(uint32_t * config);
```

Returns

None.

Description

This function configures g_wdrvext_priv parameter.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
config	pointer to the parameter array

Function

void WDRV_EXT_PrivConfig(uint32_t *config)

d) GPIO Functions

e) Private Configuration Functions

iwpriv_config_write Function

Writes to the Wi-Fi context configuration which is currently used by Wi-Fi driver.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_config_write(void * wifi_cfg);
```

Returns

None.

Description

This function reads from a passed pointer, copies everything from it, and writes to the Wi-Fi context configuration, which is currently used by the Wi-Fi driver.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
wifi_cfg	pointer to where the context configuration is stored

Function

void iwpriv_config_write(void *wifi_cfg)

iwpriv_connstatus_get Function

Gets the Wi-Fi connection status.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
IWPRIV_CONN_STATUS iwpriv_connstatus_get();
```

Returns

Status of current Wi-Fi connection. See the definition for the IWPRIV_CONN_STATUS structure.

Description

This function gets the Wi-Fi connection status.

Remarks

IWPRIV_CONNECTION_FAILED does not necessarily mean that the module fails to connect to the network. It stands on the application's perspective, and actually can be customized. For example, in the Web Server demonstrations's use case, WDRV_CSTATE_CONNECTION_PERMANENTLY_LOST is treated as a fail case and will trigger the application to restart.

Preconditions

Wi-Fi initialization must be complete.

Function

IWPRIV_CONN_STATUS iwpriv_connstatus_get(void)

iwpriv_devinfo_get Function

Gets the device information.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_devinfo_get(void * info);
```

Returns

None.

Description

This function returns the device information.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
info	pointer to where the device information is written

Function

void iwpriv_devinfo_get(void *info)

iwpriv_initialconn_set Function

Sets the initial connection status of Wi-Fi driver.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_initialconn_set(bool connect);
```

Returns

None.

Description

This function sets the initial connection status of Wi-Fi driver. After Wi-Fi initialization, it decides whether or not to start the Wi-Fi connection.

Remarks

This function is mainly used to implement prescan. It has to be called before Wi-Fi driver's initialization is finished to be effective.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
connect	boolean value which indicates whether or not to start an initial connect

Function

void iwpriv_initialconn_set(bool connect)

iwpriv_initstatus_get Function

Gets the initialization status of Wi-Fi driver.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
uint8_t iwpriv_initstatus_get();
```

Returns

Current initialization status of the Wi-Fi driver (IWPRIV_READY or IWPRIV_IN_PROGRESS).

Description

This function returns the initialization status of the Wi-Fi driver.

Remarks

None.

Preconditions

None.

Function

uint8_t iwpriv_initstatus_get(void)

iwpriv_is_servermode Function

Checks if the passed Wi-Fi context configuration is operating in server mode.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
bool iwpriv_is_servermode();
```

Returns

- true - Wi-Fi context configuration is operating in server mode

- false - Wi-Fi context configuration is not operating in server mode

Description

This function checks if the passed Wi-Fi context configuration is operating in server mode, which includes Ad hoc mode and SoftAP mode.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

bool iwpriv_is_servermode(void)

iwpriv_leftclient_get Function

Gets the left client's information.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_leftclient_get(bool * updated, TCPIP_MAC_ADDR * addr);
```

Returns

None.

Description

This function returns the left client's information when the Wi-Fi module works in server mode and has the DHCP Server enabled.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
updated	if the left client's information needs to be updated
addr	MAC address of the left client

Function

void iwpriv_leftclient_get(bool *updated, TCPIP_MAC_ADDR *addr)

iwpriv_mcastfilter_set Function

Adds a MAC address to the multi-cast filter.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
IWPRIV_STATUS iwpriv_mcastfilter_set(uint8_t * addr);
```

Returns

Status of the set operation, IWPRIV_READY or IWPRIV_ERROR. See definition for the IWPRIV_STATUS structure.

Description

This function adds a MAC address to the multi-cast filter.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
addr	pointer to where the MAC address is stored

Function

IWPRIV_STATUS iwpriv_mcastfilter_set(uint8_t *addr)

iwpriv_nettype_get Function

Gets the current network type.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_nettype_get(uint8_t * netType);
```

Returns

None.

Description

This function returns the current network type.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
netType	pointer to where the network type is written

Function

void iwpriv_nettype_get(uint8_t *netType)

iwpriv_nettype_set Function

Sets the current network type.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_nettype_set(uint8_t netType);
```

Returns

None.

Description

This function sets the current network type.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
netType	the network type to set

Function

void iwpriv_nettype_set(uint8_t netType)

iwpriv_numberofscanresults_get Function

Gets the number of scan results.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
uint16_t iwpriv_numberofscanresults_get();
```

Returns

Number of scan results.

Description

This function gets the number of scan results.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

uint16_t iwpriv_numberofscanresults_get(void)

iwpriv_powersave_config Function

Enables or disables Power Save mode in Wi-Fi driver.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_powersave_config(bool enabled);
```

Returns

None.

Description

This function enables or disables Power Save mode in Wi-Fi driver, which depends on the passed boolean value.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
enabled	boolean value which indicates to enable or disable Power Save mode in Wi-Fi driver

Function

void iwpriv_powersave_config(bool enabled)

iwpriv_prescan_start Function

Starts prescan.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_prescan_start();
```

Returns

None.

Description

This function directs the Wi-Fi driver to start a prescan.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void iwpriv_prescan_start(void)

iwpriv_scan_start Function

Starts scan.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_scan_start();
```

Returns

None.

Description

The function starts a Wi-Fi scan.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void iwpriv_scan_start(void)

iwpriv_scanstatus_get Function

Gets the prescan status.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
IWPRIV_SCAN_STATUS iwpriv_scanstatus_get();
```

Returns

Prescan status: IWPRIV_SCAN_IDLE, IWPRIV_SCAN_IN_PROGRESS, IWPRIV_SCAN_NO_AP_FOUND or IWPRIV_SCAN_SUCCESSFUL. See the definition for the IWPRIV_SCAN_STATUS structure.

Description

This function gets the prescan status.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

IWPRIV_SCAN_STATUS iwpriv_scanstatus_get(void)

iwpriv_ssid_get Function

Gets the current SSID.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_ssid_get(uint8_t * ssid, uint8_t * ssidLen);
```

Returns

None.

Description

This function returns the current SSID.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
ssid	pointer to where the SSID is written
ssidLen	pointer to where the SSID length is written

Function

void iwpriv_ssid_get(uint8_t *ssid, uint8_t *ssidLen)

iwpriv_ssid_set Function

Sets the current SSID.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_ssid_set(uint8_t * ssid, uint8_t ssidLen);
```

Returns

None.

Description

This function sets the current SSID.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
ssid	pointer to where the SSID is stored
ssidLen	pointer to where the SSID length is stored

Function

void iwpriv_ssid_set(uint8_t *ssid, uint8_t ssidLen)

iwpriv_execute Function

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_execute(IWPRIV_CMD cmd, IWPRIV_EXECUTE_PARAM * params);
```

Description

This is function iwpriv_execute.

iwpriv_get Function

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_get(IWPRIV_CMD cmd, IWPRIV_GET_PARAM * params);
```

Description

This is function iwpriv_get.

iwpriv_prescan_isfinished Function

Checks if the prescan is complete.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
bool iwpriv_prescan_isfinished();
```

Returns

None.

Description

This function checks if the prescan is complete.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

bool iwpriv_prescan_isfinished(void)

iwpriv_prescan_option_get Function

To see if prescan will run before next connection.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
bool iwpriv_prescan_option_get();
```

Returns

None.

Description

This function checks whether or not the prescan will run before next connection.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

bool iwpriv_prescan_option_get(void)

iwpriv_prescan_option_set Function

To run prescan or not.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_prescan_option_set(bool scan);
```

Returns

None.

Description

This function controls whether or not to run prescan.

Remarks

Prescan means the scan runs before the module is connected. It needs to use multiple functions in this file. Please refer to the Easy Configuration demonstration to see the correct usage of prescan.

After the the module is connected, MRF24WN module can also do regular scans. But it cannot perform a scan when the connection is in progress.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
scan	true: run prescan before next connection
false	do not run prescan before next connection

Function

void iwpriv_prescan_option_set(bool scan)

iwpriv_set Function

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_set(IWPRIV_CMD cmd, IWPRIV_SET_PARAM * params);
```

Description

This is function iwpriv_set.

iwpriv_adhocctx_set Function

Sets the Ad hoc network context information.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_adhocctx_set(void * p_cxt);
```

Returns

None.

Description

This function sets the current Ad hoc network context information by reading from a passed pointer.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
p_cxt	pointer to where the Ad hoc network context is stored

Function

void iwpriv_adhocctx_set(void *p_cxt)

iwpriv_config_read Function

Reads the Wi-Fi context configuration.

Implementation: Dynamic

File

wdrv_mrf24wn_iwpriv.h

C

```
void iwpriv_config_read(void * wifi_cfg);
```

Returns

None.

Description

This function reads the current Wi-Fi context configuration, copies and stores the whole structure to the pointer passed to the function.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
wifi_cfg	pointer to where the context configuration is written

Function

void iwpriv_config_read(void *wifi_cfg)

f) Data Types and Constants

IWPRIV_CONN_STATUS Enumeration

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef enum {
  IWPRIV_CONNECTION_FAILED = -1,
  IWPRIV_CONNECTION_SUCCESSFUL,
  IWPRIV_CONNECTION_IDLE,
  IWPRIV_CONNECTION_IN_PROGRESS,
  IWPRIV_CONNECTION_REESTABLISHED
} IWPRIV_CONN_STATUS;
```

Description

This is type IWPRIV_CONN_STATUS.

IWPRIV_STATUS Enumeration

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef enum {
  IWPRIV_ERROR = -1,
  IWPRIV_READY,
  IWPRIV_IN_PROGRESS
} IWPRIV_STATUS;
```

Description

This is type IWPRIV_STATUS.

IWPRIV_CMD Enumeration

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef enum {
  PRESCAN_OPTION_GET,
  PRESCAN_OPTION_SET,
  PRESCAN_START,
  PRESCAN_ISFINISHED_GET,
  SCAN_START,
  SCANSTATUS_GET,
  SCANRESULT_GET,
  SCANRESULTS_COUNT_GET,
  CONFIG_GET,
  CONFIG_SET,
  SSID_GET,
  SSID_SET,
  NETWORKTYPE_GET,
  NETWORKTYPE_SET,
  CONNSTATUS_GET,
  CLIENTINFO_GET,
  DEVICEINFO_GET,
  DRVSTATUS_GET,
  FWUPGRADEREQUEST_GET,
  OPERATIONMODE_GET,
  INITCONN_OPTION_SET,
  ADHOCCTX_SET,
  MULTICASTFILTER_SET,
  POWERSAVE_SET
} IWPRIV_CMD;
```

Description

This is type IWPRIV_CMD.

IWPRIV_EXECUTE_PARAM Union

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef union {
} IWPRIV_EXECUTE_PARAM;
```

Description

This is type IWPRIV_EXECUTE_PARAM.

IWPRIV_GET_PARAM Union

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef union {
  IWPRIV_PARAM_SCAN scan;
  IWPRIV_PARAM_CONFIG cfg;
  IWPRIV_PARAM_SSID ssid;
  IWPRIV_PARAM_NETWORKTYPE netType;
  IWPRIV_PARAM_CONNECT conn;
  IWPRIV_PARAM_CLIENTINFO clientInfo;
  IWPRIV_PARAM_DEVICEINFO devInfo;
```

```
  IWPRIV_PARAM_DRIVERSTATUS driverStatus;
  IWPRIV_PARAM_FWUPGRADE fwUpgrade;
  IWPRIV_PARAM_OPERATIONMODE opMode;
} IWPRIV_GET_PARAM;
```

Description

This is type IWPRIV_GET_PARAM.

IWPRIV_PARAM_CLIENTINFO Structure

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef struct {
  uint8_t * addr;
  bool updated;
} IWPRIV_PARAM_CLIENTINFO;
```

Members

Members	Description
uint8_t * addr;	it usually points to a MAC address, which is an array of 6 uint8_t elements

Description

This is type IWPRIV_PARAM_CLIENTINFO.

IWPRIV_PARAM_CONTEXT Structure

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef struct {
  void * context;
} IWPRIV_PARAM_CONTEXT;
```

Description

This is type IWPRIV_PARAM_CONTEXT.

IWPRIV_PARAM_DEVICEINFO Structure

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef struct {
  void * info;
} IWPRIV_PARAM_DEVICEINFO;
```

Description

This is type IWPRIV_PARAM_DEVICEINFO.

IWPRIV_SCAN_STATUS Enumeration

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef enum {
  IWPRIV_SCAN_SUCCESSFUL,
  IWPRIV_SCAN_IDLE,
  IWPRIV_SCAN_IN_PROGRESS,
```

```
    IWPRIV_SCAN_NO_AP_FOUND
} IWPRIV_SCAN_STATUS;
```

Description

This is type IWPRIV_SCAN_STATUS.

IWPRIV_SET_PARAM Union

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef union {
  IWPRIV_PARAM_SCAN scan;
  IWPRIV_PARAM_CONFIG cfg;
  IWPRIV_PARAM_SSID ssid;
  IWPRIV_PARAM_NETWORKTYPE netType;
  IWPRIV_PARAM_CONNECT conn;
  IWPRIV_PARAM_CONTEXT ctx;
  IWPRIV_PARAM_MULTICASTFILTER multicast;
  IWPRIV_PARAM_POWERSAVE powerSave;
} IWPRIV_SET_PARAM;
```

Description

This is type IWPRIV_SET_PARAM.

IWPRIV_PARAM_CONFIG Structure

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef struct {
  void * config;
} IWPRIV_PARAM_CONFIG;
```

Description

This is type IWPRIV_PARAM_CONFIG.

IWPRIV_PARAM_CONNECT Structure

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef struct {
  bool initConnAllowed;
  IWPRIV_CONN_STATUS status;
} IWPRIV_PARAM_CONNECT;
```

Description

This is type IWPRIV_PARAM_CONNECT.

IWPRIV_PARAM_DRIVERSTATUS Structure

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef struct {
  bool isOpen;
} IWPRIV_PARAM_DRIVERSTATUS;
```

Description

This is type IWPRIV_PARAM_DRIVERSTATUS.

IWPRIV_PARAM_FWUPGRADE Structure

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef struct {
  bool requested;
} IWPRIV_PARAM_FWUPGRADE;
```

Description

This is type IWPRIV_PARAM_FWUPGRADE.

IWPRIV_PARAM_MULTICASTFILTER Structure

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef struct {
  uint8_t * addr;
  IWPRIV_STATUS status;
} IWPRIV_PARAM_MULTICASTFILTER;
```

Members

Members	Description
uint8_t * addr;	it usually points to a MAC address, which is an array of 6 uint8_t elements

Description

This is type IWPRIV_PARAM_MULTICASTFILTER.

IWPRIV_PARAM_NETWORKTYPE Structure

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef struct {
  uint8_t type;
} IWPRIV_PARAM_NETWORKTYPE;
```

Description

This is type IWPRIV_PARAM_NETWORKTYPE.

IWPRIV_PARAM_OPERATIONMODE Structure

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef struct {
  bool isServer;
} IWPRIV_PARAM_OPERATIONMODE;
```

Description

This is type IWPRIV_PARAM_OPERATIONMODE.

IWPRIV_PARAM_POWERSAVE Structure

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef struct {
  bool enabled;
} IWPRIV_PARAM_POWERSAVE;
```

Description

This is type IWPRIV_PARAM_POWERSAVE.

IWPRIV_PARAM_SCAN Structure

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef struct {
  bool prescanAllowed;
  bool prescanFinished;
  IWPRIV_SCAN_STATUS scanStatus;
  uint16_t numberOfResults;
  uint16_t index;
  WDRV_SCAN_RESULT * result;
} IWPRIV_PARAM_SCAN;
```

Description

This is type IWPRIV_PARAM_SCAN.

IWPRIV_PARAM_SSID Structure

File

wdrv_mrf24wn_iwpriv.h

C

```
typedef struct {
  uint8_t * ssid;
  uint8_t ssidLen;
} IWPRIV_PARAM_SSID;
```

Description

This is type IWPRIV_PARAM_SSID.

Files

Files

Name	Description
wdrv_mrf24wn_api.h	MRF24WN Interface Functions
wdrv_mrf24wn_iwpriv.h	Configure optional (private) parameters of MRF24WN driver.

Description

This section lists the source and header files used by the MRF24WN Wi-Fi Driver Library.

wdrv_mrf24wn_api.h

MRF24WN Interface Functions

Functions

	Name	Description
	WDRV_EXT_CmdConnectContextChannelGet	Gets the AP channel **Implementation:** Dynamic
	WDRV_EXT_CmdNetModeIBSSSet	Sets the Wi-Fi network type to Adhoc. **Implementation:** Dynamic
	WDRV_EXT_CmdPowerSaveGet	Retrieves current power save status. **Implementation:** Dynamic
	WDRV_EXT_CmdSecWPA2Set	Sets Wi-Fi security to WPA2. **Implementation:** Dynamic
	WDRV_EXT_Initialize	Initializes the MRF24WN Wi-Fi driver. **Implementation:** Dynamic
	WDRV_EXT_PrivConfig	Configures g_wdrvext_priv parameter. **Implementation:** Dynamic
	WDRV_EXT_ScanResultGet	Reads the selected scan results back from the MRF24WN module. **Implementation:** Dynamic
	WDRV_GPIO_DeInit	Deinitializes the GPIO objects for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_GPIO_Init	Initializes the GPIO objects for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_GPIO_PowerOff	Powers off the MRF24WN module. **Implementation:** Dynamic
	WDRV_GPIO_PowerOn	Powers on the MRF24WN module. **Implementation:** Dynamic
	WDRV_INTR_Deinit	Deinitializes interrupts for Wi-Fi driver. **Implementation:** Dynamic
	WDRV_INTR_Init	Initializes interrupts for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_INTR_SourceDisable	Disables interrupts from the module. **Implementation:** Dynamic
	WDRV_INTR_SourceEnable	Enables interrupts from the module. **Implementation:** Dynamic
	WDRV_IsPowerOff	Checks if MRF24WN is turned off. **Implementation:** Dynamic
	WDRV_MRF24WN_ISR	Wi-Fi driver (MRF24WN-specific) interrupt service routine. **Implementation:** Dynamic
	WDRV_SPI_Deinit	Deinitializes the SPI object for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_SPI_In	Receives data from the module through the SPI bus. **Implementation:** Dynamic
	WDRV_SPI_Init	Initializes the SPI object for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_SPI_Out	Sends data out to the module through the SPI bus. **Implementation:** Dynamic

Description

MRF24WN Interface Functions

File Name

wdrv_mrf24wn_api.h

Company

Microchip Technology Inc.

wdrv_mrf24wn_iwpriv.h

Configure optional (private) parameters of MRF24WN driver.

Enumerations

	Name	Description
	IWPRIV_CMD	This is type IWPRIV_CMD.
	IWPRIV_CONN_STATUS	This is type IWPRIV_CONN_STATUS.
	IWPRIV_SCAN_STATUS	This is type IWPRIV_SCAN_STATUS.
	IWPRIV_STATUS	This is type IWPRIV_STATUS.

Functions

	Name	Description
	iwpriv_adhocctx_set	Sets the Ad hoc network context information. **Implementation:** Dynamic
	iwpriv_config_read	Reads the Wi-Fi context configuration. **Implementation:** Dynamic
	iwpriv_config_write	Writes to the Wi-Fi context configuration which is currently used by Wi-Fi driver. **Implementation:** Dynamic
	iwpriv_connstatus_get	Gets the Wi-Fi connection status. **Implementation:** Dynamic
	iwpriv_devinfo_get	Gets the device information. **Implementation:** Dynamic
	iwpriv_execute	This is function iwpriv_execute.
	iwpriv_get	This is function iwpriv_get.
	iwpriv_initialconn_set	Sets the initial connection status of Wi-Fi driver. **Implementation:** Dynamic
	iwpriv_initstatus_get	Gets the initialization status of Wi-Fi driver. **Implementation:** Dynamic
	iwpriv_is_servermode	Checks if the passed Wi-Fi context configuration is operating in server mode. **Implementation:** Dynamic
	iwpriv_leftclient_get	Gets the left client's information. **Implementation:** Dynamic
	iwpriv_mcastfilter_set	Adds a MAC address to the multi-cast filter. **Implementation:** Dynamic
	iwpriv_nettype_get	Gets the current network type. **Implementation:** Dynamic
	iwpriv_nettype_set	Sets the current network type. **Implementation:** Dynamic
	iwpriv_numberofscanresults_get	Gets the number of scan results. **Implementation:** Dynamic
	iwpriv_powersave_config	Enables or disables Power Save mode in Wi-Fi driver. **Implementation:** Dynamic
	iwpriv_prescan_isfinished	Checks if the prescan is complete. **Implementation:** Dynamic
	iwpriv_prescan_option_get	To see if prescan will run before next connection. **Implementation:** Dynamic
	iwpriv_prescan_option_set	To run prescan or not. **Implementation:** Dynamic
	iwpriv_prescan_start	Starts prescan. **Implementation:** Dynamic
	iwpriv_scan_start	Starts scan. **Implementation:** Dynamic
	iwpriv_scanstatus_get	Gets the prescan status. **Implementation:** Dynamic
	iwpriv_set	This is function iwpriv_set.
	iwpriv_ssid_get	Gets the current SSID. **Implementation:** Dynamic
	iwpriv_ssid_set	Sets the current SSID. **Implementation:** Dynamic

Structures

	Name	Description
	IWPRIV_PARAM_CLIENTINFO	This is type IWPRIV_PARAM_CLIENTINFO.
	IWPRIV_PARAM_CONFIG	This is type IWPRIV_PARAM_CONFIG.
	IWPRIV_PARAM_CONNECT	This is type IWPRIV_PARAM_CONNECT.
	IWPRIV_PARAM_CONTEXT	This is type IWPRIV_PARAM_CONTEXT.
	IWPRIV_PARAM_DEVICEINFO	This is type IWPRIV_PARAM_DEVICEINFO.
	IWPRIV_PARAM_DRIVERSTATUS	This is type IWPRIV_PARAM_DRIVERSTATUS.
	IWPRIV_PARAM_FWUPGRADE	This is type IWPRIV_PARAM_FWUPGRADE.
	IWPRIV_PARAM_MULTICASTFILTER	This is type IWPRIV_PARAM_MULTICASTFILTER.
	IWPRIV_PARAM_NETWORKTYPE	This is type IWPRIV_PARAM_NETWORKTYPE.
	IWPRIV_PARAM_OPERATIONMODE	This is type IWPRIV_PARAM_OPERATIONMODE.
	IWPRIV_PARAM_POWERSAVE	This is type IWPRIV_PARAM_POWERSAVE.
	IWPRIV_PARAM_SCAN	This is type IWPRIV_PARAM_SCAN.
	IWPRIV_PARAM_SSID	This is type IWPRIV_PARAM_SSID.

Unions

	Name	Description
	IWPRIV_EXECUTE_PARAM	This is type IWPRIV_EXECUTE_PARAM.
	IWPRIV_GET_PARAM	This is type IWPRIV_GET_PARAM.
	IWPRIV_SET_PARAM	This is type IWPRIV_SET_PARAM.

Description

MRF24WN Private Configuration Support

Functions in this module support the connection process for the MRF24WN.

File Name

wdrv_mrf24wn_iwpriv.h

Company

Microchip Technology Inc.

WILC1000 Wi-Fi Driver Ethernet Mode Library

This topic describes the WILC1000 Wi-Fi Driver Library.

Introduction

This library provides a low-level abstraction of the WILC1000 Wi-Fi Driver Library that is available on the Microchip family of microcontrollers with a convenient C language interface. It can be used to simplify low-level access to the module without the necessity of interacting directly with the module's registers, there by hiding differences from one microcontroller variant to another.

Note: The WILC1000 Wi-Fi Driver is compatible with the WILC1000 PICtail/PICtail Plus Daughter Board with WILC1000 firmware version 4.2.3 and later in "Ethernet mode".

Description

The Wi-Fi software library, in conjunction with the WILC1000 module, allows an application to:

- Join an existing 802.11 Wi-Fi network
- Create a 802.11 Wi-Fi network

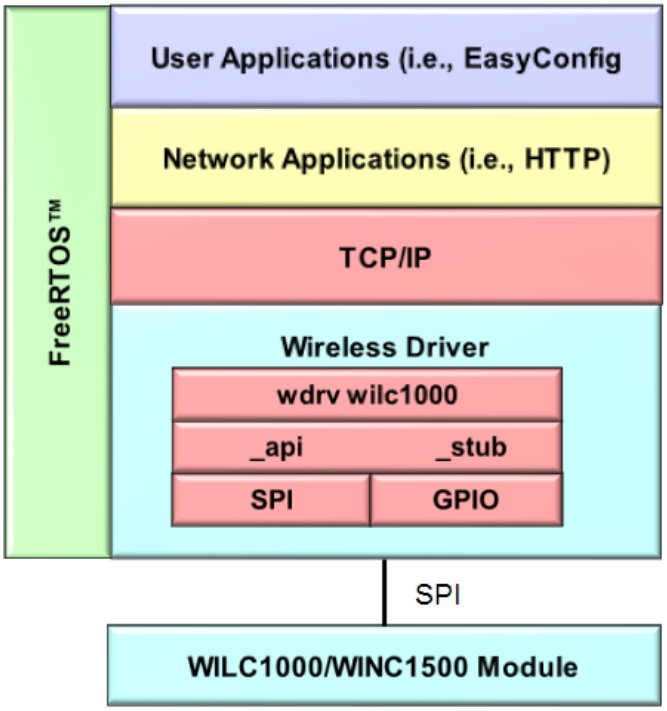
The following application services are provided by the Wi-Fi library:

- Configure a Wi-Fi connection (SSID, security mode, and so on)
- Join an existing network or create a "Soft-AP" Wi-Fi network
- Scan for other Wi-Fi devices in the area
- Receive Wi-Fi network status
- Wi-Fi power control

The MAC_layer services are not directly accessible to the application. This portion of the code resides under the TCP/IP Stack MAC module software layers and is used by stack services to transmit and receive data over a Wi-Fi network. The following diagram shows the interaction of the

primary software blocks in a Wi-Fi application.

Wi-Fi Software Block Diagram



The following table provides information that includes network mode and security mode support by the WILC1000 Wi-Fi Driver.

WILC1000 Network Connection Matrix	Network Mode	
Security Mode	Infrastructure	SoftAP
Open	YES	YES
WEP40 (64-bit)	YES	YES
WEP104 (128-bit)	YES	YES
WPA-AUTO PSK (TKIP/AES)	YES	NA
WPA-AUTO PSK (AES)	YES	YES
WPS Push Button	YES	NA
WPS PIN	YES	NA

Using the Library

This topic describes the basic architecture of the WILC1000 Wi-Fi Driver Library and provides information and examples on its use.

Description

Interface Header Files: `wdrv_wilc1000_api.h` and `wdrv_wilc1000_stub.h`

The interface to the WILC1000 Wi-Fi Driver Library is defined in the `wdrv_wilc1000_api.h` and `wdrv_wilc1000_stub.h` header files.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the WILC1000 Wi-Fi module with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The WILC1000 Wi-Fi Library provides the following functionality:

- Wi-Fi library initialization
- Wi-Fi network configuration
- Wi-Fi network connection
- Scanning for existing Wi-Fi networks

- Wi-Fi event processing
- Wi-Fi status
- Wi-FI console commands

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

TheLibrary Interface functions are divided into various sub-sections, which address one of the blocks or the overall operation of the Wi-Fi module.

Library Interface Section	Description
Wi-Fi Initialization Functions	This section provides functions that initialize the Wi-Fi library and allow its API to be used.
Wi-Fi Status Functions	This section provides functions that retrieve the Wi-Fi connection status.
Wi-Fi External Functions	This section provides public functions accessible to TCP/IP applications.
Other Functions	This section provides additional miscellaneous functions for configuring the Wi-Fi connection.
Data Types and Constants	This section provides data types and macros.

How the Library Works

This section describes how the WILC1000 Wi-Fi Driver Library operates.

Description

Before the driver is ready for use, it should be configured (compile time configuration).

There are a few run-time configuration items that are done during initialization of the driver instance, and a few that are client-specific and are done using dedicated functions.

To use the WILC1000 Wi-Fi Driver, initialization and client functions should be invoked in a specific sequence to ensure correct operation.

Configuring the Library

This section describes how to configure the WILC1000 Wi-Fi driver.

Description

The configuration of the WILC1000 Wi-Fi Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the Wi-Fi Driver. Based on the selections made, the WILC1000 Wi-Fi Driver may support the selected features. These configuration settings will apply to all instances of the WILC1000 Wi-Fi Driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Sample Functionality

The following code provides an example of Wi-Fi Driver configuration. (refer to `system.config.h`)

```
/*** SPI Driver Configuration ***/
#define DRV_SPI_NUMBER_OF_MODULES    4
/*** Driver Compilation and static configuration options. ***/
/*** Select SPI compilation units.***/
#define DRV_SPI_POLLED    0
#define DRV_SPI_ISR    1
#define DRV_SPI_MASTER    1
#define DRV_SPI_SLAVE    0
#define DRV_SPI_RM    1
#define DRV_SPI_EBM    0
#define DRV_SPI_8BIT    1
#define DRV_SPI_16BIT    0
#define DRV_SPI_32BIT    0
#define DRV_SPI_DMA    1
/*** SPI Driver Static Allocation Options ***/
#define DRV_SPI_INSTANCES_NUMBER    1
#define DRV_SPI_CLIENTS_NUMBER    1
#define DRV_SPI_ELEMENTS_PER_QUEUE    10
/*** SPI Driver DMA Options ***/
```

```
#define DRV_SPI_DMA_TXFER_SIZE    512
#define DRV_SPI_DMA_DUMMY_BUFFER_SIZE    512

/* SPI Driver Instance 0 Configuration */
#define DRV_SPI_SPI_ID_IDX0    SPI_ID_1
#define DRV_SPI_TASK_MODE_IDX0    DRV_SPI_TASK_MODE_ISR
#define DRV_SPI_SPI_MODE_IDX0    DRV_SPI_MODE_MASTER
#define DRV_SPI_ALLOW_IDLE_RUN_IDX0    false
#define DRV_SPI_SPI_PROTOCOL_TYPE_IDX0    DRV_SPI_PROTOCOL_TYPE_STANDARD
#define DRV_SPI_COMM_WIDTH_IDX0    SPI_COMMUNICATION_WIDTH_8BITS
#define DRV_SPI_SPI_CLOCK_IDX0    CLK_BUS_PERIPHERAL_2
#define DRV_SPI_BAUD_RATE_IDX0    2000000
#define DRV_SPI_BUFFER_TYPE_IDX0    DRV_SPI_BUFFER_TYPE_STANDARD
#define DRV_SPI_CLOCK_MODE_IDX0    DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL
#define DRV_SPI_INPUT_PHASE_IDX0    SPI_INPUT_SAMPLING_PHASE_AT_END


#define DRV_SPI_TRANSMIT_DUMMY_BYTE_VALUE_IDX0    0x00

#define DRV_SPI_TX_INT_SOURCE_IDX0    INT_SOURCE_SPI_1_TRANSMIT
#define DRV_SPI_RX_INT_SOURCE_IDX0    INT_SOURCE_SPI_1_RECEIVE
#define DRV_SPI_ERROR_INT_SOURCE_IDX0    INT_SOURCE_SPI_1_ERROR
#define DRV_SPI_INT_VECTOR_IDX0    INT_VECTOR_SPI1
#define DRV_SPI_INT_PRIORITY_IDX0    INT_PRIORITY_LEVEL1
#define DRV_SPI_INT_SUB_PRIORITY_IDX0    INT_SUBPRIORITY_LEVEL0
#define DRV_SPI_QUEUE_SIZE_IDX0    10
#define DRV_SPI_RESERVED_JOB_IDX0    1
#define DRV_SPI_TX_DMA_CHANNEL_IDX0    DMA_CHANNEL_1
#define DRV_SPI_TX_DMA_THRESHOLD_IDX0    16
#define DRV_SPI_RX_DMA_CHANNEL_IDX0    DMA_CHANNEL_0
#define DRV_SPI_RX_DMA_THRESHOLD_IDX0    16
/*** Timer Driver Configuration ***/
#define DRV_TMR_INTERRUPT_MODE    true
#define DRV_TMR_INSTANCES_NUMBER    1
#define DRV_TMR_CLIENTS_NUMBER    1

/*** Timer Driver 0 Configuration ***/
#define DRV_TMR_PERIPHERAL_ID_IDX0    TMR_ID_2
#define DRV_TMR_INTERRUPT_SOURCE_IDX0    INT_SOURCE_TIMER_2
#define DRV_TMR_INTERRUPT_VECTOR_IDX0    INT_VECTOR_T2
#define DRV_TMR_ISR_VECTOR_IDX0    _TIMER_2_VECTOR
#define DRV_TMR_INTERRUPT_PRIORITY_IDX0    INT_PRIORITY_LEVEL4
#define DRV_TMR_INTERRUPT_SUB_PRIORITY_IDX0 INT_SUBPRIORITY_LEVEL0
#define DRV_TMR_CLOCK_SOURCE_IDX0    DRV_TMR_CLKSOURCE_INTERNAL
#define DRV_TMR_PRESCALE_IDX0    TMR_PRESCALE_VALUE_256
#define DRV_TMR_OPERATION_MODE_IDX0    DRV_TMR_OPERATION_MODE_16_BIT
#define DRV_TMR_ASYNC_WRITE_ENABLE_IDX0    false
#define DRV_TMR_POWER_STATE_IDX0    SYS_MODULE_POWER_RUN_FULL

/*** Wi-Fi Driver Configuration ***/
#define WILC1000_INT_SOURCE INT_SOURCE_CHANGE_NOTICE
#define WILC1000_INT_VECTOR INT_VECTOR_CN

#define WDRV_SPI_INDEX 0
#define WDRV_SPI_INSTANCE sysObj.spiObjectIdx0

#define WDRV_USE_SPI_DMA

#define WDRV_NVM_SPACE_ENABLE
#define WDRV_NVM_SPACE_ADDR (48 * 1024)

#define WDRV_BOARD_TYPE WDRV_BD_TYPE_MX_ESK

#define WDRV_EXT_RTOS_TASK_SIZE 2048u
#define WDRV_EXT_RTOS_TASK_PRIORITY 2u

// I/O mappings for general control pins, including CHIP_EN, IRQN, RESET_N and SPI_SSN.
#define WDRV_CHIP_EN_PORT_CHANNEL    PORT_CHANNEL_F
#define WDRV_CHIP_EN_BIT_POS    1
```

```
#define WDRV_IRQN_PORT_CHANNEL    PORT_CHANNEL_G
#define WDRV_IRQN_BIT_POS    7

#define WDRV_RESET_N_PORT_CHANNEL    PORT_CHANNEL_F
#define WDRV_RESET_N_BIT_POS    0

#define WDRV_SPI_SSN_PORT_CHANNEL    PORT_CHANNEL_B
#define WDRV_SPI_SSN_BIT_POS    2

#define WILC1000_ON_PIC32MX_ESK

// On PIC32MX ESK, when CN9 (Pin G7) is used as external interrupt,
// it is sometimes better to use another GPIO (Pin E0) to read CN9's value.


// In this case, a jumper wire is needed to connect Pin E0 and Pin G7.
//#define WDRV_VERIFY_IRQN_BY_ANOTHER_GPIO
#if defined(WDRV_VERIFY_IRQN_BY_ANOTHER_GPIO)
// Use Pin E0. Please also make sure that Pin E0 and Pin G7 are connected (by a jumper wire).
#define WDRV_IRQN_PORT_CHANNEL_READ    PORT_CHANNEL_E
#define WDRV_IRQN_BIT_POS_READ    0
#else
// Still directly read Pin G7's value.
#define WDRV_IRQN_PORT_CHANNEL_READ    PORT_CHANNEL_G
#define WDRV_IRQN_BIT_POS_READ    7
#endif

#define WDRV_DEFAULT_NETWORK_TYPE WDRV_NETWORK_TYPE_INFRASTRUCTURE
#define WDRV_DEFAULT_CHANNEL 6
#define WDRV_DEFAULT_SSID "MicrochipDemoApp"

#define WDRV_DEFAULT_SECURITY_MODE WDRV_SECURITY_OPEN
#define WDRV_DEFAULT_WEP_KEYS_40 "5AFB6C8E77" // default WEP40 key
#define WDRV_DEFAULT_WEP_KEYS_104 "90E96780C739409DA50034FCAA" // default WEP104 key
#define WDRV_DEFAULT_PSK_PHRASE "Microchip 802.11 Secret PSK Password" // default WPA-PSK
```

Building the Library

This section lists the files that are available in the WILC1000 Wi-Fi Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/wifi/wilc1000`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
wdrv_wilc1000_stub.h	Contains Stub function prototypes for interfacing to the WILC1000 Wi-Fi Driver.
wdrv_wilc1000_api.h	Contains API function prototypes for interfacing to the WILC1000 Wi-Fi Driver.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
wdrv_wilc1000_console.c	Console module for the WILC1000 wireless driver.
wdrv_wilc1000_fw_update.c	WILC1000 firmware update support.
wdrv_wilc1000_eint.c	External interrupt handler for the WILC1000 wireless driver.
wdrv_wilc1000_timer.c	Timer functions for the WILC1000 wireless driver.

wdrv_wilc1000_gpio.c	WILC1000 GPIO support for SPI communication.
wdrv_wilc1000_spi.c	WILC1000 support for SPI communication.
wdrv_wilc1000_cli.c	WILC1000 driver CLI implementation.
wdrv_wilc1000_config_data.c	Stores and retrieves Wi-Fi configuration to/from non-volatile memory (NVM).
wdrv_wilc1000_connmgr.c	WILC1000 driver connection manager.
wdrv_wilc1000_events.c	WILC1000 driver MAC events.
wdrv_wilc1000_iwpriv.c	WILC1000 driver connection process functions.
wdrv_wilc1000_main.c	WILC1000 driver Microchip TCP/IP Stack PIC32 MAC support.
wdrv_wilc1000_osal.c	WILC1000 driver OS abstraction layer.
wdrv_wilc1000_scan_helper.c	WILC1000 driver scan helper functions.
wdrext_wilc1000.c	WILC1000 driver extended functions.
wilc1000_task.c	WILC1000 driver task handler.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	The WILC1000 Wi-Fi Driver controller has no optional files.

Module Dependencies

The WILC1000 Wi-Fi Driver Library depends on the following modules:

- SPI Driver Library
- NVM Driver Library
- UART Driver Library
- USB Driver Library
- Operating System Abstraction Layer (OSAL) Library Help
- Clock System Service Library
- System Service Library Introduction
- Console System Service Library
- File System Service Library
- Interrupt System Service Library
- Timer System Service Library
- Debug System Service Library
- Ports System Service Library
- FreeRTOS Library Help
- Crypto Library
- Peripheral Libraries
- Networking Presentation Layer Help
- TCP/IP Stack Library Help
- Command Processor System Service Library
- DMA System Service Library
- Random Number Generator System Service Library
- Common System Service Library
- TCP/IP Ethernet MAC Driver Library

Console Commands

This section describes the console commands available for the WILC1000 Wi-Fi Driver.

Description

Both the Web Server and the EasyConfig demonstrations support the followings commands, which enable control over the Wi-Fi settings.

Command: deleteconf

Parameters	Description
None.	Wi-Fi console command to erase saved Wi-Fi configuration in memory.

Command: iwconfig

Parameters	Description
[ssid <name>]	name: Specifies the name of the SSID (1-32 ASCII characters).
[mode <idle | managed>]	idle: Disconnected from the current configuration. managed: Connects in infrastructure mode to the currently set SSID.
[power <enable | disable>]	enable: Enables all Power-Saving features (PS_POLL). Will wake up to check for all types of traffic (unicast, multicast, and broadcast). disable: Disables any Power-Saving features. Will always be in an active power state.

[security <mode>]	mode: open/wep40/wep104/wpa/wpa2/pin/pbc. For example:
	`iwconfig security open`
	`iwconfig security wep40 <key>`
	`iwconfig security wep104 <key>`
	`iwconfig security wpa <key>`
	`iwconfig security wpa2 <key>`
	`iwconfig security pin <pin>`
	`iwconfig security pbc`

[scan]	Starts a Wi-Fi scan.
[scanget <scan_index>]	scan_index: Retrieves the scan result after the scan completes (1 - n).

Command: mac

Parameters	Description
None.	Wi-Fi console command to retrieve the MAC address of the MRF24WN module.

Command: ota

Parameters	Description
[http://ip-address/] [filename.bin]	Upgrade the WILC1000 firmware over-the-air. For example: http://192.168.0.4/winc1500_ota.bin

Command: readconf

Parameters	Description
None.	Wi-Fi console command to read saved Wi-Fi configuration in memory.

Command: saveconf

Parameters	Description
None.	Wi-Fi console command to save Wi-Fi configuration to memory.

Library Interface

This section describes the Application Programming Interface (API) functions of the WILC1000 Wi-Fi Driver.

Refer to each section for a detailed description.

a) Wi-Fi Initialization Functions

b) Wi-Fi Status Functions

c) Wi-Fi External Functions

d) Other Functions

e) Data Types and Constants

Files

Files

Name	Description
wdrv_wilc1000_api.h	WILC1000 wireless driver APIs.
wdrv_wilc1000_stub.h	WILC1000 wireless driver stub APIs.

Description

This section lists the source and header files used by the MRF24WN Wi-Fi Driver Library.

wdrv_wilc1000_api.h

WILC1000 wireless driver APIs.

Functions

	Name	Description
	WDRV_EXT_CmdScanOptionSet	Sets scan options. **Implementation:** Dynamic
	WDRV_EXT_CmdSSIDSet	Sets the SSID. **Implementation:** Dynamic
	WDRV_EXT_Initialize	Initializes the WILC1000 Wi-Fi driver. **Implementation:** Dynamic
	WDRV_EXT_ScanDoneSet	Indicates when a scan has completed. **Implementation:** Dynamic
	WDRV_EXT_ScanIsInProgress	Check whether host scan is now in progress or not. **Implementation:** Dynamic

Description

WILC1000 wireless driver APIs.

File Name

wdrv_wilc1000_api.h

Company

Microchip Technology Inc.

wdrv_wilc1000_stub.h

WILC1000 wireless driver stub APIs.

Description

WILC1000 wireless driver stub APIs.

File Name

wdrv_wilc1000_stub.h

Company

Microchip Technology Inc.

WINC1500 Wi-Fi Driver Ethernet Mode Library

This topic describes the WINC1500 Wi-Fi Driver Library.

Introduction

This library provides a low-level abstraction of the WINC1500 Wi-Fi Driver Library that is available on the Microchip family of microcontrollers with a convenient C language interface. It can be used to simplify low-level access to the module without the necessity of interacting directly with the module's registers, there by hiding differences from one microcontroller variant to another.

Note: The WINC1500 Wi-Fi Driver is compatible with the WINC1500 PICtail/PICtail Plus Daughter board with WINC1500 firmware version 19.5.2 and later in "Ethernet mode". The driver will also work with WINC1500 firmware version 19.4.4 with limited backward compatibility.

Description

The Wi-Fi software library, in conjunction with the WINC1500 module, allows an application to:

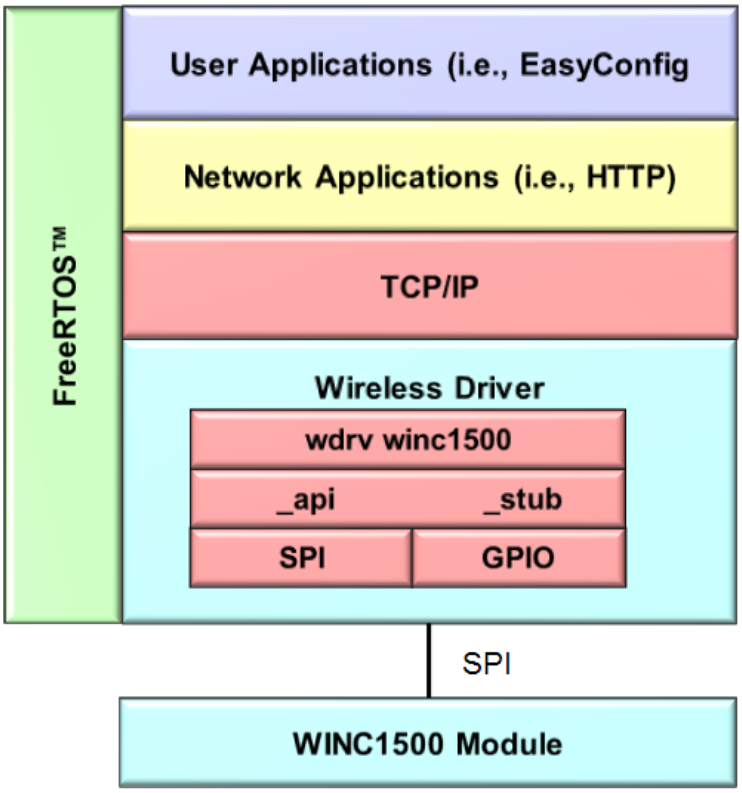
- Join an existing 802.11 Wi-Fi network
- Create a 802.11 Wi-Fi network

The following application services are provided by the Wi-Fi library:

- Configure a Wi-Fi connection (SSID, security mode, and so on)
- Join an existing network or create a "Soft-AP" Wi-Fi network
- Scan for other Wi-Fi devices in the area
- Receive Wi-Fi network status
- Wi-Fi power control

The MAC_layer services are not directly accessible to the application. This portion of the code resides under the TCP/IP Stack MAC module software layers and is used by stack services to transmit and receive data over a Wi-Fi network. The following diagram shows the interaction of the primary software blocks in a Wi-Fi application.

Wi-Fi Software Block Diagram



The following table provides information that includes network mode and security mode support by the WINC1500 Wi-Fi Driver.

WINC1500 Network Connection Matrix	Network Mode	
Security Mode	Infrastructure	SoftAP
Open	YES	YES
WEP40 (64-bit)	YES	YES
WEP104 (128-bit)	YES	YES
WPA-AUTO PSK (TKIP/AES)	YES	NA
WPA-AUTO PSK (AES)	YES	YES
WPS Push Button	YES	NA
WPS PIN	YES	NA

Using the Library

This topic describes the basic architecture of the WINC1500 Wi-Fi Driver Library and provides information and examples on its use.

Description

Interface Header Files: `wdrv_winc1500_api.h` and `wdrv_winc1500_stub.h`

The interface to the WINC1500 Wi-Fi Driver Library is defined in the `wdrv_winc1500_api.h` and `wdrv_winc1500_stub.h` header files.

Please refer to the Understanding MPLAB Harmony section for how the driver interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the WINC1500 Wi-Fi module with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The WINC1500 Wi-Fi Library provides the following functionality:

- Wi-Fi library initialization
- Wi-Fi network configuration
- Wi-Fi network connection
- Scanning for existing Wi-Fi networks
- Wi-Fi event processing
- Wi-Fi status
- Wi-FI console commands

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

TheLibrary Interface functions are divided into various sub-sections, which address one of the blocks or the overall operation of the Wi-Fi module.

Library Interface Section	Description
Wi-Fi Initialization Functions	This section provides functions that initialize the Wi-Fi library and allow its API to be used.
Wi-Fi Status Functions	This section provides functions that retrieve the Wi-Fi connection status.
Wi-Fi External Functions	This section provides public functions accessible to TCP/IP applications.
Other Functions	This section provides additional miscellaneous functions for configuring the Wi-Fi connection.
Data Types and Constants	This section provides data types and macros.

How the Library Works

This section describes how the WINC1500 Wi-Fi Driver Library operates.

Description

Before the driver is ready for use, it should be configured (compile time configuration).

There are a few run-time configuration items that are done during initialization of the driver instance, and a few that are client-specific and are done using dedicated functions.

To use the WINC1500 Wi-Fi Driver, initialization and client functions should be invoked in a specific sequence to ensure correct operation.

Configuring the Library

This section describes how to configure the WINC1500 Wi-Fi driver.

Description

The configuration of the WINC1500 Wi-Fi Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the Wi-Fi Driver. Based on the selections made, the WINC1500 Wi-Fi Driver may support the selected features. These configuration settings will apply to all instances of the WINC1500 Wi-Fi Driver.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Sample Functionality

The following code provides an example of Wi-Fi Driver configuration. (refer to `system.config.h`)

```
/*** SPI Driver Configuration ***/
#define DRV_SPI_NUMBER_OF_MODULES        4

/*** Driver Compilation and static configuration options. ***/
/*** Select SPI compilation units.***/
#define DRV_SPI_POLLED           0
#define DRV_SPI_ISR              1
#define DRV_SPI_MASTER           1
#define DRV_SPI_SLAVE            0
#define DRV_SPI_RM               1
#define DRV_SPI_EBM              0
#define DRV_SPI_8BIT             1
#define DRV_SPI_16BIT            0
#define DRV_SPI_32BIT            0
#define DRV_SPI_DMA              1

/*** SPI Driver Static Allocation Options ***/
#define DRV_SPI_INSTANCES_NUMBER         1
#define DRV_SPI_CLIENTS_NUMBER           1
#define DRV_SPI_ELEMENTS_PER_QUEUE       10

/*** SPI Driver DMA Options ***/
#define DRV_SPI_DMA_TXFER_SIZE            512
#define DRV_SPI_DMA_DUMMY_BUFFER_SIZE     512

/* SPI Driver Instance 0 Configuration */
#define DRV_SPI_SPI_ID_IDX0                SPI_ID_1
#define DRV_SPI_TASK_MODE_IDX0             DRV_SPI_TASK_MODE_ISR
#define DRV_SPI_SPI_MODE_IDX0              DRV_SPI_MODE_MASTER
#define DRV_SPI_ALLOW_IDLE_RUN_IDX0        false
#define DRV_SPI_SPI_PROTOCOL_TYPE_IDX0     DRV_SPI_PROTOCOL_TYPE_STANDARD
#define DRV_SPI_COMM_WIDTH_IDX0            SPI_COMMUNICATION_WIDTH_8BITS
#define DRV_SPI_SPI_CLOCK_IDX0             CLK_BUS_PERIPHERAL_2
#define DRV_SPI_BAUD_RATE_IDX0             2000000
#define DRV_SPI_BUFFER_TYPE_IDX0           DRV_SPI_BUFFER_TYPE_STANDARD
#define DRV_SPI_CLOCK_MODE_IDX0            DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL
#define DRV_SPI_INPUT_PHASE_IDX0           SPI_INPUT_SAMPLING_PHASE_AT_END
#define DRV_SPI_TRANSMIT_DUMMY_BYTE_VALUE_IDX0      0x00

#define DRV_SPI_TX_INT_SOURCE_IDX0         INT_SOURCE_SPI_1_TRANSMIT
#define DRV_SPI_RX_INT_SOURCE_IDX0         INT_SOURCE_SPI_1_RECEIVE
#define DRV_SPI_ERROR_INT_SOURCE_IDX0      INT_SOURCE_SPI_1_ERROR
#define DRV_SPI_INT_VECTOR_IDX0            INT_VECTOR_SPI1
#define DRV_SPI_INT_PRIORITY_IDX0          INT_PRIORITY_LEVEL1
#define DRV_SPI_INT_SUB_PRIORITY_IDX0      INT_SUBPRIORITY_LEVEL0
#define DRV_SPI_QUEUE_SIZE_IDX0            10
#define DRV_SPI_RESERVED_JOB_IDX0          1
#define DRV_SPI_TX_DMA_CHANNEL_IDX0        DMA_CHANNEL_1
#define DRV_SPI_TX_DMA_THRESHOLD_IDX0      16
#define DRV_SPI_RX_DMA_CHANNEL_IDX0        DMA_CHANNEL_0
#define DRV_SPI_RX_DMA_THRESHOLD_IDX0      16
```

```
/*** Timer Driver Configuration ***/
#define DRV_TMR_INTERRUPT_MODE            true
#define DRV_TMR_INSTANCES_NUMBER          1
#define DRV_TMR_CLIENTS_NUMBER            1

/*** Timer Driver 0 Configuration ***/
#define DRV_TMR_PERIPHERAL_ID_IDX0          TMR_ID_2
#define DRV_TMR_INTERRUPT_SOURCE_IDX0       INT_SOURCE_TIMER_2
#define DRV_TMR_INTERRUPT_VECTOR_IDX0       INT_VECTOR_T2
#define DRV_TMR_ISR_VECTOR_IDX0             _TIMER_2_VECTOR
#define DRV_TMR_INTERRUPT_PRIORITY_IDX0     INT_PRIORITY_LEVEL4
#define DRV_TMR_INTERRUPT_SUB_PRIORITY_IDX0 INT_SUBPRIORITY_LEVEL0
#define DRV_TMR_CLOCK_SOURCE_IDX0           DRV_TMR_CLKSOURCE_INTERNAL
#define DRV_TMR_PRESCALE_IDX0               TMR_PRESCALE_VALUE_256
#define DRV_TMR_OPERATION_MODE_IDX0         DRV_TMR_OPERATION_MODE_16_BIT
#define DRV_TMR_ASYNC_WRITE_ENABLE_IDX0     false
#define DRV_TMR_POWER_STATE_IDX0            SYS_MODULE_POWER_RUN_FULL

 /*** Wi-Fi Driver Configuration ***/
#define WINC1500_INT_SOURCE INT_SOURCE_CHANGE_NOTICE
#define WINC1500_INT_VECTOR INT_VECTOR_CN

#define WDRV_SPI_INDEX 0
#define WDRV_SPI_INSTANCE sysObj.spiObjectIdx0

#define WDRV_USE_SPI_DMA

#define WDRV_NVM_SPACE_ENABLE
#define WDRV_NVM_SPACE_ADDR (48 * 1024)

#define WDRV_BOARD_TYPE WDRV_BD_TYPE_MX_ESK

#define WDRV_EXT_RTOS_TASK_SIZE 2048u
#define WDRV_EXT_RTOS_TASK_PRIORITY 2u

// I/O mappings for general control pins, including CHIP_EN, IRQN, RESET_N and SPI_SSN.
#define WDRV_CHIP_EN_PORT_CHANNEL   PORT_CHANNEL_F
#define WDRV_CHIP_EN_BIT_POS        1

#define WDRV_IRQN_PORT_CHANNEL      PORT_CHANNEL_G
#define WDRV_IRQN_BIT_POS           7

#define WDRV_RESET_N_PORT_CHANNEL   PORT_CHANNEL_F
#define WDRV_RESET_N_BIT_POS        0

#define WDRV_SPI_SSN_PORT_CHANNEL   PORT_CHANNEL_B
#define WDRV_SPI_SSN_BIT_POS        2

#define WINC1500_ON_PIC32MX_ESK

// On PIC32MX ESK, when CN9 (Pin G7) is used as external interrupt,
// it is sometimes better to use another GPIO (Pin E0) to read CN9's value.
// In this case, a jumper wire is needed to connect Pin E0 and Pin G7.
//#define WDRV_VERIFY_IRQN_BY_ANOTHER_GPIO
#if defined(WDRV_VERIFY_IRQN_BY_ANOTHER_GPIO)
// Use Pin E0. Please also make sure that Pin E0 and Pin G7 are connected (by a jumper wire).
#define WDRV_IRQN_PORT_CHANNEL_READ     PORT_CHANNEL_E
#define WDRV_IRQN_BIT_POS_READ          0
#else
// Still directly read Pin G7's value.
#define WDRV_IRQN_PORT_CHANNEL_READ     PORT_CHANNEL_G
#define WDRV_IRQN_BIT_POS_READ          7
#endif

#define WDRV_DEFAULT_NETWORK_TYPE WDRV_NETWORK_TYPE_INFRASTRUCTURE
#define WDRV_DEFAULT_CHANNEL 6
#define WDRV_DEFAULT_SSID "MicrochipDemoApp"
```

```
#define WDRV_DEFAULT_SECURITY_MODE WDRV_SECURITY_OPEN
#define WDRV_DEFAULT_WEP_KEYS_40 "5AFB6C8E77" // default WEP40 key
#define WDRV_DEFAULT_WEP_KEYS_104 "90E96780C739409DA50034FCAA" // default WEP104 key
#define WDRV_DEFAULT_PSK_PHRASE "Microchip 802.11 Secret PSK Password" // default WPA-PSK
```

Building the Library

This section lists the files that are available in the WINC1500 Wi-Fi Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/wifi/winc1500`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
wdrv_winc1500_stub.h	Contains Stub function prototypes for interfacing to the WINC1500 Wi-Fi Driver.
wdrv_winc1500_api.h	Contains API function prototypes for interfacing to the WINC1500 Wi-Fi Driver.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
wdrv_winc1500_console.c	Console module for the WINC1500 wireless driver.
wdrv_winc1500_fw_update.c	WINC1500 firmware update support.
wdrv_winc1500_eint.c	External interrupt handler for the WINC1500 wireless driver.
wdrv_winc1500_timer.c	Timer functions for the WINC1500 wireless driver.
wdrv_winc1500_gpio.c	WINC1500 GPIO support for SPI communication.
wdrv_winc1500_spi.c	WINC1500 support for SPI communication.
wdrv_winc1500_cli.c	WINC1500 driver CLI implementation.
wdrv_winc1500_config_data.c	Stores and retrieves Wi-Fi configuration to/from non-volatile memory (NVM).
wdrv_winc1500_connmgr.c	WINC1500 driver connection manager.
wdrv_winc1500_events.c	WINC1500 driver MAC events.
wdrv_winc1500_iwpriv.c	WINC1500 driver connection process functions.
wdrv_winc1500_main.c	WINC1500 driver Microchip TCP/IP Stack PIC32 MAC support.
wdrv_winc1500_osal.c	WINC1500 driver OS abstraction layer.
wdrv_winc1500_scan_helper.c	WINC1500 driver scan helper functions.
wdrext_winc1500.c	WINC1500 driver extended functions.
winc1500_task.c	WINC1500 driver task handler.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	The WINC1500 Wi-Fi Driver controller has no optional files.

Module Dependencies

The WINC1500 Wi-Fi Driver Library depends on the following modules:

- SPI Driver Library
- NVM Driver Library

- UART Driver Library
- USB Driver Library
- Operating System Abstraction Layer (OSAL) Library Help
- Clock System Service Library
- System Service Library Introduction
- Console System Service Library
- File System Service Library
- Interrupt System Service Library
- Timer System Service Library
- Debug System Service Library
- Ports System Service Library
- FreeRTOS Library Help
- Crypto Library
- Peripheral Libraries
- Networking Presentation Layer Help
- TCP/IP Stack Library Help
- Command Processor System Service Library
- DMA System Service Library
- Random Number Generator System Service Library
- Common System Service Library
- TCP/IP Ethernet MAC Driver Library

Console Commands

This section describes the console commands available for the WINC1500 Wi-Fi Driver.

Description

Both the Web Server and the EasyConfig demonstrations support the followings commands, which enable control over the Wi-Fi settings.

Command: deleteconf

Parameters	Description
None.	Wi-Fi console command to erase saved Wi-Fi configuration in memory.

Command: iwconfig

Parameters	Description
[ssid <name>]	name: Specifies the name of the SSID (1-32 ASCII characters).
[mode <idle | managed>]	idle: Disconnected from the current configuration. managed: Connects in infrastructure mode to the currently set SSID.
[power <enable | disable>]	enable: Enables all Power-Saving features (PS_POLL). Will wake up to check for all types of traffic (unicast, multicast, and broadcast). disable: Disables any Power-Saving features. Will always be in an active power state.

[security <mode>]	mode: open/wep40/wep104/wpa/wpa2/pin/pbc. For example:
	`iwconfig security open`
	`iwconfig security wep40 <key>`
	`iwconfig security wep104 <key>`
	`iwconfig security wpa <key>`
	`iwconfig security wpa2 <key>`
	`iwconfig security pin <pin>`
	`iwconfig security pbc`

[scan]	Starts a Wi-Fi scan.
[scanget <scan_index>]	scan_index: Retrieves the scan result after the scan completes (1 - n).

Command: mac

Parameters	Description
None.	Wi-Fi console command to retrieve the MAC address of the MRF24WN module.

Command: ota

Parameters	Description
[http://ip-address/] [filename.bin]	Upgrade the WINC1500 firmware over-the-air. For example: http://192.168.0.4/winc1500_ota.bin

Command: readconf

Parameters	Description
None.	Wi-Fi console command to read saved Wi-Fi configuration in memory.

Command: saveconf

Parameters	Description
None.	Wi-Fi console command to save Wi-Fi configuration to memory.

Library Interface

a) Wi-Fi Initialization Functions

	Name	Description
	WDRV_CLI_Init	Initializes the console CLI interface. **Implementation:** Dynamic
	WDRV_INTR_Deinit	Deinitializes interrupts for Wi-Fi driver. **Implementation:** Dynamic
	WDRV_INTR_Init	Initializes interrupts for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_SPI_Deinit	Deinitializes the SPI object for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_SPI_Init	Initializes the SPI object for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_GPIO_DeInit	Deinitializes the GPIO objects for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_EXT_Deinitialize	Deinitializes the WINC1500 Wi-Fi driver. **Implementation:** Dynamic
	WDRV_WINC1500_ISR	Wi-Fi driver (WINC1500-specific) interrupt service routine. **Implementation:** Dynamic

b) Wi-Fi Status Functions

	Name	Description
	WDRV_EXT_CmdFWVersionGet	Retrieves FW version information. **Implementation:** Dynamic
	WDRV_EXT_ScanResultGet	Reads the selected scan results back from the WINC1500 module. **Implementation:** Dynamic
	WDRV_EXT_CmdMacAddressGet	Retrieves the WINC1500 MAC address. **Implementation:** Dynamic
	WDRV_EXT_CmdScanGet	Reads the number of scan results from the WINC1500 module. **Implementation:** Dynamic
	WDRV_EXT_CmdSSIDGet	Gets the SSID. **Implementation:** Dynamic

c) Wi-Fi External Functions

	Name	Description
	WDRV_EXT_CmdPowerSavePut	Puts the module in power save mode. **Implementation:** Dynamic
	WDRV_EXT_HWInterruptHandler	Wi-Fi driver (WINC1500-specific) interrupt service routine. **Implementation:** Dynamic

	WDRV_EXT_CmdScanOptionSet	Sets scan options. **Implementation:** Dynamic
	WDRV_EXT_ModuleUpDown	Enables or disables WINC1500 module. **Implementation:** Dynamic
	WDRV_EXT_MulticastFilterSet	Sets a multicast address filter. **Implementation:** Dynamic
	WDRV_EXT_CmdConnect	Directs the WINC1500 to connect to a Wi-Fi network. **Implementation:** Dynamic
	WDRV_EXT_CmdDisconnect	Directs the WINC1500 to disconnect from a Wi-Fi network. **Implementation:** Dynamic
	WDRV_EXT_CmdNetModeAPSet	Sets the Wi-Fi network type to SoftAP. **Implementation:** Dynamic
	WDRV_EXT_CmdNetModeBSSSet	Sets the Wi-Fi network type to Infrastructure. **Implementation:** Dynamic
	WDRV_EXT_CmdScanStart	Directs the WINC1500 module to start a scan. **Implementation:** Dynamic
	WDRV_EXT_CmdSecNoneSet	Sets Wi-Fi security to open (no security). **Implementation:** Dynamic
	WDRV_EXT_CmdSecWEPSet	Sets Wi-Fi security to use WEP. **Implementation:** Dynamic
	WDRV_EXT_CmdSecWPASet	Sets Wi-Fi security to use WPA/WPA2. **Implementation:** Dynamic
	WDRV_EXT_DataSend	Sends data packets to WINC1500 module. **Implementation:** Dynamic
	WDRV_EXT_ScanDoneSet	Indicates when a scan has completed. **Implementation:** Dynamic
	WDRV_EXT_CmdChannelSet	Sets the channel on which to operate. **Implementation:** Dynamic
	WDRV_EXT_CmdSSIDSet	Sets the SSID. **Implementation:** Dynamic
	WDRV_EXT_CmdFWUpdate	Directs the module to start firmware download and upgrade. **Implementation:** Dynamic
	WDRV_EXT_CmdSecWpsSet	Sets Wi-Fi security to use WPS. **Implementation:** Dynamic
	WDRV_EXT_CmdTxPowerSet	Sets the Tx Power at 3 levels, high, medium and low. **Implementation:** Dynamic
	WDRV_EXT_CmdConnectContextBssidGet	Gets the BSSID **Implementation:** Dynamic
	WDRV_EXT_CmdScanOptionsSet	Sets scan options. **Implementation:** Dynamic
	WDRV_EXT_CmdSSIDSet	Sets the SSID. **Implementation:** Dynamic
	WDRV_EXT_ScanIsInProgress	Check whether host scan is now in progress or not. **Implementation:** Dynamic

d) Other Functions

	Name	Description
	WDRV_INTR_SourceDisable	Disables interrupts from the module. **Implementation:** Dynamic
	WDRV_INTR_SourceEnable	Enables interrupts from the module. **Implementation:** Dynamic
	WDRV_EXT_Initialize	Initializes the WILC1000 Wi-Fi driver. **Implementation:** Dynamic
	WDRV_EXT_RssiRead	Requests RSSI for the connected AP. **Implementation:** Dynamic
	WDRV_EXT_WPSResultsRead	Reads the WPS process results back from the WINC1500 module and updates the configuration data. **Implementation:** Dynamic

	WDRV_STUB_Assert	Dumps out an error message on serial console and resets itself when the driver asserts. **Implementation:** Dynamic
	WDRV_STUB_GPIO_ChipDisable	Disables the WINC1500 chip. **Implementation:** Dynamic
	WDRV_STUB_GPIO_ChipEnable	Enables the WINC1500 chip. **Implementation:** Dynamic
	WDRV_STUB_GPIO_DeInitialize	Deinitializes the GPIO object for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_STUB_GPIO_Initialize	Initializes the GPIO object for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_STUB_GPIO_ModuleReset	Resets the WINC1500 module. **Implementation:** Dynamic
	WDRV_STUB_GPIO_ModuleUnreset	Unresets the WINC1500 module. **Implementation:** Dynamic
	WDRV_STUB_HardDelay	Waits spinning for the delay milliseconds. **Implementation:** Dynamic
	WDRV_STUB_INTR_Deinit	Deinitializes interrupts for Wi-Fi driver. **Implementation:** Dynamic
	WDRV_STUB_INTR_Init	Initializes interrupts for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_STUB_INTR_SourceDisable	Disables interrupts from the module. **Implementation:** Dynamic
	WDRV_STUB_INTR_SourceEnable	Enables interrupts from the module. **Implementation:** Dynamic
	WDRV_STUB_SPI_Deinitialize	Deinitializes the SPI object for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_STUB_SPI_In	Receives data from the module through the SPI bus. **Implementation:** Dynamic
	WDRV_STUB_SPI_Initialize	Initializes the SPI object for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_STUB_SPI_Out	Sends data out to the module through the SPI bus. **Implementation:** Dynamic

e) Data Types and Constants

	Name	Description
	_WDRV_WINC1500_API_H	This is macro _WDRV_WINC1500_API_H.
	WDRV_STUB_Print	This is macro WDRV_STUB_Print.

Description

This section describes the Application Programming Interface (API) functions of the WINC1500 Wi-Fi Driver.

Refer to each section for a detailed description.

a) Wi-Fi Initialization Functions

WDRV_CLI_Init Function

Initializes the console CLI interface.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
bool WDRV_CLI_Init();
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function initializes the console CLI interface.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

bool WDRV_CLI_Init(void)

WDRV_INTR_Deinit Function

Deinitializes interrupts for Wi-Fi driver.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_INTR_Deinit();
```

Returns

None.

Description

This function deinitializes interrupts for the Wi-Fi driver.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_INTR_Deinit(void)

WDRV_INTR_Init Function

Initializes interrupts for the Wi-Fi driver.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_INTR_Init();
```

Returns

None.

Description

This function initializes interrupts for the Wi-Fi driver.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_INTR_Init(void)

WDRV_SPI_Deinit Function

Deinitializes the SPI object for the Wi-Fi driver.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_SPI_Deinit();
```

Returns

None.

Description

This function deinitializes the SPI object for the Wi-Fi driver.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_SPI_Deinit(void)

WDRV_SPI_Init Function

Initializes the SPI object for the Wi-Fi driver.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_SPI_Init();
```

Returns

None.

Description

This function initializes the SPI object for the Wi-Fi driver.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_SPI_Init(void)

WDRV_GPIO_DeInit Function

Deinitializes the GPIO objects for the Wi-Fi driver.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_GPIO_DeInit();
```

Returns

None.

Description

This function deinitializes the GPIO objects for the Wi-Fi driver.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_GPIO_DeInit(void)

WDRV_EXT_Deinitialize Function

Deinitializes the WINC1500 Wi-Fi driver.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_Deinitialize();
```

Returns

None.

Description

This function deinitializes the WINC1500 driver.

Remarks

None

Preconditions

None.

Function

void WDRV_EXT_Deinitialize(void)

WDRV_WINC1500_ISR Function

Wi-Fi driver (WINC1500-specific) interrupt service routine.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_WINC1500_ISR();
```

Returns

None.

Description

This function is the Wi-Fi driver (WINC1500-specific) interrupt service routine.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void WDRV_WINC1500_ISR(void)

b) Wi-Fi Status Functions

WDRV_EXT_CmdFWVersionGet Function

Retrieves FW version information.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
uint32_t WDRV_EXT_CmdFWVersionGet(uint32_t * major, uint32_t * minor, uint32_t * patch);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function retrieves the module FW version information.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
major	pointer where the major number will be written
minor	pointer where the minor number will be written
patch	pointer where the patch number will be written

Function

uint32_t WDRV_EXT_CmdFWVersionGet(uint32_t *major, uint32_t *minor, uint32_t *patch);

WDRV_EXT_ScanResultGet Function

Reads the selected scan results back from the WINC1500 module.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_ScanResultGet(uint8_t listIndex, WDRV_SCAN_RESULT * p_scanResult);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

After a scan has completed this function is used to read one scan result at a time from the WINC1500 module.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
listIndex	index (0 based list) of the scan entry to retrieve
p_scanResult	pointer to where scan result is written

Function

void WDRV_EXT_ScanResultGet(uint8_t listIndex, WDRV_SCAN_RESULT *p_scanResult)

WDRV_EXT_CmdMacAddressGet Function

Retrieves the WINC1500 MAC address.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
uint32_t WDRV_EXT_CmdMacAddressGet(uint8_t * MacAddr);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function retrieves the WINC1500 MAC address.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
MacAddr	Pointer where MAC address will be written (must point to a 6 bytes buffer)

Function

uint32_t WDRV_EXT_CmdMacAddressGet(uint8_t *MacAddr)

WDRV_EXT_CmdScanGet Function

Reads the number of scan results from the WINC1500 module.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_CmdScanGet(uint16_t * numOfResults);
```

Returns

None.

Description

This function reads the number of scan results from the WINC1500 module.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
numOfResults	pointer where the number of scan results will be written

Function

void WDRV_EXT_CmdScanGet(uint16_t *numOfResults)

WDRV_EXT_CmdSSIDGet Function

Gets the SSID.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_CmdSSIDGet(uint8_t * ssid, uint8_t * length);
```

Returns

None.

Description

This function returns the SSID and SSID Length.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
ssid	pointer to buffer where SSID will be written
length	number of bytes in SSID

Function

void WDRV_EXT_CmdSSIDGet(uint8_t *ssid, uint8_t *length)

c) Wi-Fi External Functions

WDRV_EXT_CmdPowerSavePut Function

Puts the module in power save mode.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
uint32_t WDRV_EXT_CmdPowerSavePut(bool enable, uint8_t mode, uint16_t listenInterval);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

The function places the module in power save mode.

Remarks

This works only with Infrastructure mode. Do not call this in other modes.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
enable	true will put the module in power save mode.
mode	0 : manual mode - not synchronized to AP beacon ; 1. deep automatic mode - ieee802.11 power save mode.
listenInterval	STA wakes up per this beacon interval.

Function

uint32_t WDRV_EXT_CmdPowerSavePut(bool enable, uint8_t mode, uint16_t listenInterval)

WDRV_EXT_HWInterruptHandler Function

Wi-Fi driver (WINC1500-specific) interrupt service routine.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_HWInterruptHandler();
```

Returns

None.

Description

This function is the Wi-Fi driver (WINC1500-specific) interrupt service routine.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void WDRV_EXT_HWInterruptHandler(void)

WDRV_EXT_CmdScanOptionSet Function

Sets scan options.

Implementation: Dynamic

File

wdrv_wilc1000_api.h

C

```
uint32_t WDRV_EXT_CmdScanOptionSet(uint8_t numOfSlots, uint8_t slotTime, uint8_t probesPerSlot, uint8_t
rssiThreshold);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

The function sets scan options.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
numOfSlots	The min number of slots is 2 for every channel, every slot the module will send Probe Request on air, and wait/listen for PROBE RESP/BEACONS for the slotTime.
slotTime	The time that the module will wait on every channel listening to the frames on air.
probesPerSlot	Number of probe requests to be sent per channel scan slot.
rssiThreshold	The RSSI threshold of the AP which will be connected to directly.

Function

uint32_t WDRV_EXT_CmdScanOptionSet(uint8_t numOfSlots, uint8_t slotTime, uint8_t probesPerSlot, uint8_t rssiThreshold);

WDRV_EXT_ModuleUpDown Function

Enables or disables WINC1500 module.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_ModuleUpDown(uint32_t up);
```

Returns

None.

Description

This function enables or disables WINC1500 module.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
up	1: enable; 0: disable.

Function

void WDRV_EXT_ModuleUpDown(uint32_t up)

WDRV_EXT_MulticastFilterSet Function

Sets a multicast address filter.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
uint32_t WDRV_EXT_MulticastFilterSet(uint8_t * addr);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function allows the application to configure up to 8 Multicast address filters on the WINC1500 module.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
addr	the pointer of the multicast mac address.

Function

uint32_t WDRV_EXT_MulticastFilterSet(uint8_t *addr)

WDRV_EXT_CmdConnect Function

Directs the WINC1500 to connect to a Wi-Fi network.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
uint32_t WDRV_EXT_CmdConnect();
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function causes the WINC1500 to connect to a Wi-Fi network. Upon connection, or a failure to connect, an event will be generated.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete and relevant connection parameters must have been set.

Function

uint32_t WDRV_EXT_CmdConnect(void)

WDRV_EXT_CmdDisconnect Function

Directs the WINC1500 to disconnect from a Wi-Fi network.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
uint32_t WDRV_EXT_CmdDisconnect();
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function causes the WINC1500 to disconnect from a Wi-Fi network.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete and a connection must be in progress.

Function

uint32_t WDRV_EXT_CmdDisconnect(void)

WDRV_EXT_CmdNetModeAPSet Function

Sets the Wi-Fi network type to SoftAP.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_CmdNetModeAPSet();
```

Returns

None.

Description

This function sets the Wi-Fi network type to SoftAP.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void WDRV_EXT_CmdNetModeAPSet(void)

WDRV_EXT_CmdNetModeBSSSet Function

Sets the Wi-Fi network type to Infrastructure.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_CmdNetModeBSSSet();
```

Returns

None.

Description

This function sets the Wi-Fi network type to Infrastructure.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void WDRV_EXT_CmdNetModeBSSSet(void)

WDRV_EXT_CmdScanStart Function

Directs the WINC1500 module to start a scan.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
uint32_t WDRV_EXT_CmdScanStart();
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function directs the WINC1500 module to start a scan.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

uint32_t WDRV_EXT_CmdScanStart(void)

WDRV_EXT_CmdSecNoneSet Function

Sets Wi-Fi security to open (no security).

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_CmdSecNoneSet();
```

Returns

None.

Description

This function sets the Wi-Fi security to open. One can only connect to an AP that is running in open mode.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete and in an unconnected state.

Function

void WDRV_EXT_CmdSecNoneSet(void)

WDRV_EXT_CmdSecWEPSet Function

Sets Wi-Fi security to use WEP.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_CmdSecWEPSet(uint8_t * key, uint16_t len);
```

Returns

None.

Description

This function sets the Wi-Fi security to WEP. One can only connect to an AP that is running the same WEP mode.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete and in an unconnected state.

Parameters

Parameters	Description
key	pointer to the WEP key buffer
len	WEP key length

Function

void WDRV_EXT_CmdSecWEPSet(uint8_t *key, uint16_t len)

WDRV_EXT_CmdSecWPASet Function

Sets Wi-Fi security to use WPA/WPA2.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_CmdSecWPASet(uint8_t * key, uint16_t len);
```

Returns

None.

Description

This function sets the Wi-Fi security to WPA/WPA2. One can only connect to an AP that is running the same WPA/WPA2 mode.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete and in an unconnected state.

Parameters

Parameters	Description
key	pointer to the WPA key buffer
len	WPA key length

Function

void WDRV_EXT_CmdSecWPASet(uint8_t *key, uint16_t len)

WDRV_EXT_DataSend Function

Sends data packets to WINC1500 module.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
uint32_t WDRV_EXT_DataSend(uint16_t segSize, uint8_t * p_segData);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function sends data packets to the WINC1500 module.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
seqSize	data size
p_seqData	pointer to the data buffer

Function

uint32_t WDRV_EXT_DataSend(uint16_t segSize, uint8_t *p_segData)

WDRV_EXT_ScanDoneSet Function

Indicates when a scan has completed.

Implementation: Dynamic

File

wdrv_wilc1000_api.h

C

```
void WDRV_EXT_ScanDoneSet();
```

Returns

None.

Description

This function indicates when a scan has completed.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void WDRV_EXT_ScanDoneSet(void)

WDRV_EXT_CmdChannelSet Function

Sets the channel on which to operate.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_CmdChannelSet(uint16_t channel);
```

Returns

None.

Description

This function sets the channel on which to operate.

Remarks

This works only with SoftAP mode. Do not call this in other modes.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
channel	target channel

Function

void WDRV_EXT_CmdChannelSet(uint16_t channel)

WDRV_EXT_CmdSSIDSet Function

Sets the SSID.

Implementation: Dynamic

File

wdrv_wilc1000_api.h

C

```
void WDRV_EXT_CmdSSIDSet(uint8_t * ssid, uint16_t len);
```

Returns

None.

Description

This function sets the SSID and SSID length.

Remarks

Do not include a string terminator in the SSID length. SSIDs are case-sensitive. SSID length must be less than or equal to 32.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
ssid	pointer to SSID buffer
len	number of bytes in SSID

Function

void WDRV_EXT_CmdSSIDSet(uint8_t *ssid, uint16_t len)

WDRV_EXT_CmdFWUpdate Function

Directs the module to start firmware download and upgrade.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_CmdFWUpdate();
```

Returns

None.

Description

This function directs the module to start the firmware download and upgrade.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void WDRV_EXT_CmdFWUpdate(void)

WDRV_EXT_CmdSecWpsSet Function

Sets Wi-Fi security to use WPS.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
uint32_t WDRV_EXT_CmdSecWpsSet(bool pinMode, uint8_t * key, uint16_t keyLen);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function sets the Wi-Fi security to WPS. One can only connect to an AP that supports WPS.

Remarks

None

Preconditions

Wi-Fi initialization must be complete and in an unconnected state.

Parameters

Parameters	Description
pinMode	0: PBC mode; 1: PIN mode
key	pointer of the PIN buffer
keyLen	PIN length

Function

int32_t WDRV_EXT_CmdSecWpsSet(bool pinMode, uint8_t *key, uint16_t keyLen)

WDRV_EXT_CmdTxPowerSet Function

Sets the Tx Power at 3 levels, high, medium and low.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
uint32_t WDRV_EXT_CmdTxPowerSet(uint32_t level);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

The function sets the module's Tx power.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
level	1 : high - 18 dBm PA gain , 2 : medium - 12 dBm PA gain, 3 : low - 6 dBm PA gain.

Function

uint32_t WDRV_EXT_CmdTxPowerSet(uint32_t level)

WDRV_EXT_CmdConnectContextBssidGet Function

Gets the BSSID

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
uint32_t WDRV_EXT_CmdConnectContextBssidGet(uint8_t * bssId);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function gets the current AP's BSSID.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
bssId	pointer where the current AP's BSSID will be written

Function

uint32_t WDRV_EXT_CmdConnectContextBssidGet(uint8_t *bssId)

WDRV_EXT_CmdScanOptionsSet Function

Sets scan options.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
uint32_t WDRV_EXT_CmdScanOptionsSet(uint8_t numOfSlots, uint8_t slotTime, uint8_t probesPerSlot, uint8_t
rssiThreshold);
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

The function sets scan options.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
numOfSlots	The min number of slots is 2 for every channel, every slot the module will send Probe Request on air, and wait/listen for PROBE RESP/BEACONS for the slotTime.
slotTime	The time that the module will wait on every channel listening to the frames on air.
probesPerSlot	Number of probe requests to be sent per channel scan slot.
rssiThreshold	The RSSI threshold of the AP which will be connected to directly.

Function

uint32_t WDRV_EXT_CmdScanOptionsSet(uint8_t numOfSlots, uint8_t slotTime, uint8_t probesPerSlot, uint8_t rssiThreshold);

WDRV_EXT_CmdSSIDSet Function

Sets the SSID.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_CmdSSIDSet(uint8_t * ssid, uint8_t len);
```

Returns

None.

Description

This function sets the SSID and SSID length.

Remarks

SSIDs are case-sensitive. SSID length must be less than or equal to 32.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
ssid	pointer to SSID buffer
len	number of bytes in SSID

Function

void WDRV_EXT_CmdSSIDSet(uint8_t *ssid, uint16_t len)

WDRV_EXT_ScanIsInProgress Function

Check whether host scan is now in progress or not.

Implementation: Dynamic

File

wdrv_wilc1000_api.h

C

```
bool WDRV_EXT_ScanIsInProgress();
```

Returns

- true - Host scan is in progress
- false - Host scan is not in progress

Description

Check whether host scan is now in progress or not.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void WDRV_EXT_ScanIsInProgress(void)

d) Other Functions

WDRV_INTR_SourceDisable Function

Disables interrupts from the module.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_INTR_SourceDisable();
```

Returns

None.

Description

This function disables interrupts from the module.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void WDRV_INTR_SourceDisable(void)

WDRV_INTR_SourceEnable Function

Enables interrupts from the module.

Implementation: Dynamic

File

wdrv_mrf24wn_api.h

C

```
void WDRV_INTR_SourceEnable();
```

Returns

None.

Description

This function enables interrupts from the module.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void WDRV_INTR_SourceEnable(void)

WDRV_EXT_Initialize Function

Initializes the WILC1000 Wi-Fi driver.

Implementation: Dynamic

File

wdrv_wilc1000_api.h

C

```
void WDRV_EXT_Initialize(WDRV_HOOKS const *const ehooks, bool initWait);
```

Returns

None.

Description

This function initializes the WILC1000 Wi-Fi driver, making it ready for clients to use.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
ehooks	pointer to WDRV layer hooks
initWait	true will put WDRV in wait during initialization

Function

void WDRV_EXT_Initialize(WDRV_HOOKS const *const ehooks, bool initWait)

WDRV_EXT_RssiRead Function

Requests RSSI for the connected AP.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
uint32_t WDRV_EXT_RssiRead();
```

Returns

- 0 - Indicates success
- Non-zero value - Indicates failure

Description

This function requests RSSI for the connected AP.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void WDRV_EXT_RssiRead(void)

WDRV_EXT_WPSResultsRead Function

Reads the WPS process results back from the WINC1500 module and updates the configuration data.

Implementation: Dynamic

File

wdrv_winc1500_api.h

C

```
void WDRV_EXT_WPSResultsRead(WDRV_CONFIG * config, uint32_t * status);
```

Returns

None.

Description

After the WPS process has completed, this function is used to read the WPS process results from the WINC1500 module and update the configuration data.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Parameters

Parameters	Description
config	pointer to where configuration data will be updated
status	pointer to where WPS process status will be written

Function

void WDRV_EXT_WPSResultsRead(WDRV_CONFIG *config, uint32_t *status)

WDRV_STUB_Assert Function

Dumps out an error message on serial console and resets itself when the driver asserts.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_Assert(int condition, const char * msg, const char * file, int line);
```

Returns

None.

Description

Dumps out an error message on serial console and resets itself when the driver asserts.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Parameters

Parameters	Description
condition	asserts if false
msg	error message
file	file name
line	line number where driver asserts.

Function

WDRV_STUB_Assert(int condition, const char *msg, const char *file, int line)

WDRV_STUB_GPIO_ChipDisable Function

Disables the WINC1500 chip.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_GPIO_ChipDisable();
```

Returns

None.

Description

This function disables the WINC1500 chip.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_STUB_GPIO_ChipDisable(void)

WDRV_STUB_GPIO_ChipEnable Function

Enables the WINC1500 chip.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_GPIO_ChipEnable();
```

Returns

None.

Description

This function enables the WINC1500 chip.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_STUB_GPIO_ChipEnable(void)

WDRV_STUB_GPIO_DeInitialize Function

Deinitializes the GPIO object for the Wi-Fi driver.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_GPIO_DeInitialize();
```

Returns

None.

Description

This function deinitializes the GPIO object for the Wi-Fi driver.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_STUB_GPIO_DeInitialize(void)

WDRV_STUB_GPIO_Initialize Function

Initializes the GPIO object for the Wi-Fi driver.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_GPIO_Initialize();
```

Returns

None.

Description

This function initializes the GPIO object for the Wi-Fi driver.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_STUB_GPIO_Initialize(void)

WDRV_STUB_GPIO_ModuleReset Function

Resets the WINC1500 module.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_GPIO_ModuleReset();
```

Returns

None.

Description

This function resets the WINC1500 module.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_STUB_GPIO_ModuleReset(void)

WDRV_STUB_GPIO_ModuleUnreset Function

Unresets the WINC1500 module.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_GPIO_ModuleUnreset();
```

Returns

None.

Description

This function unresets the WINC1500 module.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Parameters

Parameters	Description
board	Microchip development kit type (i.e., PIC32MZ EC Starter Kit, PIC32 Ethernet Starter Kit, etc.)

Function

void WDRV_STUB_GPIO_ModuleUnreset(void)

WDRV_STUB_HardDelay Function

Waits spinning for the delay milliseconds.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_HardDelay(uint16_t delay);
```

Returns

None.

Description

This function has driver wait spinining for the delay milliseconds.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Parameters

Parameters	Description
board	duration to spin.

Function

WDRV_STUB_HardDelay(uint16_t delay)

WDRV_STUB_INTR_Deinit Function

Deinitializes interrupts for Wi-Fi driver.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_INTR_Deinit();
```

Returns

None.

Description

This function deinitializes interrupts for the Wi-Fi driver.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Function

void WDRV_STUB_INTR_Deinit(void)

WDRV_STUB_INTR_Init Function

Initializes interrupts for the Wi-Fi driver.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_INTR_Init(void (*isr)(void));
```

Returns

None.

Description

This function initializes interrupts for the Wi-Fi driver.

Remarks

None.

Preconditions

The TCP/IP stack should be initialized.

Parameters

Parameters	Description
isr	function pointer to the interrupt service handler.

Function

void WDRV_STUB_INTR_Init(void (*isr)(void))

WDRV_STUB_INTR_SourceDisable Function

Disables interrupts from the module.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_INTR_SourceDisable();
```

Returns

None.

Description

This function disables interrupts from the module.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void WDRV_STUB_INTR_SourceDisable(void)

WDRV_STUB_INTR_SourceEnable Function

Enables interrupts from the module.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_INTR_SourceEnable();
```

Returns

None.

Description

This function enables interrupts from the module.

Remarks

None.

Preconditions

Wi-Fi initialization must be complete.

Function

void WDRV_STUB_INTR_SourceEnable(void)

WDRV_STUB_SPI_Deinitialize Function

Deinitializes the SPI object for the Wi-Fi driver.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_SPI_Deinitialize();
```

Returns

None.

Description

This function deinitializes the SPI object for the Wi-Fi driver.

Remarks

None.

Preconditions

None.

Function

void WDRV_STUB_SPI_Deinitialize(void)

WDRV_STUB_SPI_In Function

Receives data from the module through the SPI bus.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
bool WDRV_STUB_SPI_In(unsigned char *const buf, uint32_t size);
```

Returns

None.

Description

This function receives data from the module through the SPI bus.

Remarks

None.

Preconditions

SPI driver should be initialized.

Parameters

Parameters	Description
buf	buffer pointer of input data
size	the input data size

Function

bool WDRV_STUB_SPI_In(unsigned char *const buf, uint32_t size)

WDRV_STUB_SPI_Initialize Function

Initializes the SPI object for the Wi-Fi driver.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
void WDRV_STUB_SPI_Initialize();
```

Returns

None.

Description

This function initializes the SPI object for the Wi-Fi driver.

Remarks

None.

Preconditions

None.

Function

void WDRV_STUB_SPI_Initialize(void)

WDRV_STUB_SPI_Out Function

Sends data out to the module through the SPI bus.

Implementation: Dynamic

File

wdrv_winc1500_stub.h

C

```
bool WDRV_STUB_SPI_Out(unsigned char *const buf, uint32_t size);
```

Returns

True - Indicates success False - Indicates failure

Description

This function sends data out to the module through the SPI bus.

Remarks

None.

Preconditions

SPI driver should be initialized.

Parameters

Parameters	Description
buf	buffer pointer of output data
size	the output data size

Function

bool WDRV_STUB_SPI_Out(unsigned char const *buf, uint32_t size)

e) Data Types and Constants

_WDRV_WINC1500_API_H Macro

File

wdrv_winc1500_api.h

C

```
#define _WDRV_WINC1500_API_H
```

Description

This is macro _WDRV_WINC1500_API_H.

WDRV_STUB_Print Macro

File

wdrv_winc1500_stub.h

C

```
#define WDRV_STUB_Print(x) SYS_CONSOLE_PRINT x
```

Description

This is macro WDRV_STUB_Print.

Files

Files

Name	Description
wdrv_winc1500_api.h	WINC1500 wireless driver APIs.
wdrv_winc1500_stub.h	WINC1500 wireless driver stub APIs.

Description

This section lists the source and header files used by the MRF24WN Wi-Fi Driver Library.

wdrv_winc1500_api.h

WINC1500 wireless driver APIs.

Functions

	Name	Description
	WDRV_CLI_Init	Initializes the console CLI interface. **Implementation:** Dynamic
	WDRV_EXT_CmdChannelSet	Sets the channel on which to operate. **Implementation:** Dynamic
	WDRV_EXT_CmdConnect	Directs the WINC1500 to connect to a Wi-Fi network. **Implementation:** Dynamic
	WDRV_EXT_CmdConnectContextBssidGet	Gets the BSSID **Implementation:** Dynamic
	WDRV_EXT_CmdDisconnect	Directs the WINC1500 to disconnect from a Wi-Fi network. **Implementation:** Dynamic
	WDRV_EXT_CmdFWUpdate	Directs the module to start firmware download and upgrade. **Implementation:** Dynamic
	WDRV_EXT_CmdFWVersionGet	Retrieves FW version information. **Implementation:** Dynamic
	WDRV_EXT_CmdMacAddressGet	Retrieves the WINC1500 MAC address. **Implementation:** Dynamic
	WDRV_EXT_CmdNetModeAPSet	Sets the Wi-Fi network type to SoftAP. **Implementation:** Dynamic
	WDRV_EXT_CmdNetModeBSSSet	Sets the Wi-Fi network type to Infrastructure. **Implementation:** Dynamic
	WDRV_EXT_CmdPowerSavePut	Puts the module in power save mode. **Implementation:** Dynamic
	WDRV_EXT_CmdScanGet	Reads the number of scan results from the WINC1500 module. **Implementation:** Dynamic
	WDRV_EXT_CmdScanOptionsSet	Sets scan options. **Implementation:** Dynamic
	WDRV_EXT_CmdScanStart	Directs the WINC1500 module to start a scan. **Implementation:** Dynamic
	WDRV_EXT_CmdSecNoneSet	Sets Wi-Fi security to open (no security). **Implementation:** Dynamic
	WDRV_EXT_CmdSecWEPSet	Sets Wi-Fi security to use WEP. **Implementation:** Dynamic
	WDRV_EXT_CmdSecWPASet	Sets Wi-Fi security to use WPA/WPA2. **Implementation:** Dynamic
	WDRV_EXT_CmdSecWpsSet	Sets Wi-Fi security to use WPS. **Implementation:** Dynamic
	WDRV_EXT_CmdSSIDGet	Gets the SSID. **Implementation:** Dynamic
	WDRV_EXT_CmdSSIDSet	Sets the SSID. **Implementation:** Dynamic
	WDRV_EXT_CmdTxPowerSet	Sets the Tx Power at 3 levels, high, medium and low. **Implementation:** Dynamic
	WDRV_EXT_DataSend	Sends data packets to WINC1500 module. **Implementation:** Dynamic
	WDRV_EXT_Deinitialize	Deinitializes the WINC1500 Wi-Fi driver. **Implementation:** Dynamic
	WDRV_EXT_HWInterruptHandler	Wi-Fi driver (WINC1500-specific) interrupt service routine. **Implementation:** Dynamic
	WDRV_EXT_Initialize	Initializes the WINC1500 Wi-Fi driver. **Implementation:** Dynamic

	WDRV_EXT_ModuleUpDown	Enables or disables WINC1500 module. **Implementation:** Dynamic
	WDRV_EXT_MulticastFilterSet	Sets a multicast address filter. **Implementation:** Dynamic
	WDRV_EXT_RssiRead	Requests RSSI for the connected AP. **Implementation:** Dynamic
	WDRV_EXT_ScanResultGet	Reads the selected scan results back from the WINC1500 module. **Implementation:** Dynamic
	WDRV_EXT_WPSResultsRead	Reads the WPS process results back from the WINC1500 module and updates the configuration data. **Implementation:** Dynamic
	WDRV_WINC1500_ISR	Wi-Fi driver (WINC1500-specific) interrupt service routine. **Implementation:** Dynamic

Macros

	Name	Description
	_WDRV_WINC1500_API_H	This is macro _WDRV_WINC1500_API_H.

Description

WINC1500 wireless driver APIs.

File Name

wdrv_winc1500_api.h

Company

Microchip Technology Inc.

wdrv_winc1500_stub.h

WINC1500 wireless driver stub APIs.

Functions

	Name	Description
	WDRV_STUB_Assert	Dumps out an error message on serial console and resets itself when the driver asserts. **Implementation:** Dynamic
	WDRV_STUB_GPIO_ChipDisable	Disables the WINC1500 chip. **Implementation:** Dynamic
	WDRV_STUB_GPIO_ChipEnable	Enables the WINC1500 chip. **Implementation:** Dynamic
	WDRV_STUB_GPIO_DeInitialize	Deinitializes the GPIO object for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_STUB_GPIO_Initialize	Initializes the GPIO object for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_STUB_GPIO_ModuleReset	Resets the WINC1500 module. **Implementation:** Dynamic
	WDRV_STUB_GPIO_ModuleUnreset	Unresets the WINC1500 module. **Implementation:** Dynamic
	WDRV_STUB_HardDelay	Waits spinning for the delay milliseconds. **Implementation:** Dynamic
	WDRV_STUB_INTR_Deinit	Deinitializes interrupts for Wi-Fi driver. **Implementation:** Dynamic
	WDRV_STUB_INTR_Init	Initializes interrupts for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_STUB_INTR_SourceDisable	Disables interrupts from the module. **Implementation:** Dynamic
	WDRV_STUB_INTR_SourceEnable	Enables interrupts from the module. **Implementation:** Dynamic
	WDRV_STUB_SPI_Deinitialize	Deinitializes the SPI object for the Wi-Fi driver. **Implementation:** Dynamic

	WDRV_STUB_SPI_In	Receives data from the module through the SPI bus. **Implementation:** Dynamic
	WDRV_STUB_SPI_Initialize	Initializes the SPI object for the Wi-Fi driver. **Implementation:** Dynamic
	WDRV_STUB_SPI_Out	Sends data out to the module through the SPI bus. **Implementation:** Dynamic

Macros

	Name	Description
	WDRV_STUB_Print	This is macro WDRV_STUB_Print.

Description

WINC1500 wireless driver stub APIs.

File Name

wdrv_winc1500_stub.h

Company

Microchip Technology Inc.

WINC1500 Socket Mode Driver Library

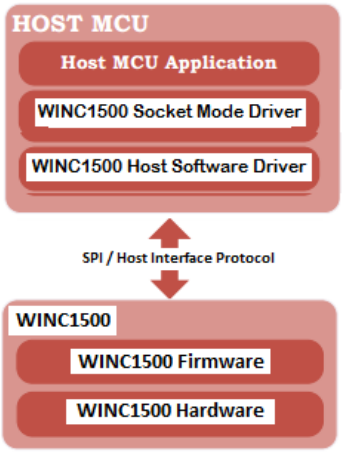
This section provides documentation for the WINC1500 Socket Mode Driver Library.

Introduction

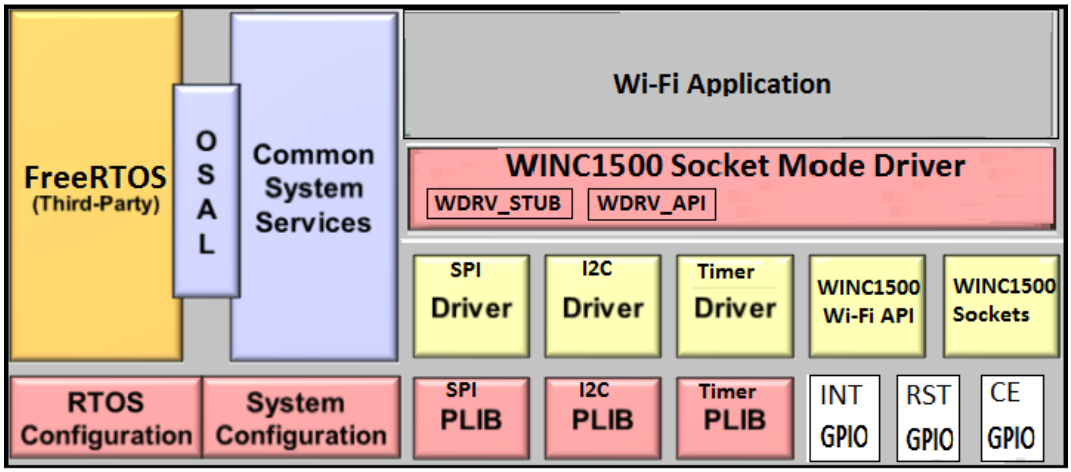
This library provides a low-level abstraction of the WINC1500 Socket Mode Driver Library that is available on the Microchip family of microcontrollers with a convenient C language interface. It can be used to simplify low-level access to the module without the necessity of interacting directly with the module's registers, there by hiding differences from one microcontroller variant to another.

Description

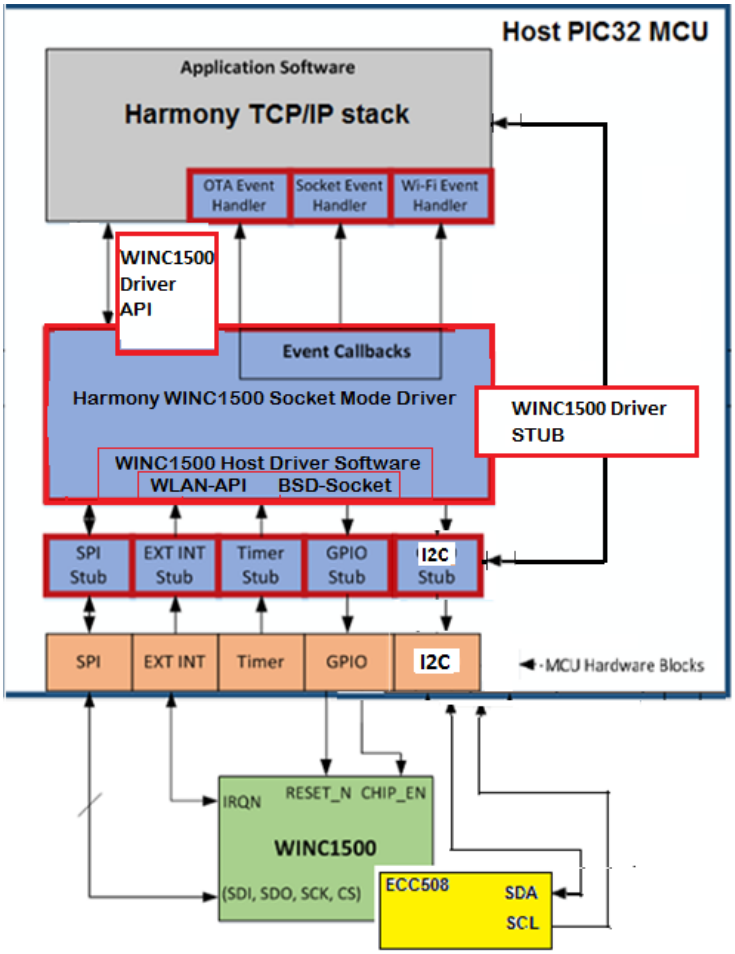
The WINC1500 Socket Mode Driver library is a C library provides the host MCU application with APIs for WLAN and socket operations, off-loading the host MCU TCP/IP networking and transport layer operations to the WINC1500 module firmware.



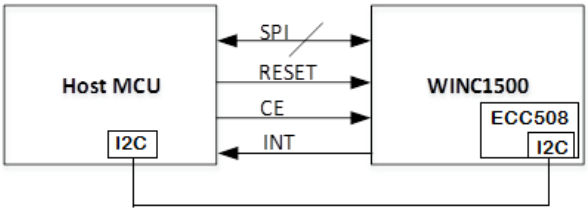
The MPLAB Harmony Integrated Software Framework blocks for the WINC1500 Wi-Fi Application are shown in the following diagram.



The following diagram shows the partitioning of the MPLAB Harmony WINC1500 Socket Mode Driver software on a MCU. Further discussions reference this diagram.



In the previous diagram, the WINC1500 module requires only a SPI interface, a timer, 2 GPIOs for CE and INT, and an interrupt line to connect to the host PIC32 MCU, as shown in the following figure. The figure also shows the I2C interface between the ECC508 device and the Host MCU; however, this feature is not available in the current release and will be available in a future release MPLAB Harmony.



Using the Library

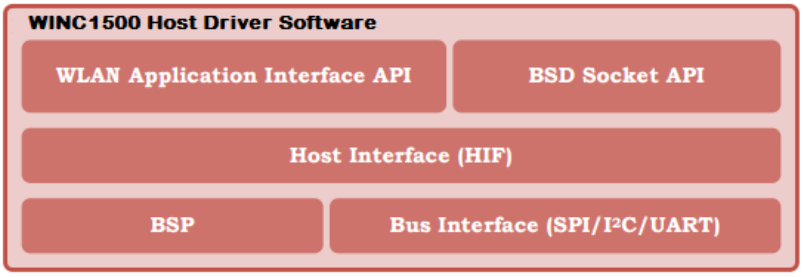
This topic describes the basic architecture of the WINC1500 Wi-Fi Driver Library and provides information and examples on its use.

Description

The WINC1500 Socket Mode Driver Library implements the MPLAB Harmony device driver and communicates with the WINC1500 Host Driver Software to access and control the external WINC1500 module which firmware consists of the following key features:

- Wi-Fi IEEE 802.11 b/g/n STA, AP, and Wi-Fi Direct® modes
- Wi-Fi Protected Setup (WPS)
- Support of WEP, WPA/WPA2 personal, and WPA/WPA2 Enterprise security
- Embedded network stack protocols
- Embedded TCP/IP stack with BSD-style socket API
- Embedded network protocols – DHCP client/server – DNS resolver client – SNTP client for UTC time synchronization
- Embedded TLS security abstracted behind BSD-style socket API
- HTTP Server for provisioning over AP mode
- 8 MB internal Flash memory with OTA firmware upgrade
- Low power consumption with different power saving modes
- SPI, I2C, and UART support

The WINC1500 Host Driver Software is a C library which provides the host MCU application with necessary APIs to perform necessary WLAN and socket operations. The architecture of the WINC1500 Host Driver Software which runs on the host MCU is shown below, and the components of the host driver are described in the following diagram.



WLAN Application Interface API

This module provides an interface to the application for all Wi-Fi operations and any non-IP related operations. This includes the following services:

- Wi-Fi STA management operations
- Wi-Fi Scan
- Wi-Fi Connection management (Connect, Disconnect, Connection status, etc.) – WPS activation/deactivation
- Wi-Fi AP enable/disable
- Wi-Fi Direct enable/disable
- Wi-Fi power save control API
- Wi-Fi monitoring (Sniffer) mode

This interface is defined in the file: m2m_wifi.h.

Socket API

This module provides the socket communication APIs that are mostly compliant with the well-known BSD sockets. To comply with the nature of MCU application environment, there are differences in API prototypes and in usage of some APIs between WINC1500 sockets and BSD sockets.

This interface is defined in the file: socket.h.

Host Interface (HIF)

The Host Interface is responsible for handling the communication between the host driver and the WINC1500 firmware. This includes interrupt handling, DMA and HIF command/response management. The host driver communicates with the firmware in a form of commands and responses formatted by the HIF layer.

The interface is defined in the file: m2m_hif.h.

Board Support Package (BSP)

The Board Support Package abstracts the functionality of a specific host MCU platform. This allows the driver to be portable to a wide range of hardware and hosts. Abstraction includes: pin assignment, power on/off sequence, reset sequence and peripheral definitions (Push buttons, LEDs…etc.).

The minimum required BSP functionality is defined in the file: nm_bsp.h.

Serial Bus Interface

The Serial Bus Interface module abstracts the hardware associated with implementing the bus between the Host and the WINC1500. The serial bus interface abstracts I2C, SPI, or UART bus interface. The basic bus access operations (Read and Write) are implemented in this module as appropriate for the interface type and the specific hardware.

The bus interface APIs are defined in the file: nm_bus_wrapper.h.

Using the WINC1500 Socket Mode Driver Library

The interface to the WINC1500 Socket Driver Library is defined in these header files:

- wdrv_winc1500_api.h
- wdrv_winc1500_stub.h

Any C language source (.c) file that uses the WINC1500 Socket Mode Driver library should include both wdrv_winc1500_api.h and wdrv_winc1500_stub.h.

Abstraction of the WINC1500 Wi-Fi Application

The major blocks of software comprise of a WINC1500 Wi-Fi application are listed and described in the following table.

Software Block	Description
Application Software	This is the application code. Note that three event handlers (OTA, Socket, and Wi-Fi) are part of this block. The event handlers contain callback functions that the driver calls and the application processes.
WINC1500 Socket Mode Driver	This is the WINC1500 Socket Mode driver. The driver API and Stub functions provide application software with setup for the SPI interface between the host MCU and the external WINC1500 module, and provide application software with control and socket data services to the external WINC1500 module via the WINC1500 Host Software Driver.
WINC1500 Socket Mode Driver Stub Functions	The driver Stub functions provide application software with control to the PIC32 MCU hardware blocks (SPI, EXT_INT, Timer, GPIO, I2C) for configuration of Host PIC32 MCU specific hardware and event handling: • SPI Interface • GPIO control • Timer • Interrupt from WINC1500 • Wi-Fi, TPC/IP socket, and OTA event handling
WINC1500 Socket Mode Driver API Functions	The driver Ext functions provide application software access to the driver's system interface for initialize and de-initialize of the driver, as well as setting up the driver hardware interrupt handler and the WINC1500 interrupt service routine. Note: Not all API functions are used in the socket mode driver library. Only these four API functions are used: • WDRV_EXT_Initialize • WDRV_EXT_Deinitialize • WDRV_EXT_HWInterruptHandler • WDRV_WINC1500_ISR
WINC1500 Host Software Driver APIs	The WINC1500 Host Software Driver provides the host MCU application with necessary APIs to perform necessary WLAN and BSD socket operations by offloading these operations to the WINC1500 module firmware. See Section 3 for details of these APIs.

Abstraction Model

This library provides a low-level abstraction of the WINC1500 Wi-Fi module with a convenient C language interface. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The WINC1500 Wi-Fi Library provides the following functionality:

- Wi-Fi library initialization
- Wi-Fi network configuration
- Wi-Fi network connection
- Scanning for existing Wi-Fi networks
- Wi-Fi event processing
- Wi-Fi status
- Wi-FI console commands

Library Overview

Refer to the Driver Library Overview section for information on how the driver operates in a system.

The Library Interface functions are divided into various sub-sections, which address one of the blocks or the overall operation of the WINC1500 Socket Mode Driver Library.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, hard interrupt handler, and interrupt service routine. See wdrv_winc1500_api.h, and wdrv_winc1500_stub.h.
Data Transfer Functions	Provides data transfer functions available in the configuration. See wdrv_winc1500_api.h.
Status Functions	Provides status functions. See m2m_wifi.h
Miscellaneous Functions	Provides miscellaneous driver functions.

How the Library Works

This section describes how the WINC1500 Socket Mode Driver Library operates.

Description

The library provides host PIC32 MCU Wi-Fi application with interface support for the external WINC1500 module and offloads Wi-Fi and BSD socket operations to the WINC1500 module firmware.

Configuring the SPI Driver

This section describes the configuration settings for the WINC1500 Socket Mode Driver Library.

Description

Configuration

The WINC1500 hardware requires a specific configuration of the SPI driver to work correctly. Inside the MHC SPI driver configuration make sure to select:

- SPI clock rate of 8000000 or less
- Input phrase of SPI_INPUT_SAMPLING_PHASE_AT_END
- Clock mode of DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL

Recommended Settings

- Interrupt Driver mode
- Enhanced Buffer mode
- DMA mode enabled
- DMA Block Transfer Size to 512
- Size of DMA Buffer for dummy data to 512
- Ensure when setting up DMA in interrupt mode that the DMA interrupts are a higher priority than the SPI Driver interrupt

Examples

```
/*** SPI Driver Configuration ***/
#define DRV_SPI_NUMBER_OF_MODULES 6
/*** Driver Compilation and static configuration options. ***/
/*** Select SPI compilation units.***/
#define DRV_SPI_POLLED 0
#define DRV_SPI_ISR 1
#define DRV_SPI_MASTER 1
#define DRV_SPI_SLAVE 0
#define DRV_SPI_RM 1
#define DRV_SPI_EBM 0
#define DRV_SPI_8BIT 1
#define DRV_SPI_16BIT 0
#define DRV_SPI_32BIT 0
#define DRV_SPI_DMA 1
/*** SPI Driver Static Allocation Options ***/
#define DRV_SPI_INSTANCES_NUMBER 1
#define DRV_SPI_CLIENTS_NUMBER 1
#define DRV_SPI_ELEMENTS_PER_QUEUE 10
/*** SPI Driver DMA Options ***/
#define DRV_SPI_DMA_TXFER_SIZE 512
#define DRV_SPI_DMA_DUMMY_BUFFER_SIZE 512
/* SPI Driver Instance 0 Configuration */
#define DRV_SPI_SPI_ID_IDX0 SPI_ID_1
```

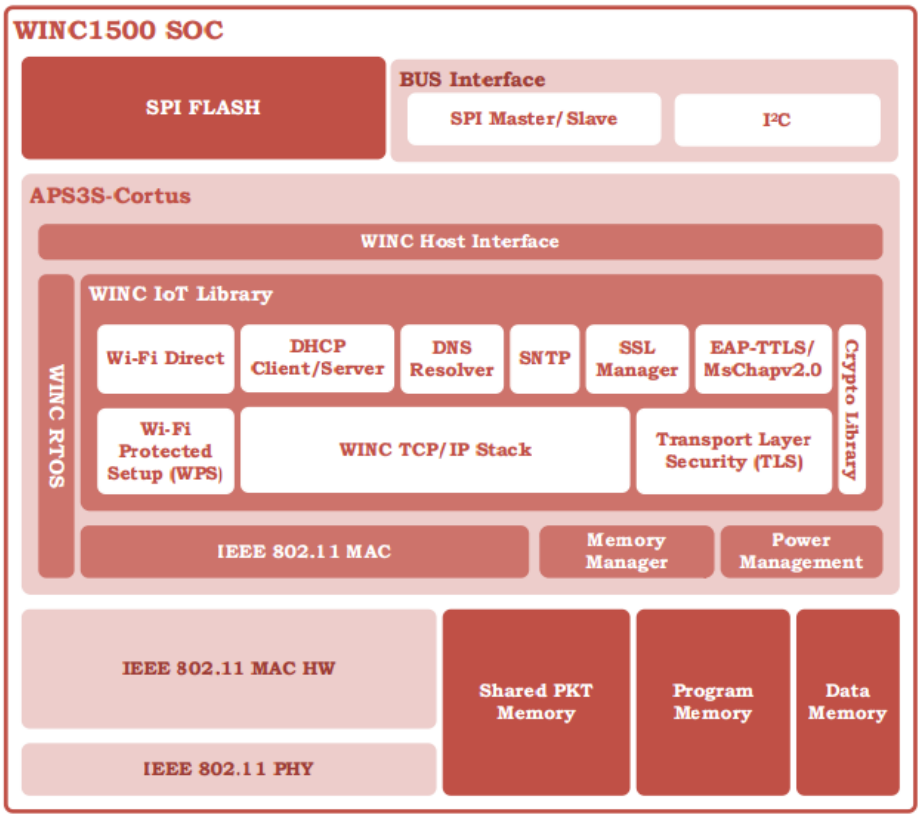
```
#define DRV_SPI_TASK_MODE_IDX0 DRV_SPI_TASK_MODE_ISR
#define DRV_SPI_SPI_MODE_IDX0 DRV_SPI_MODE_MASTER
#define DRV_SPI_ALLOW_IDLE_RUN_IDX0 false
#define DRV_SPI_SPI_PROTOCOL_TYPE_IDX0 DRV_SPI_PROTOCOL_TYPE_STANDARD
#define DRV_SPI_COMM_WIDTH_IDX0 SPI_COMMUNICATION_WIDTH_8BITS
#define DRV_SPI_CLOCK_SOURCE_IDX0 SPI_BAUD_RATE_PBCLK_CLOCK
#define DRV_SPI_SPI_CLOCK_IDX0 CLK_BUS_PERIPHERAL_2
#define DRV_SPI_BAUD_RATE_IDX0 8000000
#define DRV_SPI_BUFFER_TYPE_IDX0 DRV_SPI_BUFFER_TYPE_STANDARD
#define DRV_SPI_CLOCK_MODE_IDX0 DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL
#define DRV_SPI_INPUT_PHASE_IDX0 SPI_INPUT_SAMPLING_PHASE_AT_END
#define DRV_SPI_TRANSMIT_DUMMY_BYTE_VALUE_IDX0 0x00
#define DRV_SPI_TX_INT_SOURCE_IDX0 INT_SOURCE_SPI_1_TRANSMIT
#define DRV_SPI_RX_INT_SOURCE_IDX0 INT_SOURCE_SPI_1_RECEIVE
#define DRV_SPI_ERROR_INT_SOURCE_IDX0 INT_SOURCE_SPI_1_ERROR
#define DRV_SPI_TX_INT_VECTOR_IDX0 INT_VECTOR_SPI1_TX
#define DRV_SPI_RX_INT_VECTOR_IDX0 INT_VECTOR_SPI1_RX
#define DRV_DRV_SPI_ERROR_INT_VECTOR_IDX0 INT_VECTOR_SPI1_FAULT
#define DRV_SPI_TX_INT_PRIORITY_IDX0 INT_PRIORITY_LEVEL1
#define DRV_SPI_TX_INT_SUB_PRIORITY_IDX0 INT_SUBPRIORITY_LEVEL0
#define DRV_SPI_RX_INT_PRIORITY_IDX0 INT_PRIORITY_LEVEL1
#define DRV_SPI_RX_INT_SUB_PRIORITY_IDX0 INT_SUBPRIORITY_LEVEL0
#define DRV_SPI_ERROR_INT_PRIORITY_IDX0 INT_PRIORITY_LEVEL1
#define DRV_SPI_ERROR_INT_SUB_PRIORITY_IDX0 INT_SUBPRIORITY_LEVEL0
#define DRV_SPI_QUEUE_SIZE_IDX0 10
#define DRV_SPI_RESERVED_JOB_IDX0 1
#define DRV_SPI_TX_DMA_CHANNEL_IDX0 DMA_CHANNEL_1
#define DRV_SPI_TX_DMA_THRESHOLD_IDX0 16
#define DRV_SPI_RX_DMA_CHANNEL_IDX0 DMA_CHANNEL_0
#define DRV_SPI_RX_DMA_THRESHOLD_IDX0 16
```

WINC1500 Module Firmware Overview

Provides an overview of the firmware for the WINC1500 Module.

Description

The firmware comprises the Wi-Fi IEEE-802.11 MAC layer and embedded protocol stacks which offload the host MCU. The components of the system are described in the following sub-sections.



WLAN APIs

This WLAN APIs provide an interface to the application for all Wi-Fi operations and any non-IP related operations. This includes the following services:

- Wi-Fi STA management operations
- Wi-Fi Scan
- Wi-Fi Connection management (Connect, Disconnect, Connection status, etc.)
- WPS activation/deactivation
- Wi-Fi AP enable/disable
- Wi-Fi Direct enable/disable
- Wi-Fi power save control API
- Wi-Fi monitoring (Sniffer) mode

This interface is defined in the file: `m2m_wifi.h`.

Socket API

The socket APIs are mostly compliant with the BSD sockets and to comply with the nature of MCU application environment, there are differences in API prototypes and in usage of some APIs between WINC1500 sockets and BSD sockets.

This interface is defined in the file: `socket.h`.

IoT Library

The IoT library provides a set of networking protocols in WINC1500 firmware. It offloads the host MCU from networking and transport layer protocols. The following sections describe the components of WINC1500 IoT library.

- WINC1500 TCP/IP STACK - The WINC TCP/IP is an IPv4.0 stack based on the uIP TCP/IP stack (pronounced micro IP).
- DHCP CLIENT/SERVER - A DHCP client is embedded in WINC1500 firmware that can obtain an IP configuration automatically after connecting to a Wi-Fi network. WINC1500 firmware provides an instance of a DHCP server that starts automatically when WINC AP mode is enabled. When the host MCU application activates the AP mode, it is allowed to configure the DHCP Server IP address pool range within the AP configuration parameters.
- DNS RESOLVER – WINC1500 firmware contains an instance of an embedded DNS resolver. This module can return an IP address by resolving the host domain names supplied with the socket API call gethostbyname.
- SNTP - The SNTP (Simple Network Time Protocol) module implements an SNTP client used to synchronize the WINC1500 internal clock to the UTC clock.
- EAP-TTLS/MSCHAPV2.0 - This module implements the authentication protocol EAP-TTLS/MsChapv2.0 used for establishing a Wi-Fi connection with an AP by with WPA-Enterprise security.
- TRANSPORT LAYER SECURITY - For TLS implementation.

- WI-FI PROTECTED SETUP - For WPS protocol implementation.
- WI-FI DIRECT - For Wi-Fi Direct protocol implementation.
- CRYPTO LIBRARY - The Crypto Library contains a set of cryptographic algorithms used by common security protocols. This library has an implementation of the following algorithms:
  - MD4 - Hash algorithm (Used only for MsChapv2.0 digest calculation)
  - MD5 - Hash algorithm
  - SHA-1 - Hash algorithm
  - SHA-256 - Hash algorithm
  - DES Encryption (Used only for MsChapv2.0 digest calculation)
  - MS-CHAPv2.0 (Used as the EAP-TTLS inner authentication algorithm)
  - AES-128, AES-256 Encryption (Used for securing WPS and TLS traffic)
  - BigInt module for large integer arithmetic (for Public Key Cryptographic computations)
  - RSA Public Key cryptography algorithms (includes RSA Signature and RSA Encryption algorithms)

Host Interface Driver Wi-Fi Events

Provides information on the Host Interface Driver Wi-Fi events.

Description

There are four categories of events:

• Wi-Fi events

• Socket events

• OTA (Over-The-Air) update events

• Error Events

Wi-Fi Events

Wi-Fi events must be customized to suit the application. The WINC1500 socket driver calls the event callback function to notify the application of Wi-Fi events. The p_eventData parameter points to a ‘C’ union of containing all possible Wi-Fi event data. Not all events have data associated with them – in this case the pointer will be NULL. When an event occurs, the event data should be read as soon as possible before another event occurs which will overwrite data from the previous event.

If the event data is to be retrieved outside the event handler function, the utility function m2m_wifi_get_wifi_event_data() returns a pointer to the t_wifiEventData union.

Socket Events

Socket events are handled, but must be customized to suit the application. The WINC1500 driver calls the socket event callback function to notify the application of socket events. The p_eventData parameter points to a ‘C’ union of containing all possible socket event data. Not all events have data associated with them – in this case the pointer will be NULL. When an event occurs, the event data should be read as soon as possible before another event occurs which will overwrite data from the previous event.

If the event data is to be retrieved outside the event handler function, the utility function m2m_wifi_get_socket_event_data() returns a pointer to the t_socketEventData union.

OTA Events

OTA events are associated with downloading and switching to a new WINC1500 firmware image downloaded via the Wi-Fi network. The WINC1500 driver calls the OTA event callback function to notify the application of OTA events. The p_eventData parameter points to a ‘C’ structure containing the OTA event data.

If the event data is to be retrieved outside the event handler function, the utility function m2m_wifi_get_ota_event_data() returns a pointer to the t_otaEventData structure.

Error Events

The application is notified of error events via the callback. Error codes are defined in: `wf_errors.h`.

Configuring the Library

This section describes how to configure the WINC1500 Wi-Fi driver.

Description

The configuration of the WINC1500 Wi-Fi Driver is based on the file `system_config.h`.

This header file contains the configuration selection for the WINC1500 Socket Mode Driver Library. Based on the selection, the WINC1500 Socket Mode Driver Library may support the selected features. These configuration settings will apply to all instances of the WINC1500 Socket Mode Driver Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Building the Library

This section lists the files that are available in the WINC1500 Wi-Fi Driver Library.

Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/driver/wifi/winc1500`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
wdrv_winc1500_stub.h	Contains Stub function prototypes for interfacing to the WINC1500 Wi-Fi Driver.
wdrv_winc1500_api.h	Contains API function prototypes for interfacing to the WINC1500 Wi-Fi Driver.

Required File(s)

MHC ***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source Folder Name	Description
/driver/wifi/winc1500/dev/console/wdrv_winc1500_console.c	Console module for WINC1500 wireless driver.
/driver/wifi/winc1500/dev/gpio/wdrv_winc1500_eint.c	External interrupt handler for WINC1500 wireless driver.
/driver/wifi/winc1500/dev/gpio/wdrv_winc1500_gpio.c	GPIO interface for WINC1500 wireless driver.
/driver/wifi/winc1500/dev/spi/wdrv_winc1500_spi.c	Support SPI communications to the WINC1500 module.
/driver/wifi/winc1500/dev/timer/wdrv_winc1500_timer.c	Timer functions for WINC1500 wireless driver.
/driver/wifi/winc1500/osal/wdrv_winc1500_osal.c	OS abstraction layer for WINC1500 wireless driver.
/driver/wifi/winc1500/wireless_driver_extension/common/source/nm_common.c	This module contains common APIs implementations.
/driver/wifi/winc1500/wireless_driver_extension/driver/source/m2m_hif.c	This module contains M2M host interface API implementations.
/driver/wifi/winc1500/wireless_driver_extension/driver/source/m2m_ota.c	WINC1500 IoT OTA Interface.
/driver/wifi/winc1500/wireless_driver_extension/driver/source/m2m_periph.c	WINC1500 Peripherals Application Interface.
/driver/wifi/winc1500/wireless_driver_extension/driver/source/m2m_wifi.c	This module contains M2M Wi-Fi APIs implementation.
/driver/wifi/winc1500/wireless_driver_extension/driver/source/nmasic.c	This module contains WINC1500 ASIC specific internal APIs.
/driver/wifi/winc1500/wireless_driver_extension/driver/source/nmbus.c	This module contains WINC1500 bus APIs implementation.
/driver/wifi/winc1500/wireless_driver_extension/driver/source/nmdrv.c	This module contains WINC1500 M2M driver APIs implementation.
/driver/wifi/winc1500/wireless_driver_extension/driver/source/nmspi.c	This module contains WINC1500 SPI protocol bus APIs implementation.
/driver/wifi/winc1500/wireless_driver_extension/socket/source/socket.c	WINC1500 BSD Compatible Socket Interface.
/driver/wifi/winc1500/wireless_driver_extension/spi_flash/source/spi_flash.c	WINC1500 SPI flash interface.

/driver/wifi/winc1500/wireless_driver_extension/wdrvext_winc1500.c	WINC1500 wireless driver extension.
/driver/wifi/winc1500/wireless_driver_extension/winc1500_fw_update.c	WINC1500 firmware update support.
/driver/wifi/winc1500/wireless_driver_extension/winc1500_task.c	Entry point of WINC1500 core driver.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	The WINC1500 Wi-Fi Driver controller has no optional files.

Module Dependencies

The WINC1500 Socket Mode Driver Library depends on the following modules:

- SPI Driver Library
- SPI Flash Driver Library
- WINC1500 Wi-Fi Driver Ethernet Mode Library
- Timer Driver Library
- USART Driver Library

Library Interface

This section describes the Application Programming Interface (API) functions of the WINC1500 Socket Mode Driver. Refer to the *WINC1500 Wi-Fi Driver Library* > Library Interface section for information on the APIs in that library that are also used by the WINC1500 Socket Mode Driver.

Refer to each section for a detailed description.

WINC1500 Firmware Update Utility

Refer to WINC1500 Firmware Update Guide for detailed information on using the firmware update utility.

Index

L

M

N

O

P

R

S

T

U

V

W