MPLAB Harmony Test Harness User's Guide

MPLAB Harmony Integrated Software Framework

MPLAB Harmony Test Harness User's Guide

This section describes how to effectively use the MPLAB Harmony Test Harness.

Description

This user's guide describes how to use the MPLAB Harmony Test Harness to manage tests to verify that a library functions as expected and is compatible with the different execution models supported by MPLAB Harmony.

Additional information on the Test Harness Library and example tests is available in the Test Libraries Help.

Introduction

Introduces the MPLAB Harmony Test Harness.

Description

The Test Harness can execute up to 16 tests, each with up to five state machine "Tasks" functions (see **Note**). Each test implements a single test. Each test can have any number of sub-tests. If all sub-tests pass, the test passes. If any sub-test fails, the test fails. If all test pass, the harness reports an overall passing result. If any test fails, the harness reports an overall failure result.

Note: These limitations are imposed by the MHC configuration support and can be exceeded by modifying the system configuration source code as necessary. However, doing so is beyond the scope of this documentation.

The library help and usage documentation defines the library's interface and expected behavior. A test that fulfills the requirements of the Test Harness, must be written to call the library's interface and to verify its expected behavior. Configurations of the Test Harness project must then be created to validate the library's behavior in each of the supported execution environments.

The Test Harness supports the following MPLAB Harmony execution environments:

- Bare Metal (No OS), Polled Execution
- Bare Metal (No OS), Interrupt-driven Execution
- RTOS-Based (Multi-threaded), Polled Execution
- RTOS-Based (Multi-threaded), Interrupt-driven Execution

Error messages, individual test results and other messages are given as they occur, provided that a SYS_DEBUG output method is available and has been configured. Otherwise, a debugger must be used to check the results stored in an internal data structure of the Test Harness once the Test Harness has gone into it's idle state (at which point it executes a hard-coded debug breakpoint when built with debug support).

Assumptions and Prerequisites

Provides information on prerequisites and outlines assumptions on items the user should be familiar with before using the Test Harness.

Description

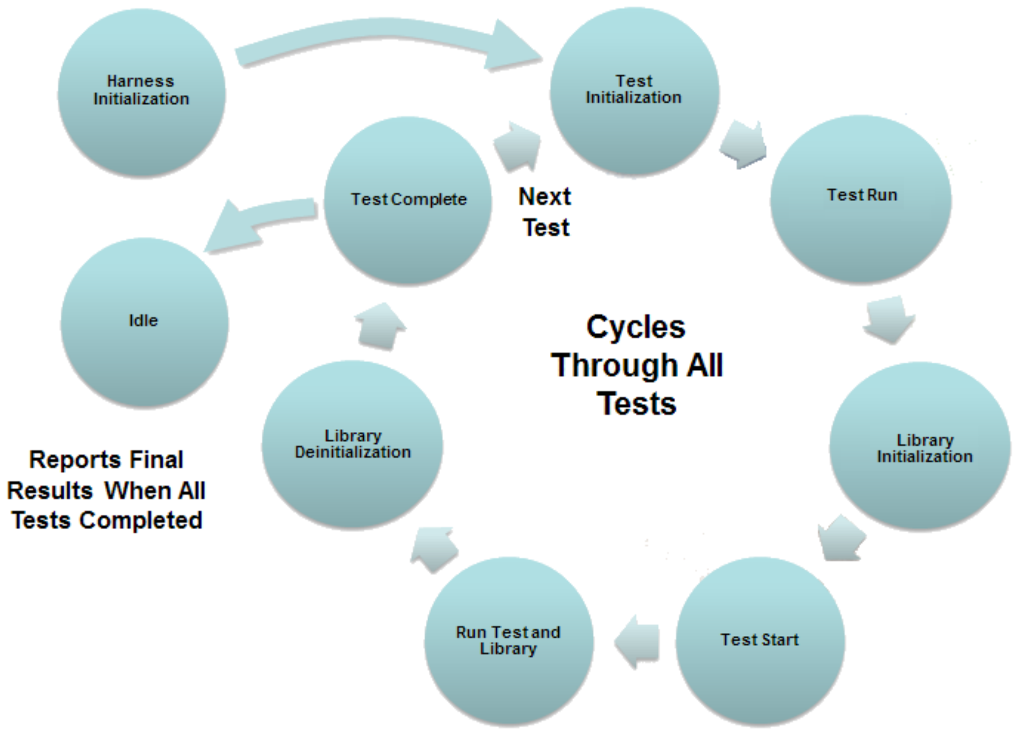
This user's guide assumes that you are familiar with the MPLAB X IDE, XC32 language tools, PIC32, C-language programming, and MPLAB Harmony, and that you have met all MPLAB Harmony development prerequisites. See the MPLAB Harmony Help Prerequisites section for additional information.

Test Harness State Machine

Describes the State Machine for the Test Harness Library.

Description

The basic logic of the Test Harness state machine is shown in the following figure.



The test harness is initialized by calling the TEST_Initialize function and passing in a pointer to a data structure that identifies how many tests to run and provides all of the information necessary to manage the tests and library to be tested by each. The test harness state machine then runs as the TEST_HarnessTasks function is called from repeatedly from within the appropriate system loop. After the harness has initialized itself, its state machine will drop into a cycle that repeats for each test in the test list until it runs out of tests. Tests are executed sequentially, one after another. Messages are "printed" using the SYS DEBUG message functions (assuming that they are appropriately configured) and the harness accumulates results. When the last test is complete, the harness will report over-all test results and go idle. In a debug configuration, the "idle"state Tasks execution contains a hard-coded debug breakpoint where the harness will stop execution and return control to the debugger for analysis of the results.

While somewhat simplified for this discussion, the basic cycle for each test is shown in the state machine above diagram. The harness starts off by entering a Test Initialization state that calls the "Initialize" function for the current test, passing it any necessary initialization data. The harness then transitions to a Test Run state where it allows the test's Tasks function(s) to be called from the appropriate system loop so that the test's state machine can run. It stays in this state until the test calls the TEST_LibraryInitialize function to indicate that it is ready for the harness to initialize the library under test. When that happens, the harness transitions to a the Library Initialization state where it calls the "Initialize" function for each instance (up to three) of the library, passing in the necessary initialization data. Then, the harness enters the Test Start state where it call the test's "Start" function to instruct it to begin active testing of the LUT, after which it transitions to the Run Test and Library state where it allows both the test and the LUT state machines to run freely.

The test's state machine, consisting of up to five Tasks functions (limited by the MHC configuration settings), interacts with the API functions provided by the LUT and any other resources required to verify that the LUT is operating as expected. The test reports intermediate results by calling the TEST_HasPassedSubtest function, passing in "true" to indicate success and "false" to indicate a failure. The test can also "print" error or informational messages unsign the SYS DEBUG service, if it is appropriately configured. This continues, with the harness counting sub-tests that pass and fail, until the test either calls the TEST_HasCompleted function, reports that it is idle when its "Status" function is called, or the test harness timeout occurs (which indicates a test failure). When one of these events occurs, the harness transitions to the Library Deinitialization state where it calls the LUT's "Deinitialize" function and checks its status to verify that it deinitialized correctly. Finally, the harness transitions to the Test Complete state where it reports the final results of the current test, advances to the next test in the list, and repeats the cycle.

For each test, if all sub-tests pass (and no timeout occurs), the test harness counts that test as having passed. For each test, the harness itself reports a sub-test pass result if the LUT reports a SYS_STATUS_READY status (or greater) after being initialized and a SYS_STATUS_UNINITIALIZED status after being deinitialized. These are in addition to any sub-test results reported by the test itself. After all tests are completed (or any test fails), the harness "prints" overall test pass/fail results and goes idle.

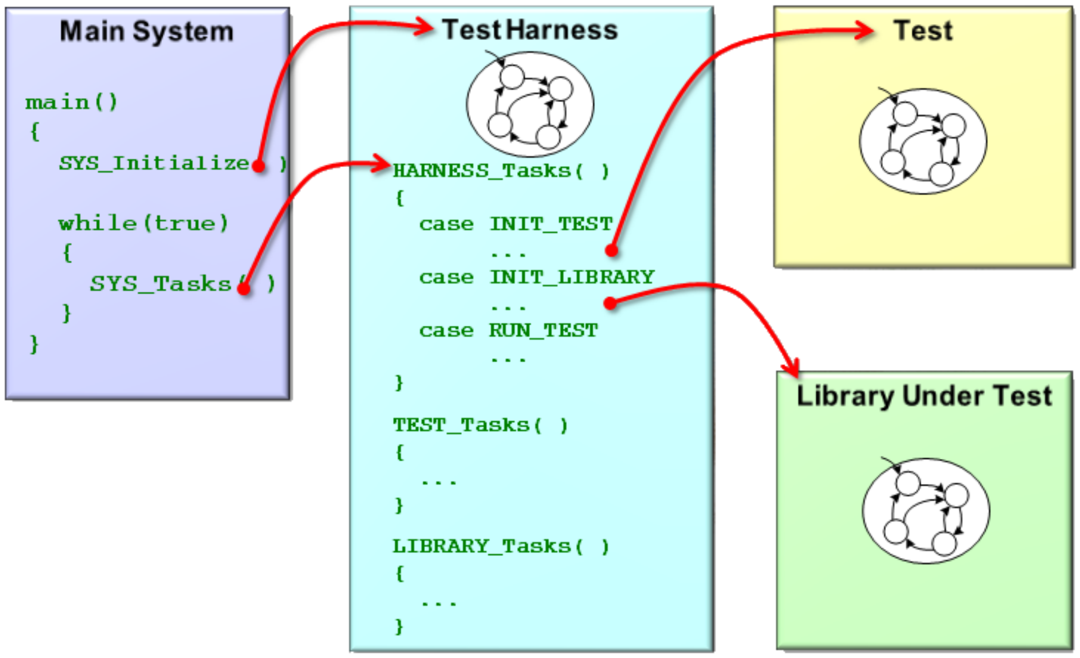
Test Execution Environments

Provides information on test execution environments

Description

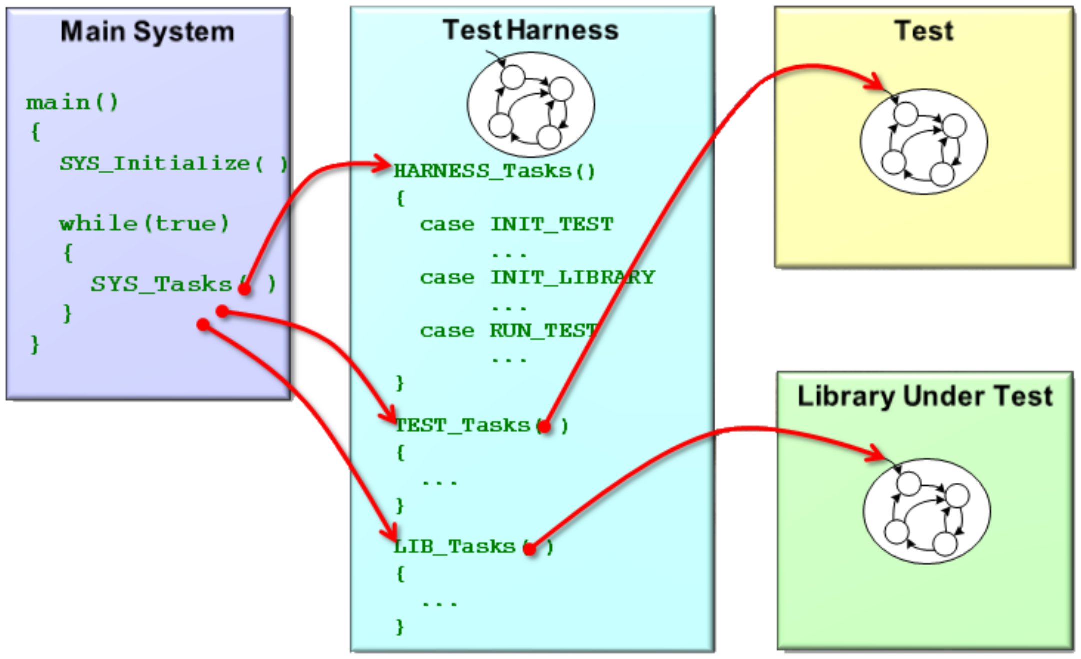
The Test Harness sits between the test and the normal MPLAB Harmony "main" function. In all configurations, the Test Harness controls initialization of the test application and the Library Under Test (LUT), as shown in the following diagram.

Initialization



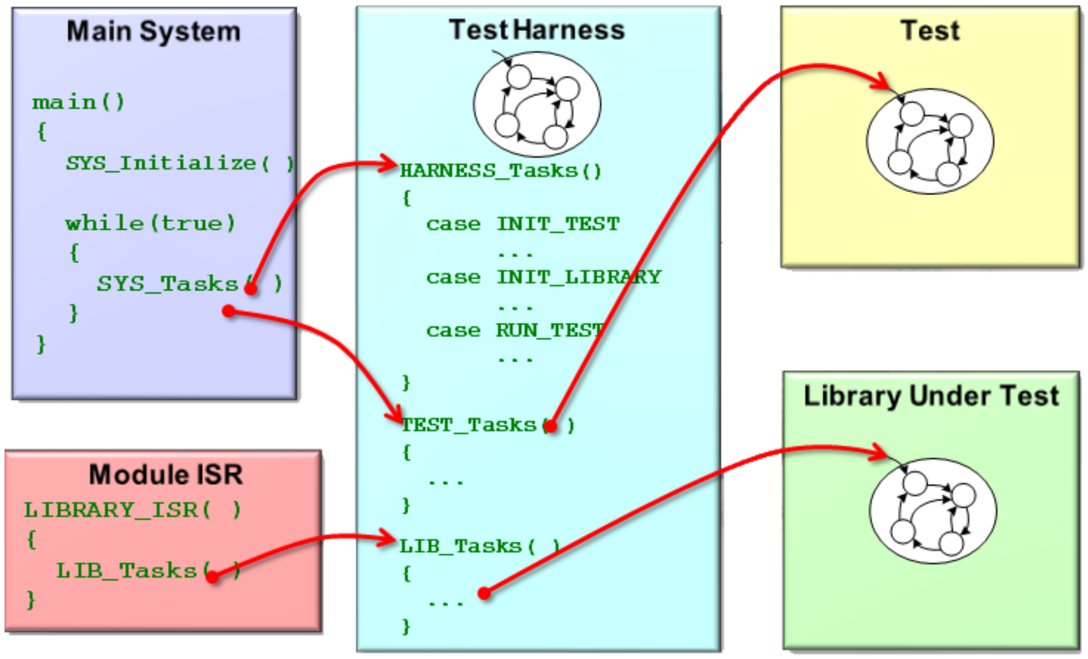
After initialization, the test harness state machine manages the execution of the test and the LUT. However, the test and LUT each execute within their own potentially different tasks environments, as described in the following sections.

Bare-Metal Polled Tests



In a bare-metal polled configuration, all of the state machine tasks functions execute in the context of the main system loop. This includes the tasks function for the test harness, the test, and the LUT. However, the test and library tasks are only executed when the test harness state machine permits so that the harness can track and control them.

Bare-Metal Interrupt-Driven



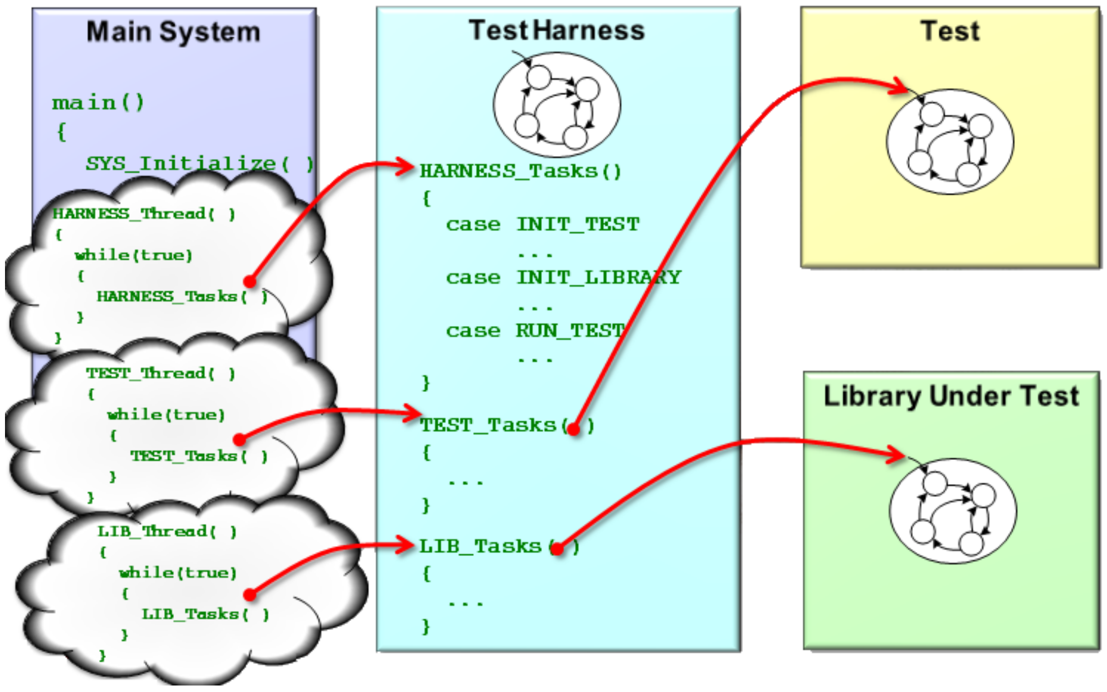
```
main()
{
   SYS_Initialize( )

   while(true)
   {
      SYS_Tasks( )
   }
}
```

```
LIBRARY_ISR( )
{
   LIB_Tasks( )
}
```

```
HARNESS_Tasks()
{
   case INIT_TEST
        ...
   case INIT_LIBRARY
        ...
   case RUN_TEST
        ...
}
TEST_Tasks( )
{
   ...
}
LIB_Tasks( )
{
   ...
}
```

In an interrupt-driven test configuration in a bare-metal environment (with no OS), one or more of the tasks functions for the library under test may run within the context of an Interrupt Service Routine (ISR) that is defined for that peripheral module, depending on what is appropriate for the LUT. These ISR-ready Tasks functions are still called indirectly through the test harness so that it can capture spurious interrupts and manage the execution of the test and LUT.

RTOS-Based Polled



```
main()
{
   SYS_Initialize( )
```

```
HARNESS_Thread( )
{
  while(true)
  {
    HARNESS_Tasks( )
  }
}
```

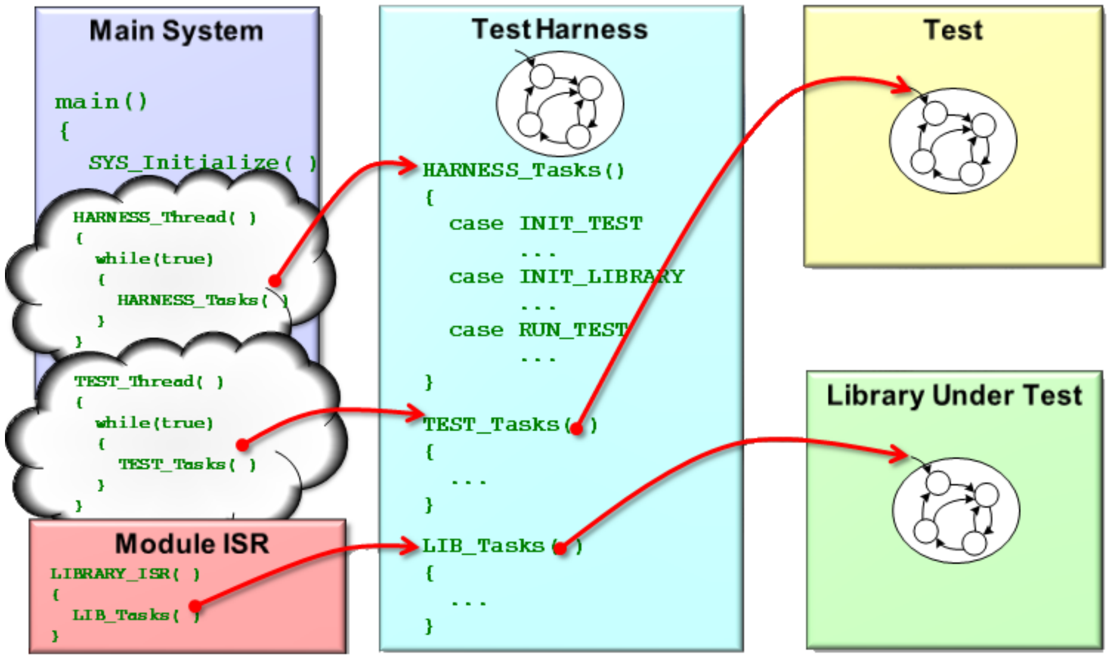
```
TEST_Thread( )
{
  while(true)
  {
    TEST_Tasks( )
  }
}
```

```
LIB_Thread( )
{
  while(true)
  {
    LIB_Tasks( )
  }
}
```

```
HARNESS_Tasks()
{
   case INIT_TEST
        ...
   case INIT_LIBRARY
        ...
   case RUN_TEST
        ...
}
TEST_Tasks( )
{
   ...
}
LIB_Tasks( )
{
   ...
}
```

In an RTOS-based polled configuration, the Tasks functions for the test harness, the test, and the LUT are each called from its own RTOS thread loop. This supports testing of the OSAL mechanisms used to protect the shared resources used by the library when different clients call it from different thread contexts while it is running in its own thread context.

RTOS-Based Interrupt-Driven



In an interrupt-driven test configuration in an RTOS-based environment, one or more of the tasks functions for the library under test may run within the context of an Interrupt Service Routine (ISR) that is defined for that peripheral module, depending on what is appropriate for the LUT. As before (in the bare-metal, interrupt-driven environment), these ISR-ready Tasks functions are called indirectly through the test harness so that it can capture spurious interrupts and manage the execution of the test and LUT. But, in an RTOS environment, the test and the test harness Tasks functions are called from their own RTOS thread loops. Also, there may be RTOS-specific restrictions or requirements on the ISRs themselves.

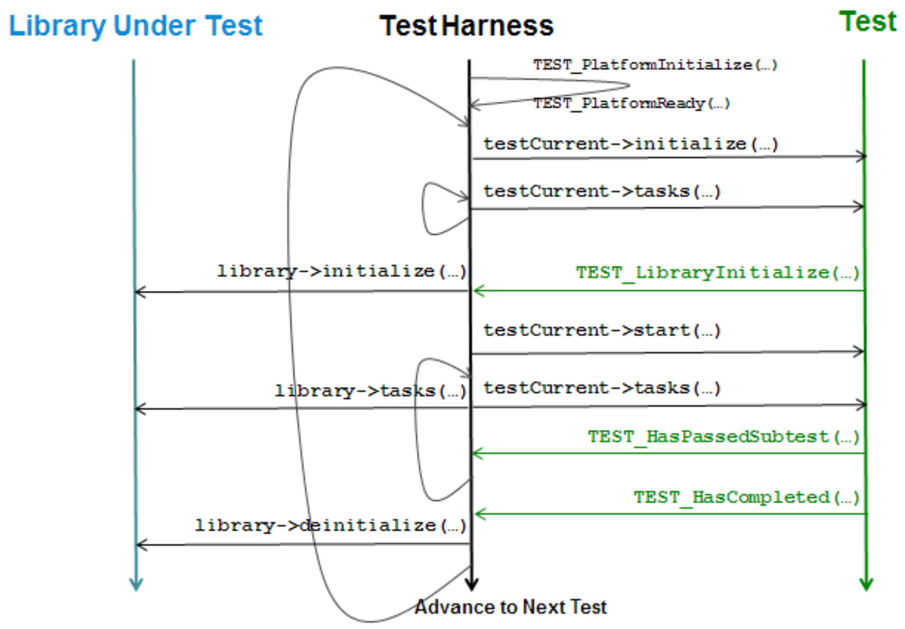
Harness, Test, and Module Interactions

Describes the interactions between the Test Harness, the test, and the LUT.

Description

There are a number of interactions between the Test Harness, the test application, and the LUT, as shown by the following diagram.

Test Harness Interactions



Before running any tests, the harness calls a configuration specific function: TEST_PlatformInitialize. If required, the system configuration can implement this function (usually in `system_init.c`) to initialize any hardware specific items such as timers or special I/O devices. If this process takes time to complete, the code that does so can call TEST_PlatformReady when it is finished. If there are no platform initialization requirements, then this process can be safely ignored as there is a weak implementation of the TEST_PlatformInitialize function built into the harness that simply returns "true" to indicate that the platform is already initialized.

Once the harness is ready, calls the current test's initialization function through a pointer provided in the TEST_DATA structure in the test list. Then, it runs the test's state machine by calling its tasks function using a pointer from the same test data structure. When the test indicates it is ready for the harness to initialize the library (by calling TEST_ModuleUnderTestInitialize), the harness then calls the library's initialization function using a pointer to it provided in a LIBRARY_UNDER_TEST data structure. After successfully initializing the library, the harness starts the actual testing process by calling the test's start function (also using a pointer in the TEST_DATA structure).

After all the initializations are completed and the test has been started, the harness continues calling the tasks functions of both the test and the library (in potentially very different contexts, as described previously). While the test is running, it calls the TEST_HasPassedSubtest function, passing in a Boolean (“true” or “false”) to indicate the results of individual sub-tests. When the test is finished, it reports an idle status when its "Status" function is called (this is not shown in the diagram) or it can call the TEST_HasCompleted function to end the test prematurely. At that point, the harness deinitializes the library, stops running both the test and the library, and advances to the next test.

Configuring the Test Harness

Provides information on how to configure the Test Harness.

Description

The MPLAB Harmony Test Harness has a number of build options, primarily to configure the test time-out timer and OSAL idle time-out values. Normally, you will need to adjust the timer values to account for the selected clock frequency and for hardware resource conflicts if the default timer is used by something else in the system.

Additionally, you should configure the debug message service for the desired output method. If you do not have or do not want to use a debug output method, you should configure the SYS_DEBUG_PRINT and SYS_DEBUG_MESSAGE macros to “throw away” (ignore) the message parameter and hit a breakpoint if the debug error level parameter is greater or equal to SYS_ERROR_ERROR, as shown in the following code example.

```
// *****************************************************************************
/* Debug System Service Configuration Options

   Remarks:
    Configures debug print and message macros to execute a software breakpoint
    if the error level is equal or greater SYS_ERROR_ERROR.
*/

#define SYS_DEBUG_PRINT(level, format, ...)  \
        do{if ((level) <= SYS_ERROR_ERROR) SYS_DEBUG_BreakPoint();}while(0)
#define SYS_DEBUG_MESSAGE(level, message)    \
        do{if ((level) <= SYS_ERROR_ERROR) SYS_DEBUG_BreakPoint();}while(0)
```

For additional information on Test Harness configuration options, refer to the Test Harness Library.

Initializing the Test Harness

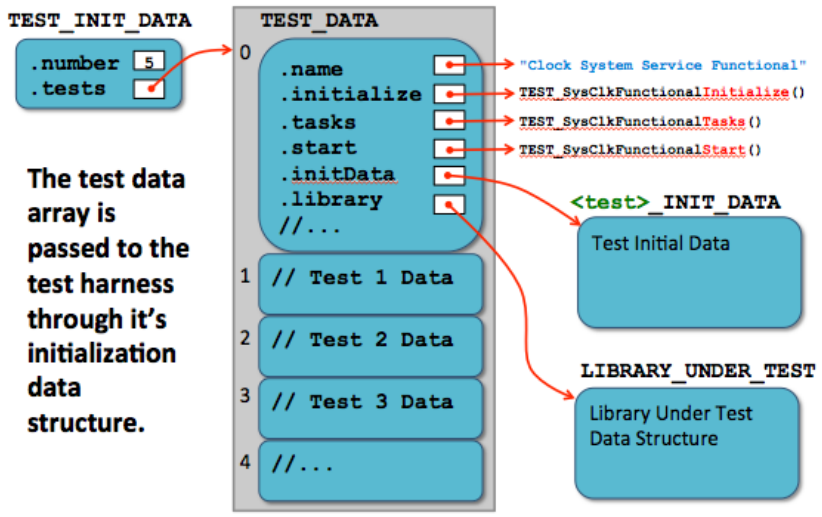
Provides information on initializing the Test Harness.

Description

The MPLAB Harmony Test Harness requires a number of properly initialized data structures to identify the tests it will run and the libraries they will test. The MHC will generate the necessary code to define these initialization structures in the `system_init.c` file of the current configuration. This section describes the relationships between these structures and the fundamental purposes of each.

Note: The Test Harness assumes a one-to-one relationship between a test and a LUT. When testing a stack of related libraries it is best to test them one at a time, starting at the bottom of the stack and working up to higher-level libraries. If testing a desired library requires other libraries to be running, then they will need to run outside of the control of the Test Harness in exactly the way they are normally configured by the MHC.

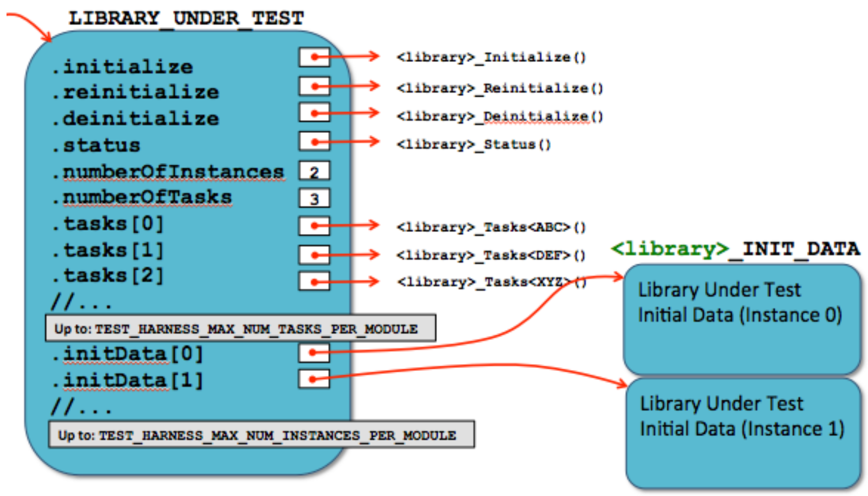
Test Harness Initialization Data


TEST_INIT_DATA
.number 5
.tests
The test data array is passed to the test harness through it's initialization data structure.
TEST_DATA
0
.name
.initialize
.tasks
.start
.initData
.library
//...
"Clock System Service Functional"
TEST_SysClkFunctionalInitialize()
TEST_SysClkFunctionalTasks()
TEST_SysClkFunctionalStart()
<test>_INIT_DATA
Test Initial Data
1 // Test 1 Data
2 // Test 2 Data
3 // Test 3 Data
4 //...
LIBRARY_UNDER_TEST
Library Under Test Data Structure

The test harness initialization data structure (TEST_INIT_DATA) contains the number of tests listed in the test data array and provides a pointer to the array. Each entry in the test data array is a TEST_DATA structure. This structure contains pointers to a display name string as well as the initialize, tasks, start, and status functions for an individual test instance. It also contains pointers to the test's own initial data structure (each test can have it's own "init" structure, just like any Harmony module) and to a "Library Under Test" data structure that identifies the library to be tested by that specific test. There are also few other data items in each structure in the array that are managed by the test harness itself to track the test's progress. So, this array must remain in RAM (i.e., it cannot be declared as "const").

The test's initialization data structure is a standard MPLAB Harmony system module "init" (SYS_MODULE_INIT) data structure that can be extended by the test itself to contain any required initialization data. The LUT data structure from the previous figure is shown in more detail in the following figure.

Module Under Test Data Structure



Each LIBRARY_UNDER_TEST structure contains enough information to initialize and run a single Library Under Test (LUT). It contains pointers to all system-interface functions of the LUT as well as pointers to each module instance’s “Init” data structure. Since a module can have more than one tasks function, the LIBRARY_UNDER_TEST structure contains an array of pointers to the "Tasks" functions and an integer value identifying how many Tasks functions there are. Also, since a dynamic library can support multiple instances of itself, the LIBRARY_UNDER_TEST structure also contains an array of pointers to the LUT’s "Init" data structure. Both of these array sizes are configurable in the source code, but they are values are effectively fixed by the requirements of the MHC templates.

While the MHC will generate the necessary code to initialize these structures, it is necessary to understand what they contain and what purpose each structure serves to effectively enter the appropriate data into the MHC . The data types are defined in the test harness (`<install-dir>/framework/test/test_harness.h`) header file, but they also use data types defined in other MPLAB Harmony system headers. The root of this data structure tree is the TEST_INIT_DATA structure, a pointer to which is passed into the TEST_Initialize function, which is called from the SYS_Initialize function.

It is generally best to start out with a single test add more as needed, later. The initialization of the number member of the TEST_INIT_DATA will adjust during the build if the MHC generated code is used. Be sure to give each test a meaningful name string in the test data structure so you can easily to identify each test in the debug output. The test harness uses the pointers to the initialize, tasks, start, and status functions in the sample test data structure to execute the test, so be sure that these match the names of the functions defined by your test. (**Note:** These can be given as NULL if you want to do an initial build test before you have implemented your test.)

During test development, leave the index member at zero (0) and define the initData member as NULL. You can initialize them later, once you have implemented your test and defined its initialization data structure type. Of course, this structure will also need to be initialized in the `system_init.c` file before it is referenced by the TEST_DATA structure, test, the MHC has no knowledge of it so you will have to initialize it yourself manually.

If your test requires knowledge of the initial data passed to the library under test (and it will if it verifies the initial settings of the library), the LIBRARY_UNDER_TEST data structure will need pointers to the "init" data structures for each instance of the library. Of course, these will need to be cast to SYS_MODULE_INIT structure pointers; just like they are when they are passed to the library’s "initialize" routine. Notice that the tasks member is an array of pointers to the LUT’s tasks functions. It must have exactly as many initialized entries as specified in the numberOfTasks member. This number is limited to no more than the value defined by the TEST_HARNESS_MAX_NUM_TASKS_PER_MODULE setting. Unfortunately, this value is not configurable from within the MHC as it is restricted by MHC templates. However, it can be modified by hand in the source code.

After initializing these data structures, it is a good idea to do a test build and verify that the project has no errors or warnings. You can also debug the test harness state machine and verify that it recognizes and executes your test. If the harness is executed with no test list data, it will report an error and to into the idle (APP_TH_STATE_IDLE) state, providing a failure result (harness.result == false).

Creating a New Test Harness Project

Describes the process to create a new MPLAB Harmony Test Harness project.

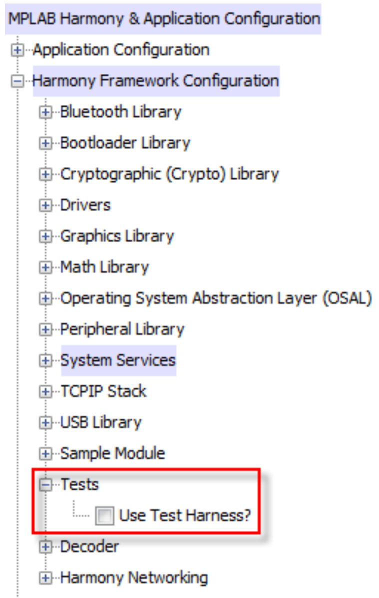
Description

A MPLAB Harmony test harness project can effectively integrate a number tests for multiple modules. However, it is not efficient to attempt to maximize the number of tests within a single project. It is better to create a comprehensive test plan that defines the appropriate test requirements and use it to identify a strategic set of tests, test projects, and project configurations to validate a specific library.

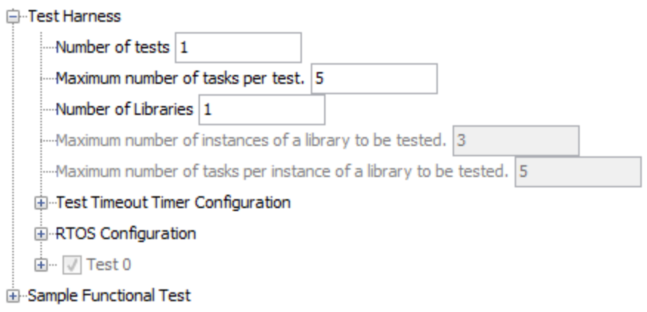
To create a new MPLAB Harmony test harness project, perform the following steps.

Note: These steps are listed sequentially, but it is usually best to work incrementally. Generate your initial configuration with a minimum of changes to the defaults and start developing your test. As you complete the test development, come back to the MHC and add the new test functions and information, regenerate, the configuration, then build, and debug your test.

1. Using MPLAB X IDE, create a new MPLAB Harmony project.
   - Be sure to select the appropriate MPLAB Harmony installation and PIC32 processor
   - Rename the project and its initial (default) configuration to something appropriate for your test purposes
   - Be sure to set the new project as the "main" project
   - Set optimization level to zero (–O0) to facilitate debugging. Clear the “Enable recommended compiler optimizations” option in the "File Generation" dialog when you click **Generate** in the MHC.
   - It is usually easiest to start with a bare-metal, polled configuration as that is the easiest to debug
2. Open the MHC and use it to create the initial test configuration.
3. Add and configure the library under test and any required dependencies.
4. Select **Use Test Harness?** within the *Harmony Framework Configuration* > *Tests* options.

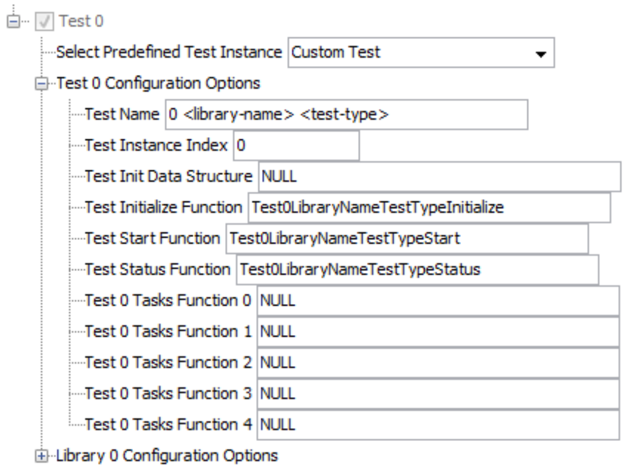


5. Configure the test harness. Refer to the online Help associated with each option for a description of what it means. The test harness directly utilizes a timer (via the Timer PLIB) to provide a timeout capability and OSAL functions, even if no RTOS is selected.



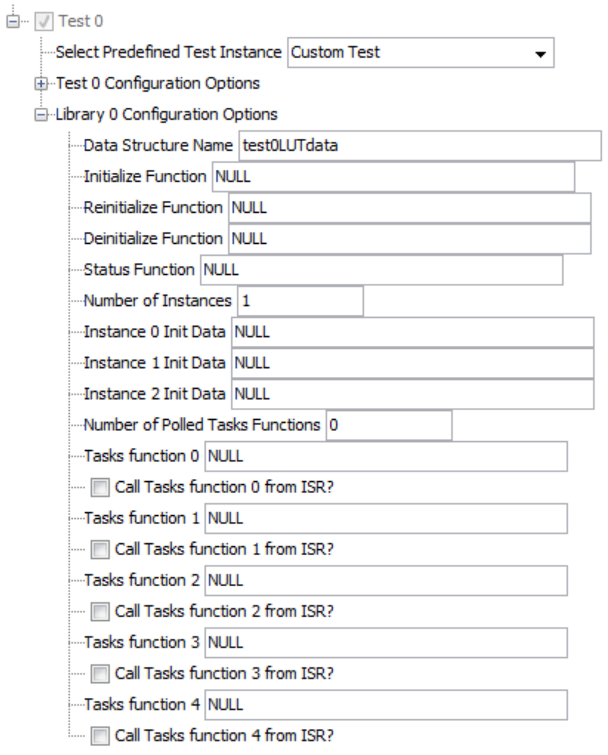
Test Harness
Number of tests 1
Maximum number of tasks per test. 5
Number of Libraries 1
Maximum number of instances of a library to be tested. 3
Maximum number of tasks per instance of a library to be tested. 5
Test Timeout Timer Configuration
RTOS Configuration
Test 0
Sample Functional Test

6. Configure each test, entering the names of the test, its required functions and data structure and other information. Note that the MPLAB Harmony installation does provide some predefined tests. These can be selected and used to test supported pre-existing libraries. However, you will normally need to configure a "Custom Test" and fill in all of the required information manually as no predefined test will be provided for a newly created library.



Test 0
- Select Predefined Test Instance: Custom Test
- Test 0 Configuration Options
  - Test Name: 0 <library-name> <test-type>
  - Test Instance Index: 0
  - Test Init Data Structure: NULL
  - Test Initialize Function: Test0LibraryNameTestTypeInitialize
  - Test Start Function: Test0LibraryNameTestTypeStart
  - Test Status Function: Test0LibraryNameTestTypeStatus
  - Test 0 Tasks Function 0: NULL
  - Test 0 Tasks Function 1: NULL
  - Test 0 Tasks Function 2: NULL
  - Test 0 Tasks Function 3: NULL
  - Test 0 Tasks Function 4: NULL
- Library 0 Configuration Options

7. Enter the library configuration options for the library under test that is associated with each test. This will include the library's function names and initialization data structure name as well as information on the number of instances, polled and interrupt-driven tasks functions (and associated interrupt details).



Test 0
- Select Predefined Test Instance: Custom Test
- Test 0 Configuration Options
- Library 0 Configuration Options
  - Data Structure Name: test0LUTdata
  - Initialize Function: NULL
  - Reinitialize Function: NULL
  - Deinitialize Function: NULL
  - Status Function: NULL
  - Number of Instances: 1
  - Instance 0 Init Data: NULL
  - Instance 1 Init Data: NULL
  - Instance 2 Init Data: NULL
  - Number of Polled Tasks Functions: 0
  - Tasks function 0: NULL
  - Call Tasks function 0 from ISR?
  - Tasks function 1: NULL
  - Call Tasks function 1 from ISR?
  - Tasks function 2: NULL
  - Call Tasks function 2 from ISR?
  - Tasks function 3: NULL
  - Call Tasks function 3 from ISR?
  - Tasks function 4: NULL
  - Call Tasks function 4 from ISR?

8. Generate the configuration. This will insert the test harness library into your project and generate the necessary test list, initial data structures and other configuration code. You may need to make updates to these later as you develop your tests so that you can easily execute them under control of the test harness.

Example

```
/* Test List */
static TEST_DATA testList[] =
{
    {
        .name       = "Sample Module Functional Test 0",
        .initialize = TEST_SampleFunctionalInitialize,
        .tasks      = {
                        TEST_SampleFunctionalTasks,
                        TEST_SampleFunctionalTasks,
                        NULL,
                        NULL,
                        TEST_SampleFunctionalTasks,
                      },
        .start      = TEST_SampleFunctionalStart,
        .status     = TEST_SampleFunctionalStatus,
        .index      = 0,
        .initData   = (SYS_MODULE_INIT *)&testSampleInit,
        .library    = &testSample0LUTdata
    }
};

/* Test Harness Initialization Structure */
static const TEST_INIT_DATA testData =
{
    .numberOfTests  = sizeof(testList)/sizeof(TEST_DATA),
    .tests          = testList
};
```

9. Implement your test as required to test the desired library.
10. Create new project configurations for each new execution environment, hardware platform, or test configuration.

After each configuration change, you should do a test build. If you have entered names for functions or data structures that do not yet exist, you will see build errors for them. If you do not enter any new information, the MHC will place "NULL" pointers in those locations and the project will build, but of course, the harness will not call your test until it has the required configuration information.

Developing a Test

Provides information on developing a test application.

Description

A test should exercise the API of the library under test and verify that the expected results occur. Doing so may require significant development effort. It may also require development of specific supporting hardware. Each test can have any number of sub-tests. A sub-test is counted any time a sub-test result is reported to the test harness by calling the TEST_HasPassedSubtest function with a true or false parameter value. The test harness keeps a count of every time this function is called and a count of how many times it was called with a true parameter. If these two numbers are equal, then the harness counts that the test has passed once it has completed. To indicate that the test has completed, the test application can either call the TEST_HasCompleted function or report a SYS_STATUS_TEST_COMPLETED status and enter its idle state. Once either of these conditions is true, the harness will no longer call the test's tasks function and the test effectively stops running.

A test consists of the following primary functions and any supporting code to exercise and verify the operation of the Library Under Test (LUT).

- Test_<test-short-name>Initialize
- Test_<test-short-name>Tasks
- Test_<test-short-name>Start
- Test_<test-short-name>Status

The Initialize function places the test in its initial state and initializes any internal data, but it does not interact with the LUT. The Tasks function maintains the test's state machine and performs the actual test. And, the Start function tells the test when the LUT has been initialized and it is safe to interact with it.

It is easiest to develop a test by starting from an existing sample test and modifying it to suit your needs. Sample test files are in the MPLAB Harmony installation in the following locations.

- `<install-dir>/framework/test/test_sample_functional.h`
- `<install-dir>/framework/test/src/test_sample_functional.c`

Test Development Process

To get started developing a test application perform the following steps.

1. Copy the `test_sample_functional.c` and `test_sample_functional.h` files from the locations listed previously.
2. Rename these files to reflect the short name of your test and add them to your project.
3. Edit the test's `.c` file, as follows:
   - Rename any references to the sample module with references to the library under test
   - Be sure to change the name of the test's header file in the `#include` statement
   - Be sure to change the state enumeration names and local data type names
   - Delete the *GIVE_DATA" and *GET_DATA" states and the functions that implement them. They are not needed and will be replaced by states that you will define. (Keep the *INIT, *INIT_WAIT, *INITIAL_DATA, *COMPLETED, and *IDLE states. Normally you will reuse these.)
   - Delete all members of the *LIB_DATA structure, except the state and initData member. You will reuse them.
   - Rename the testSampleFunc static global to variable to represent the short name of your test
   - In the VerifyInitialData function, replace the internal logic of this function with a single call to TEST_HasPassedSubtest(false) and return the new <test_STATE_COMPLETED state. You will need to implement this function appropriately later.
   - Delete the AbleToSendNewData and AbleToReadSentData functions. You will not need them.
   - Rename the TestSampleFuncionalInitialize function to reflect the short name of your test and delete unneeded initialization coded
   - Rename the TEST_SampleFunctionStart function to reflect the short name of your test
   - Rename the TEST_SampleFunctionTasks function to reflect the short name of your test and delete code for unneeded states transitions
4. Edit the test's `.h` file.
   - Rename any references to the sample library with references to the library under test
   - Be sure to change the name of the _TEST_SAMPLE_FUNCTIONAL_H macro
   - Delete the inclusion of the `sample_module.h` file, including the API header for the library under test instead
   - Rename the test application's interface functions (both the prototype and the comment block) to match the new names given in the `.c` file
5. Rebuild the project and fix any build errors.
   - The sample function test was designed to test multiple instances of the sample library. If your test is also designed to test multiple instances of the LUT, then you can rename and use the <test>_LIB_INSTANCES_NUMBER configuration option. But, be sure to define it in the `system_config.h` file to set internal array sizes.
   - Fix any editing errors and change any references to the sample library to refer to the desired library under test
6. Once you have a clean build of a test application skeleton, you need to add it to the test application list as described in Initializing the Test Harness and begin developing the test to exercise and validate the module under test.

Reviewing Results

Provides information on reviewing results.

Description

The Test Harness accumulates results given to it by the tests. If SYS DEBUG output is supported and configured, then results will be displayed textually on the debug monitor terminal used. However, depending upon the testing needs or the hardware used, it may not be possible to support a debug output method. In this case, it is necessary to utilize the debugger to analyze the TEST_HARNESS_DATA structure to determine the results of the test. The following members of this structure give the over-all results once the test harness enters its idle state.

- testsCount - This is the total number of test applications executed
- testsPassed - This is the number of test applications that reported no sub-test failures, resulting in a passing result
- result - This is the final overall result, which is true (1) if all sub-tests in all test applications passed; otherwise, false (0) is returned

Note: Using debugger scripting, it is possible to automate the programming, execution, and results analysis process so that automated testing may occur.

Adding the Test Application to the Test List

Description

In order for the Test Harness to initialize and run a test application (and the module it tests), it must be added to the test list. To add a test application to the test list, you must create and initialize specific members of the TEST_DATA structure that makes up that test's entry in the test list array, as shown in the following example.

Example: Array of Tests

```
// *****************************************************************************
/* Array of Tests

   This is an array of structures that provide data necessary for the harness to
   run a test.  It lists all tests that the harness will run
*/

static TEST_DATA tests[] =
{
    {
        .name       = "USART Driver Functional",
        .initialize = TEST_DrvUsartFuncInitialize,
        .tasks      = TEST_DrvUsartFuncTasks,
        .start      = TEST_DrvUsartFuncStart,
        .index      = 0,
        .initData   = (SYS_MODULE_INIT *)&testDrvUsartFuncInitData,
        .library    = &testDrvUsartFuncData
    }
};
```

The name member is a pointer to a string that contains the display name of the test, when debut output is supported. The initialize, tasks, and start members are pointers to the initialize, tasks, and start functions (respectively) of the test application. The index member identifies to which instance of the test application this test list entry refers. (For now, this is normally zero.) The initData member is a pointer to a structure that holds any initialization data required by the test application. In the sample test this structure is used to obtain pointers to the initialization data passed to the LUT. And finally, the library member is a pointer to a LIBRARY_UNDER_TEST structure (described below) that contains information about the LUT.

The LIBRARY_UNDER_TEST structure must be initialized to provide all the necessary information about the system interface of the library under test so that the harness can initialize and run the library, as shown in the following example.

Example: Initializing the Library Under Test

```
/* USART Driver Module */
LIBRARY_UNDER_TEST lutDriverUsart =
{
    .initialize         = DRV_USART_Initialize,
    .reinitialize       = NULL,
    .deinitialize       = DRV_USART_Deinitialize,
    .status             = DRV_USART_Status,
    .numberOfInstances  = 2,
    .numberOfTasks      = 3,
    .tasks              = {
                            DRV_USART_TasksTransmit,
                            DRV_USART_TasksReceive,
                            DRV_USART_TasksError
                          },
    .initData           = {
                            (SYS_MODULE_INIT *)&drvUsart0InitData,
                            (SYS_MODULE_INIT *)&drvUsart1InitData
                          }
};
```

In the previous example, the LUT is the USART driver. It has initialize, deinitialize, and status functions as well as three tasks functions (DRV_USART_TasksTransmit, DRV_USART_TasksReceive, and DRV_USART_TasksError). Pointers to these functions are given as initialization values for the initialize, reinitialize, deinitialize, and tasks array members of this structure. The number of entries in the tasks array are configurable by defining the TEST_HARNESS_MAX_NUM_TASKS_PER_MODULE option and the number of tasks functions that must be called to support the LUT must be identified by initializing the numberOfTasks member, as shown above. Lastly, the number of instances of the LUT to be tested must be identified by initializing the numberOfInstances member and pointers to the initialization data of each instance (in order, starting from instance 0) must be given as initialization values of the initData array (also starting from entry 0 and going in order). The number of possible entries in the initData array is defined by the TEST_HARNESS_MAX_NUM_INSTANCES_PER_MODULE configuration option.

Finally, a pointer to the test list array is then passed into the initialization structure (TEST_INIT_DATA) of the Test Harness in the `system_init.c` file to make all of this information available to the Test Harness, which then passes it to the test application and the LUT.

Once the test list and all associated data structures are initialized properly, the Test Harness can properly manage the test application and its associated module under test. To add additional test applications and LUTs, add additional entries to the test list and initialize them similarly. If you use the method shown in the Test Harness examples to calculate the size of the test list array the compiler will automatically calculate the number of entries in the array.

Note: The Test Harness assumes a one-to-one relationship between a test and a library under test. When testing a stack of related libraries it is best to test them one at a time, starting at the bottom of the stack and working up to higher-level libraries. If testing a desired library requires other libraries to be running, these libraries will need to run outside of the control of the Test Harness in exactly the way they are normally configured by the MHC.

Adding Additional Configurations

Provides information on adding additional configurations.

Description

It is usually easiest to create a bare-metal polled configuration first because you can more easily debug the harness, test, and library state machines. Once you have a good functional test that passes when running polled, add an interrupt driven configuration and ensure that the library works as expected when interrupt driven. If the library under test does not have an associated interrupt, a timer interrupt can be used to simulate an interrupt driven mode as is done in the “sample” module. While this is not strictly necessary for libraries that are not intended to run in an interrupt-driven mode, it can help make the library more robust.

Once bare metal testing is completed, create RTOS-based configurations and test the library in a polled, multi-threaded environment. Verify thread safety and proper blocking operation and experiment with different relative thread priorities between the test and library threads. Then, add another configuration to test the library running interrupt-driven in an RTOS environment. There are unique issues possible in each of these environments and testing in all of then will create the most reliable library. Also, while use of the MPLAB Harmony OSAL will help make your library RTOS-independent, but it is best to create test configurations for every supported RTOS to verify that there are no resource conflicts or unique issues between the library and a specific supported RTOS.

Refer to Tips and Tricks for the process to add additional configurations.

Test Requirements

Describes test requirements.

Description

While this user's guide does describe how to use the MPLAB Harmony Test Harness to control and execute a test within the different execution environments supported by MPLAB Harmony, it does not describe the requirements of that test.

The MPLAB Harmony compatibility requirements are described in the MPLAB Harmony Compatibility Guide, in the What is MPLAB Harmony? section of the help documentation in the `<install-dir>/doc` folder included in the MPLAB Harmony installation. In that same folder is a fillable PDF form (`harmony_compatibility_worksheet.pdf`) that contains a compatibility checklist worksheet that you can fill out to track and report testing methods and results.

Index