

Class B Library Reference Guide

Table of Contents

1. Introduction.....	4
2. Architecture.....	5
3. Software Revision.....	6
4. Porting.....	7
5. Analysis Metrics.....	8
5.1. MISRA.....	8
6. Demo Application.....	9
7. ADC.....	11
7.1. Overview.....	11
7.2. Functions.....	11
7.3. Macros.....	15
7.4. Enums.....	16
7.5. Data Structure Documentation.....	16
7.6. Source Code Reference.....	17
8. CLOCK.....	18
8.1. Overview.....	18
8.2. Functions.....	18
8.3. Macros.....	20
8.4. Enums.....	21
8.5. Source Code Reference.....	21
9. CPU.....	22
9.1. Overview.....	22
9.2. Functions.....	22
9.3. Macros.....	30
9.4. Enums.....	30
9.5. Global Variables.....	30
9.6. Source Code Reference.....	31
10. CRC.....	32
10.1. Overview.....	32
10.2. Functions.....	32
10.3. Macros.....	34
10.4. Enums.....	34
10.5. Data Structure Documentation.....	34
10.6. Source Code Reference.....	35
11. FLASH.....	36
11.1. Overview.....	36
11.2. Functions.....	36
11.3. Enums.....	41
11.4. Data Structure Documentation.....	41

11.5. Source Code Reference.....	41
12. GPIO.....	43
12.1. Overview.....	43
12.2. Functions.....	43
12.3. Macros.....	49
12.4. Source Code Reference.....	50
13. INTERRUPT.....	51
13.1. Overview.....	51
13.2. Functions.....	51
13.3. Macros.....	62
13.4. Enums.....	74
13.5. Data Structure Documentation.....	75
13.6. Source Code Reference.....	77
14. PC.....	78
14.1. Overview.....	78
14.2. Functions.....	78
14.3. Source Code Reference.....	79
15. SRAM.....	80
15.1. Overview.....	80
15.2. Functions.....	80
15.3. Macros.....	84
15.4. Enums.....	85
15.5. Source Code Reference.....	85
16. TIMER.....	86
16.1. Overview.....	86
16.2. Functions.....	86
16.3. Enums.....	89
16.4. Source Code Reference.....	89
17. COMMON.....	91
17.1. Overview.....	91
17.2. Functions.....	91
17.3. Macros.....	95
17.4. Enums.....	96
17.5. Global Variables.....	97
17.6. Data Structure Documentation.....	97
17.7. Source Code Reference.....	98
18. Microchip Information.....	100
18.1. Trademarks.....	100
18.2. Legal Notice.....	100
18.3. Microchip Devices Code Protection Feature.....	100

1. Introduction

The dsPIC33A Class B Library is a collection of implemented diagnostics APIs of various modules of a microcontroller. These diagnostic APIs help in achieving the IEC 60730 standard to support the Class B compliance. These routines can be directly integrated with the end user's application to test and verify the critical functionalities of a microcontroller.

For more details, refer to the following sections.

2. Architecture

The Class B library utilizes modular architecture, with each diagnostic described in this manual implemented in its own C source file. This allows users to incorporate only the required diagnostic files into their application project, effectively optimizing memory usage.

The API functions for diagnostics in various modules are organized into individual folders, with each folder containing the corresponding source code for its module.

3. **Software Revision**

The dsPIC33A Class B library referred to in this document is at version v3.1.0

4. Porting

The Class B library is usually tested for one superset device in a family of devices. If you need to use the Class B library for another device of the same device family, some configuration changes are needed to use the code. These changes are described in the “Porting Guide” document present in the docs folder.

5. Analysis Metrics

5.1. MISRA

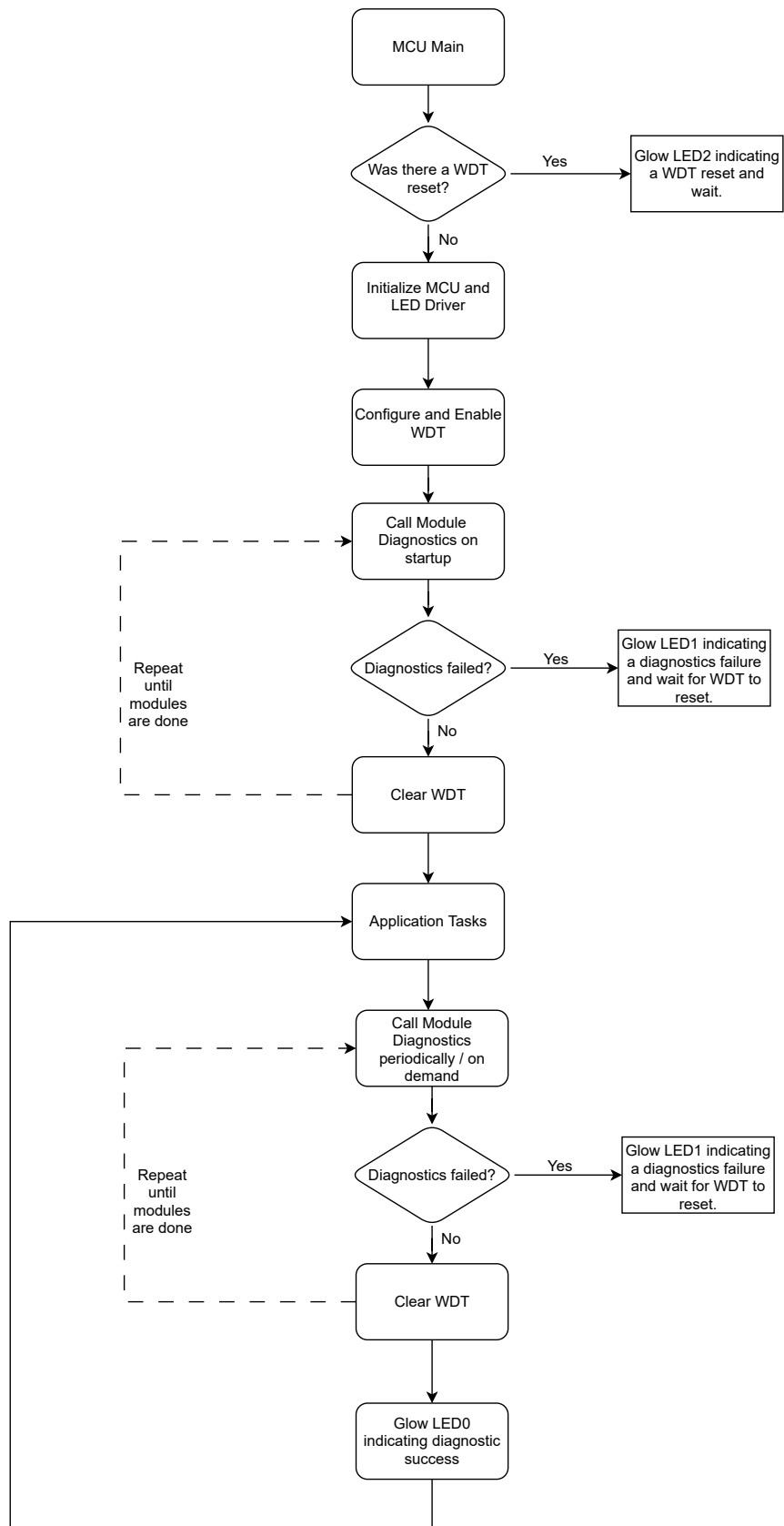
The Static Analysis is performed using the LDRA toolsuite with the MISRA 2023 rules. All the 'Mandatory' rules are satisfied. Barring a few exceptions, all the 'Required' rules are also satisfied.

The exceptions are documented in the code inline as comments with a tag, LDRA_EXCLUDE, with corresponding justification. The 'Advisory' violations are kept as is.

6. Demo Application

The demo application to demonstrate the usage of the Class B library is present in the apps folder. The demo can be run using the Curiosity Development Board (EV74H48A) with the respective device DIM.

The following flowchart shows the demo application structure using the Class B library:



7. ADC

7.1. Overview

This document describes the diagnostics API for the ADC module. The safety requirements described for ADC diagnostics in the Safety Manual are implemented in the following APIs.

- [DIAG_ADC_StartupTest](#)
- [DIAG_ADC_BoundaryMonitorTest](#)
- [DIAG_ADC_LinearityMonotonicityTest](#)

Constraints/Limitations:

- ADC interrupts need to be disabled before calling the diagnostics.
- DAC is used to provide a input signal to ADC for diagnostics. A dedicated DAC1 shall be assigned.

Integration Rules:

- Nil

7.2. Functions

7.2.1. Function Documentation

7.2.1.1. DIAG_ADC_BoundaryMonitorTest()

uint32_t DIAG_ADC_BoundaryMonitorTest (DIAG_ADC_CHANNELS adcChannelNumber, uint32_t boundaryValueLow, uint32_t boundaryValueHigh)

Software Requirement Reference ID : SW_ADC_BOUNDARY_MONITOR_TEST_01 The diagnostic API is aimed at detecting failures in the circuits dedicated to ADC boundary monitoring. On successful detection the diagnostics returns a pass otherwise fail. ADC Interrupts for the corresponding channel shall be disabled before calling the diagnostics.

Parameters:

adcChannelNumber	- ADC channel number
boundaryValueLow	- Lower Boundary Value
boundaryValueHigh	- Higher Boundary Value

Returns:

DIAG_PASS
DIAG_FAIL
DIAG_INVALID_PARAM

Figure 7-1. ADC Boundary Monitor Test

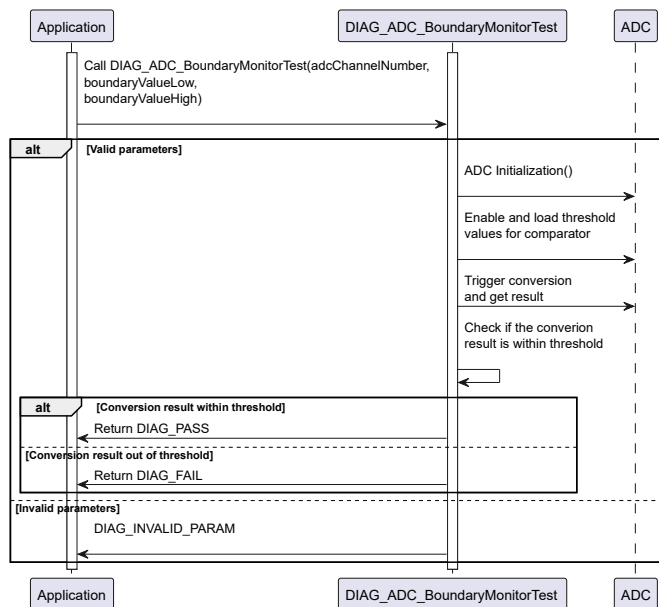
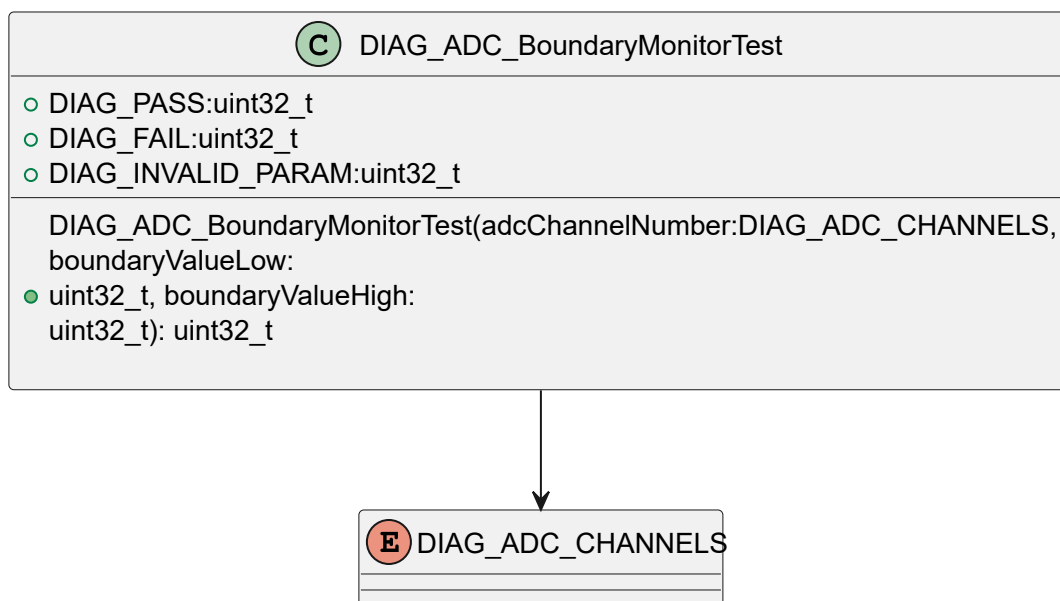


Figure 7-2. ADC Boundary Monitor Class Diagram



7.2.1.2. DIAG_ADC_LinearityMonotonicityTest()

uint32_t DIAG_ADC_LinearityMonotonicityTest (DIAG_ADC_CHANNELS adcChannelNumber, uint8_t tolerance)

Software Requirement Reference ID : SW_ADC_LINEARITY_MONOTONICITY_TEST_01 The diagnostic API performs a linearity and monotonicity diagnostic test on an Analog-to-Digital Converter (ADC) channel using a Digital-to-Analog Converter (DAC) to generate input signals. The goal is to verify that the ADC output increases monotonically and linearly as the input voltage is incremented, and to check for any bits stuck at high or low values. ADC Interrupts for the corresponding channel shall be disabled before calling the diagnostics.

Design:

1. **Input Signal Generation DAC Usage:** A Digital-to-Analog Converter (DAC) is used to generate the input signal for the ADC. **Ramp Creation:** The DAC output is incremented in steady steps, creating a ramp-like analog signal that covers the ADC's input range.
2. **Linearity, Monotonicity, Bit Stuck Checks** Initializes the DAC with the starting input value. Performs an initial ADC conversion and records the result. Gradually increases the DAC output in specified increments, conducting ADC conversions at each increment. In the present design, DAC output is incremented by 16 units. At each increment: Determines the difference between the current and previous ADC readings. Assesses whether this difference falls within the acceptable range defined by the increment and tolerance thereby validating linearity and monotonicity. Increases the failure count if the deviation is outside the permissible limit and flags the test as failed. Updates variables that monitor for bits stuck at high or low states in the ADC output. Sets the previous result to the current value for the next comparison.

Bit-Stuck Verification: After completing all increments, evaluates the tracking variables to identify any bits stuck at high or low levels. Flags the test as failed if any stuck bits are detected.

Parameters:

adcChannelNumber	- ADC channel number
tolerance	- The tolerance in counts, which specifies maximum permissible deviation from the ideal ADC output increment for each test step. Typically, the step size can be given as the tolerance but can be changed as per the noise level in the usage environment.

Returns:

DIAG_PASS
DIAG_FAIL
DIAG_INVALID_PARAM

Figure 7-3. ADC Linearity Monotonicity Test

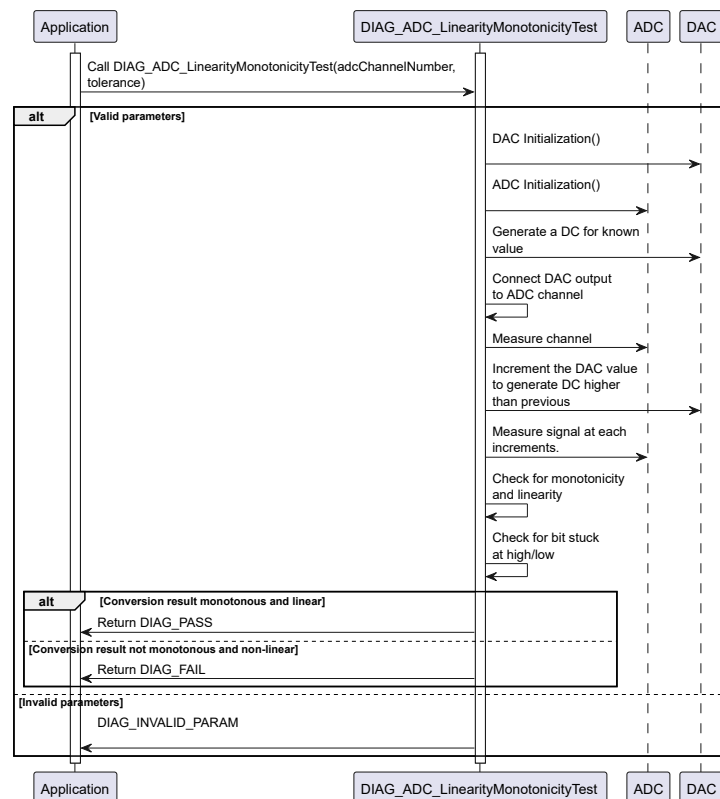
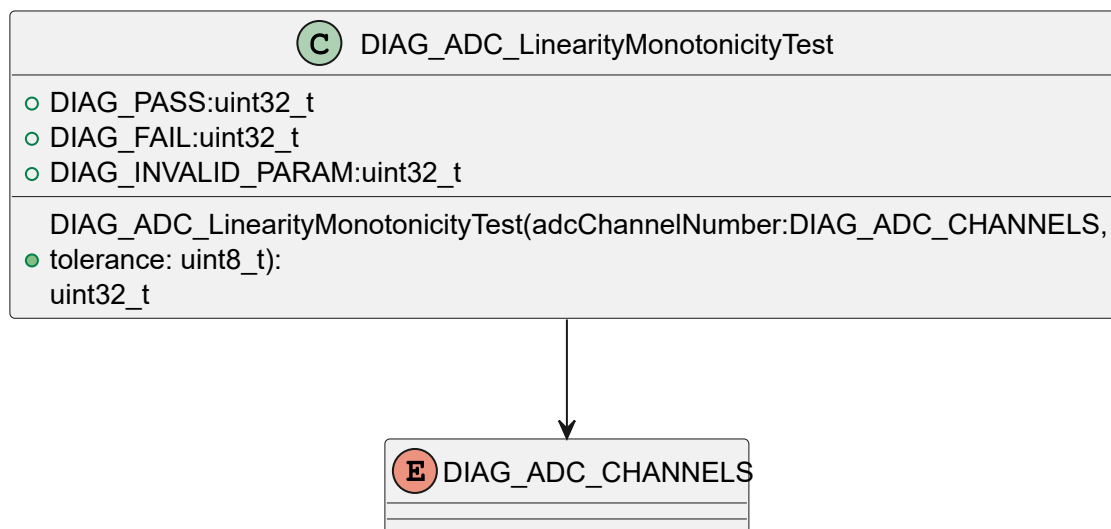


Figure 7-4. ADC Linearity Monotonicity Class Diagram



7.2.1.3. DIAG_ADC_StartupTest()

`uint32_t DIAG_ADC_StartupTest (DIAG_ADC_CHANNELS adcChannelNumber)`

Software Requirement Reference ID : SW_ADC_STARTUP_TEST_01 The diagnostic API is aimed at detecting failures in the overall ADC module. On successful detection the diagnostics returns a pass otherwise fail. ADC Interrupts for the corresponding channel shall be disabled before calling the diagnostics.

Parameters:

<code>adcChannelNumber</code>	- ADC channel number
-------------------------------	----------------------

Returns:

<code>DIAG_PASS</code> <code>DIAG_FAIL</code> <code>DIAG_INVALID_PARAM</code>

Figure 7-5. ADC StartupTest

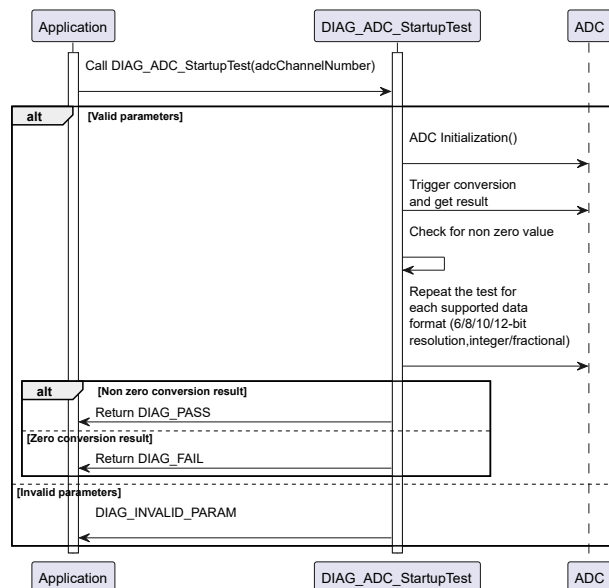
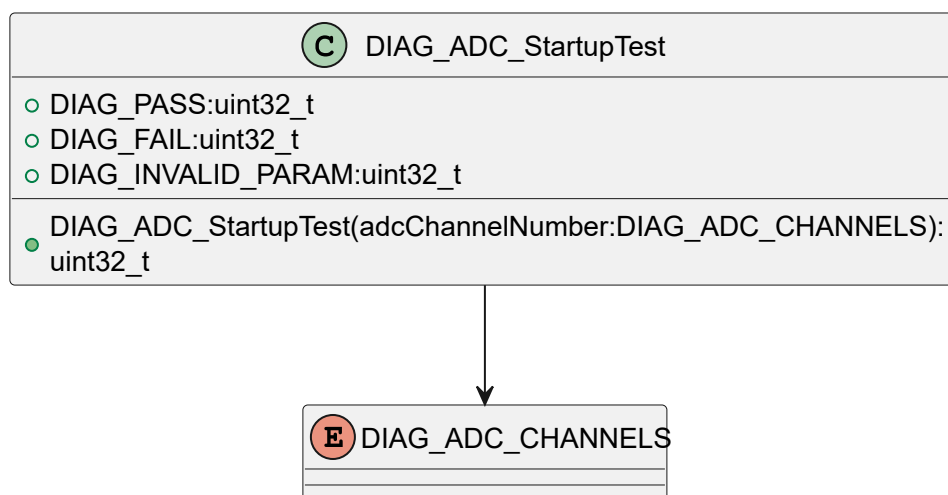


Figure 7-6. ADC StartupTest Class Diagram



7.2.1.4. DIAG_ADC_GetLinearityMonotonicityResult()

DIAG_ADC_LIN_MON_RESULT DIAG_ADC_GetLinearityMonotonicityResult (void)

Software Requirement Reference ID : SW_ADC_LINEARITY_MONOTONICITY_TEST_01The diagnostic API returns the result of linearity monotonicity test.

Parameters:

None

Returns:

DIAG_ADC_LIN_MON_RESULT

7.3. Macros

7.3.1. Definition Documentation

7.3.1.1. DIAG_ADC_LIN_INPUT_START_RANGE

```
#define DIAG_ADC_LIN_INPUT_START_RANGE 0x00D0U
```

The initial input range for ADC linearity diagnostics as determined by the DAC's range available within the device.

7.3.1.2. DIAG_ADC_LIN_INPUT_END_RANGE

```
#define DIAG_ADC_LIN_INPUT_END_RANGE 0x0F30U
```

The end input range for ADC linearity diagnostics as determined by the DAC's range available within the device.

7.4. Enums

7.4.1. Enumeration Type Documentation

7.4.1.1. DIAG_ADC_CHANNELS

```
enum DIAG_ADC_CHANNELS
```

Defines the ADC Channels available.

7.5. Data Structure Documentation

7.5.1. DIAG_ADC_LIN_MON_RESULT Struct Reference

7.5.1.1. Detailed Description

Structures

Maintains the linearity monotonicity test results.

7.5.1.1.1. Data Fields

- uint32_t [resultLinMon](#)
- uint16_t [lastTestFailCurrentResult](#)
- uint16_t [lastTestFailPreviousResult](#)
- uint16_t [failCount](#)
- uint16_t [stuckAtHighValue](#)
- uint16_t [stuckAtLowValue](#)

7.5.1.2. Field Documentation

7.5.1.2.1. resultLinMon

```
DIAG_ADC_LIN_MON_RESULT::resultLinMon
```

Linearity Monotonicity Result.

7.5.1.2.2. lastTestFailCurrentResult

```
DIAG_ADC_LIN_MON_RESULT::lastTestFailCurrentResult
```

ADC result for the last failed current iteration.

7.5.1.2.3. lastTestFailPreviousResult

```
DIAG_ADC_LIN_MON_RESULT::lastTestFailPreviousResult
```

ADC result for the last failed previous iteration.

7.5.1.2.4. failCount

```
DIAG_ADC_LIN_MON_RESULT::failCount
```

Number of failures in the ADC conversions.

7.5.1.2.5. stuckAtHighValue

DIAG_ADC_LIN_MON_RESULT::stuckAtHighValue

Indicates which bit is stuck at high value.

7.5.1.2.6. stuckAtLowValue

DIAG_ADC_LIN_MON_RESULT::stuckAtLowValue

Indicates which bit is stuck at low value.

7.6. Source Code Reference

The following table provides a list of APIs and their location in the source code:

MODULE	DIAGNOSTIC API	SOURCE CODE REFERENCE
ADC	- DIAG_ADC_SetBoundaryValues - DIAG_ADC_IsOutOfBounds - DIAG_ADC_BoundaryMonitorTest	diag_adc_boundary_monitor_test.c
ADC	- DIAG_ADC_LinearityMonotonicityTest - DIAG_ADC_GetLinearityMonotonicityResult	diag_adc_linearity_monotonicity_test.c
ADC	DIAG_ADC_StartupTest	diag_adc_startup_test.c

8. CLOCK

8.1. Overview

This document describes the diagnostics API for the CLOCK module. The safety requirements described for CLOCK diagnostics in the Safety Manual are implemented in the following APIs.

- [DIAG_CLOCK_FscmTest](#)
- [DIAG_CLOCK_SetClockMonFaultStatus](#)

Constraints/Limitations:

- Nil

Integration Rules:

- The time window value WINPR that is chosen for collecting the monitored clock samples should be long enough for collecting the number of samples. Accordingly the low frequency threshold, high frequency threshold for fail condition and for warning condition should be chosen. A good example is provided in the datasheet under section Clock Monitor Module.

8.2. Functions

8.2.1. Function Documentation

8.2.1.1. DIAG_CLOCK_SetClockMonFaultStatus()

void DIAG_CLOCK_SetClockMonFaultStatus (uint32_t status)

Software Requirement Reference ID : SW_FAIL_SAFE_CLOCK_MONITOR_TEST

Sets the clock monitor fault status.

Parameters:

status	The status value to set for clock fault injection.
--------	--

Updates the global clockFaultInjectStatus variable with the provided status value.

8.2.1.2. DIAG_CLOCK_FscmTest()

uint32_t DIAG_CLOCK_FscmTest (DIAG_CLOCK_MON_INST clkMonInstance,
DIAG_CLOCK_MON_REF_CLK monClkSrc, DIAG_CLOCK_MON_REF_CLK refClkSrc,
DIAG_CLOCK_FLTINJ_TYPE fltinjType)

Software Requirement Reference ID : SW_FAIL_SAFE_CLOCK_MONITOR_TEST

Performs a failsafe clock monitor (FSCM) test by configuring a clock monitor instance, injecting a specified clock fault, and verifying fault detection.

This function sets up the selected clock monitor instance with the specified monitored and reference clock sources, enables the monitor, waits for a valid buffer value, injects a chosen fault type, and checks if the fault is detected within a timeout period. The result indicates whether the fault was successfully detected by the hardware.

Parameters:

in	clkMonInstance	The clock monitor instance to be tested. Possible values: - CLOCKMON_INSTANCE_1: Clock Monitor Instance 1 - CLOCKMON_INSTANCE_2: Clock Monitor Instance 2 - CLOCKMON_INSTANCE_3: Clock Monitor Instance 3 - CLOCKMON_INSTANCE_4: Clock Monitor Instance 4 - MAX_CLOCK_MONITOR_INSTANCE: Maximum number of clock monitor instances (not for use as a valid instance)
in	monClkSrc	The clock source to be monitored. Possible values (also used for refClkSrc): - SERIAL_TEST_MODE_CLOCK: Serial test mode clock - FRC: Fast RC oscillator - BFRC: Backup Fast RC oscillator - POSC: Primary oscillator - LPRC: Low-power RC oscillator - PLL1_FOUT: PLL1 output - PLL2_FOUT: PLL2 output - PLL1_VCO: PLL1 VCO - PLL2_VCO: PLL2 VCO - REF2: Reference clock 2 - REF1: Reference clock 1
in	refClkSrc	The reference clock source against which the monitored clock is compared. Uses the same enum and possible values as monClkSrc.
in	fltInjType	The type of fault to inject for testing the clock monitor's detection capability. Possible values: - CLOCK_LOW_FREQ_DRIFT_FLTINJ: Injects a low frequency drift fault - CLOCK_HIGH_FREQ_DRIFT_FLTINJ: Injects a high frequency drift fault - CLOCK_CATASTROPHIC_FLTINJ: Injects a catastrophic clock failure

Returns:

DIAG_PASS if the injected fault is detected by the clock monitor, DIAG_FAIL otherwise.

Note:

- This function is blocking and will wait until either a fault is detected or the timeout expires.
- The function manipulates hardware registers directly and should be used with care.
- The enums DIAG_CLOCK_MON_INST, DIAG_CLOCK_MON_REF_CLK, and DIAG_CLOCK_FLTINJ_TYPE are defined in the relevant header.

See also:

[DIAG_CLOCK_SetClockMonFaultStatus](#)

Example usage:

```
uint32_t result = DIAG_CLOCK_FscmTest(CLOCKMON_INSTANCE_1, FRC, POSC,  
CLOCK_LOW_FREQ_DRIFT_FLTINJ);  
if (result == DIAG_PASS) {  
    // Fault detected successfully  
} else {  
    // Fault not detected  
}
```

Figure 8-1. Clock Fail Safe Clock Monitor Sequence diagram

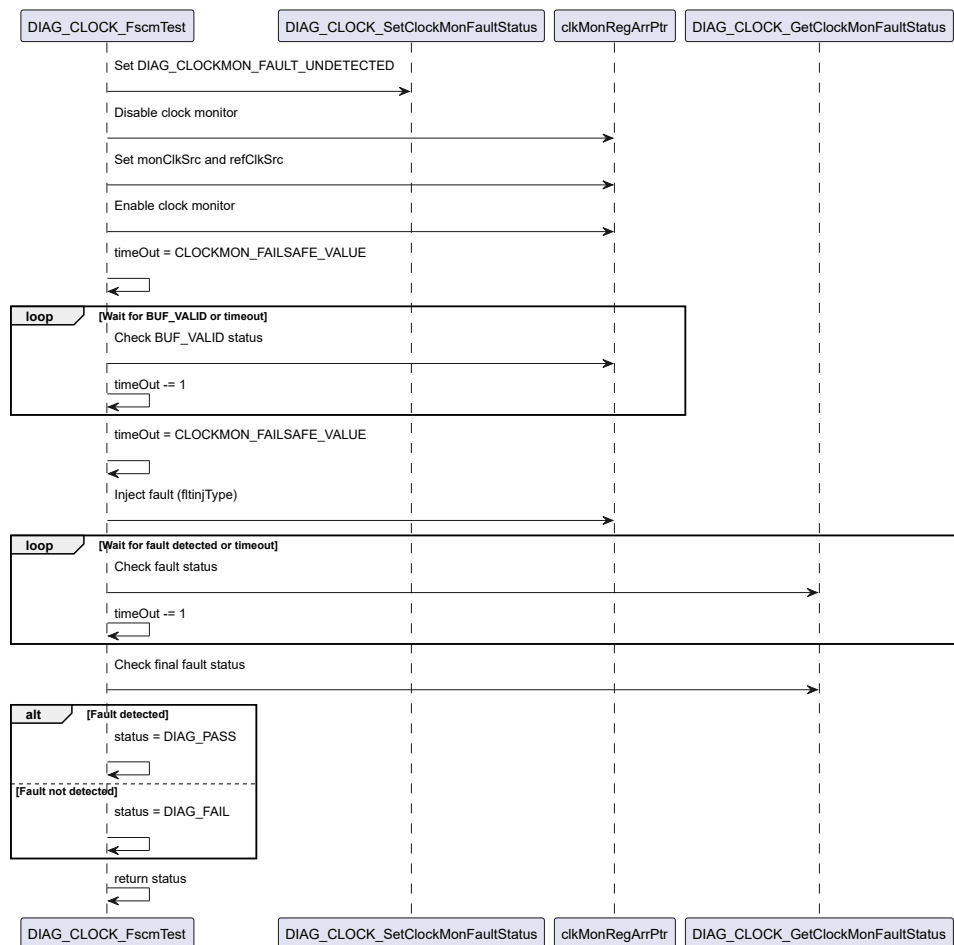
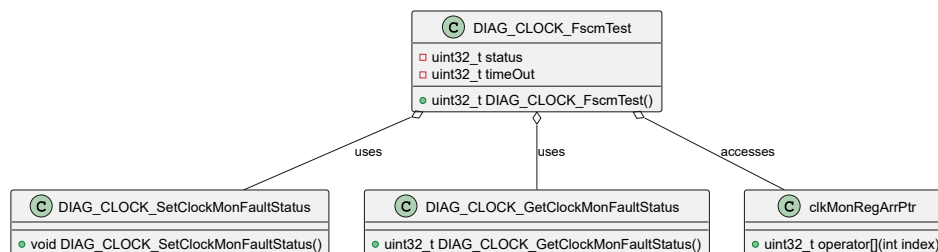


Figure 8-2. Clock Fail Safe Clock Monitor Class diagram



8.3. Macros

8.3.1. Definition Documentation

8.3.1.1. DIAG_CLOCKMON_FAULT_UNDETECTED

```
#define DIAG_CLOCKMON_FAULT_UNDETECTED 0x55U
```

Indicates that no fault has been detected by the clock monitor.

This macro defines the value returned or set when the clock monitor has not detected any fault condition. Used as a status indicator in diagnostic routines.

8.3.1.2. DIAG_CLOCKMON_FAULT_DETECTED

#define DIAG_CLOCKMON_FAULT_DETECTED 0xAAU

Indicates that a fault has been detected by the clock monitor.

This macro defines the value returned or set when the clock monitor has detected a fault condition. Used as a status indicator in diagnostic routines.

8.4. Enums

8.4.1. Enumeration Type Documentation

8.4.1.1. DIAG_CLOCK_FLTINJ_TYPE

enum DIAG_CLOCK_FLTINJ_TYPE

Specifies the type of clock fault to inject during the diagnostic test.

This enumeration is used to select the nature of the fault that will be injected into the monitored clock to test the detection capability of the clock monitor.

8.4.1.2. DIAG_CLOCK_MON_REF_CLK

enum DIAG_CLOCK_MON_REF_CLK

Enumerates the possible clock sources for both monitored and reference clocks.

This enumeration is used to specify the clock source to be monitored or used as a reference in the clock monitor diagnostic test.

8.4.1.3. DIAG_CLOCK_MON_INST

enum DIAG_CLOCK_MON_INST

Identifies the available clock monitor hardware instances.

This enumeration is used to select which clock monitor instance is to be configured and tested by the diagnostic function.

8.5. Source Code Reference

The following table provides a list of APIs and their location in the source code:

MODULE	DIAGNOSTIC API	SOURCE CODE REFERENCE
CLOCK	- DIAG_CLOCK_SetClockMonFaultStatus - DIAG_CLOCK_FscmTest	diag_clock_fscmtest.c

9. CPU

9.1. Overview

This document describes the diagnostics API for the CPU module. The safety requirements described for CPU diagnostics in the Safety Manual are implemented in the following APIs.

- [DIAG_CPU_RegResetStateTest](#)
- [DIAG_CPU_ControlRegTest](#)
- [DIAG_CPU_SelfTest](#)

Constraints/Limitations:

- The [DIAG_CPU_RegResetStateTest](#) API does not test some registers (W0 - W15, DSRPAG, TBLPAG, PCL and PCH) and some specific control/status bits (DC, N, OV, SZ, C and SFA).
- The [DIAG_CPU_ControlRegTest](#) API does not test some registers (PCL, PCH, DSRPAG, DSWPAG, DOSTARTL, DOSTARTH, YPAG, MSTRPR and CTXTSTAT) and some specific control/status bits (OA, OB, OAB, DA, DC, RA, N, OV, Z, C, EDT, DL<2:0>, IPL<3>, SFA and BREN).

Integration Rules:

- The [DIAG_CPU_RegResetStateTest](#) API must be called by the application before initializing any peripherals, enabling any interrupts or performing any computations. If the function returns a result of DIAG_FAIL, then it is recommended for the application program to call the diagnostic function again to ensure that the correct default values have been retained in the registers.
- The [DIAG_CPU_ControlRegTest](#) API must be periodically called by the application, with the calling interval selected suitably to satisfy the Fault Tolerant Time Interval requirements of the application.
- The [DIAG_CPU_SelfTest](#) API must be periodically called by the application, with the calling interval selected suitably to satisfy the Fault Tolerant Time Interval requirements of the application.
- The [DIAG_CPU_SelfTest](#) API provides a wrapper function for Math Error Trap routine. The wrapper functions can be called inside the respective ISRs as shown below:

```
void __attribute__((interrupt,auto_psv)) _MathError (void)
{
    DIAG_CPU_CheckFlagMathError (userMathError);
}
```

9.2. Functions

9.2.1. Function Documentation

9.2.1.1. DIAG_CPU_RegResetStateTest()

uint32_t DIAG_CPU_RegResetStateTest (void)

Software Requirement Reference ID : SW_CPU_REGISTER_RESET_STATE_TEST_01 Tests the reset state of CPU registers against expected values.

This function verifies that a set of CPU registers are in their expected reset state by comparing their current values (masked with a reset mask) to the expected reset values. The function iterates through a lookup table containing the expected reset values and masks for each register. If a register does not match its expected reset value, the function increments a failure counter and attempts to restore the register to its expected reset state.

The function uses a pointer to a CPU register map to access the registers. After checking all registers, it sets a diagnostic return value to indicate pass or fail, depending on whether any mismatches were found. The result is then returned to the caller.

The function is typically used as part of a diagnostic or self-test routine to ensure that CPU registers are correctly reset after a system reset or during initialization.

Note:

The function assumes that the CPU register map and the reset value lookup table are correctly defined and accessible. It also assumes that the `DIAG_CPU_SetRetValue()` and `DIAG_CPU_GetRetValue()` functions are implemented to handle diagnostic results.

Returns:

uint32_t Returns the diagnostic result:

- DIAG_PASS if all registers are in their expected reset state.
- DIAG_FAIL if any register does not match its expected reset value.

Figure 9-1. CPU Reg Reset State Test Sequence Diagram

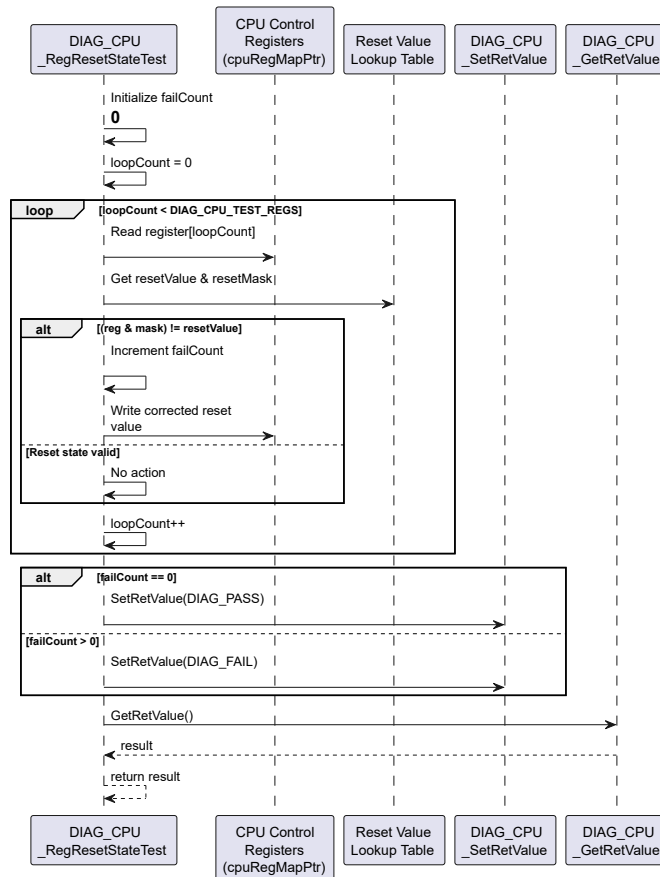
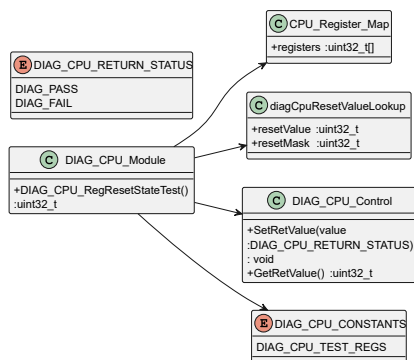


Figure 9-2. CPU Reg Reset State Test Class Diagram



9.2.1.2. DIAG_CPU_ControlRegTest()

uint32_t DIAG_CPU_ControlRegTest (void)

Software Requirement Reference ID : SW_CPU_CONTROL_REGISTER_TEST_01 Tests the functionality of CPU control registers by writing and verifying test patterns.

This function performs a comprehensive test of the CPU's control registers to ensure their correct operation. It uses a lookup table containing two test values and a mask for each register. For each register that is not read-only, the function:

- Saves the current register value.
- Writes the first test value and verifies that the register correctly stores and returns the value (masked).
- Writes the second test value and verifies it similarly.
- Restores the original register value after testing.
- If a register fails to store or return the expected value, a status flag is set for that register.

The function also saves and restores the interrupt control register (INTCON2) to ensure that interrupts are disabled during the test, preventing interference.

Registers that are read-only or not tested are skipped (indicated by zero test values in the lookup table). The function calls an auxiliary test for non-memory-mapped control registers as well.

At the end of the test, the function sets a diagnostic result (pass/fail) based on whether any register failed the test, and returns this result.

Note:

The function assumes that the CPU register map, control test lookup table, and related diagnostic functions/macros are correctly defined and accessible. It also assumes that the DIAG_CPU_SetRetValue() and DIAG_CPU_GetRetValue() functions are implemented to handle diagnostic results.

Returns:

uint32_t Returns the diagnostic result:

- DIAG_PASS if all tested registers pass the test.
- DIAG_FAIL if any tested register fails.

Figure 9-3. CPU Control Register Test Sequence Diagram

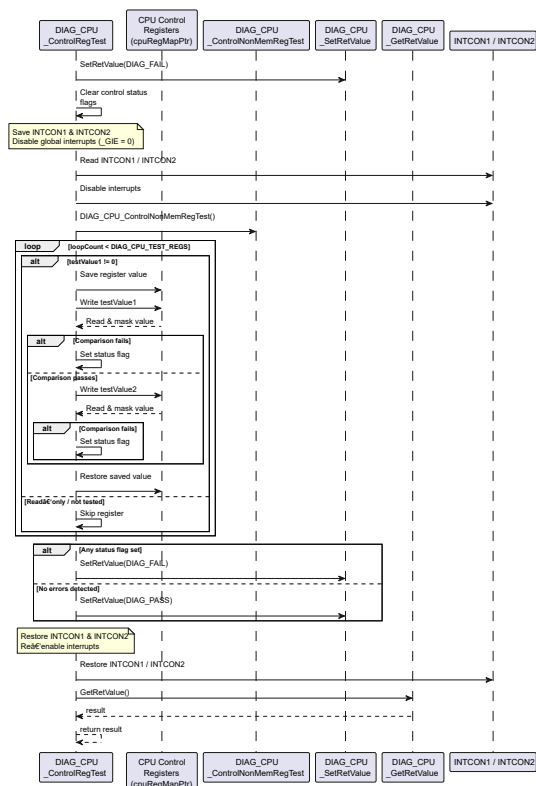
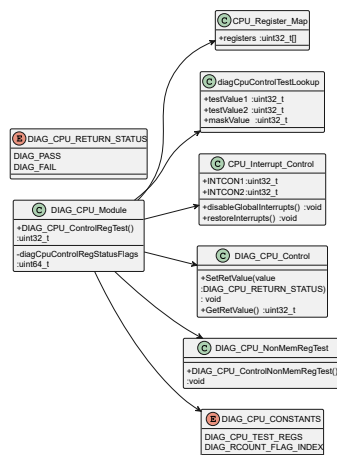


Figure 9-4. CPU Control Register Test Activity Diagram



9.2.1.3. DIAG_CPU_SelfTest()

uint32_t DIAG_CPU_SelfTest (DIAG_CPU_SELF_TEST_SUBSET executionMode)

Software Requirement Reference ID : SW_CPU_SELF_TEST_01Executes a comprehensive or subset-based CPU self-test routine.

This function performs a self-test of the CPU by executing a series of diagnostic test subsets, depending on the specified execution mode. It is designed to verify the integrity and correct operation of various CPU components, including memory-mapped and non-memory-mapped registers, as well as RAM regions used for diagnostics.

The function begins by saving the current interrupt control state and disabling global interrupts to ensure atomicity and prevent interference during the self-test. It then saves the contents of specific test RAM regions to a temporary array using inline assembly, preserving their state for later restoration.

The overall self-test algorithm is subdivided into eight Test Subset functions (each representing a different subset of CPU functionality). Each individual subset can be executed and this execution mode is labelled CPU_SELF_TEST_SUBSET with the suffix of the corresponding subset index. Alternately, all the Test Subset functions may be executed during a single API call. This execution mode is labeled CPU_SELF_TEST_ALL. The API allows the user to choose either of the above modes of execution through the choice of API function argument.

The self-test can be executed in three modes:

- **DIAG_CPU_SELF_TEST_ALL:** All available test subsets are executed sequentially. The pass/fail status of each subset is recorded, and the overall result is set to pass only if all subsets pass.
- **DIAG_CPU_SELF_TEST_SUBSET_INVALID:** If an invalid subset is specified, the function returns an invalid parameter result.
- **Specific Subset:** Only the specified test subset is executed, and its result is returned. The corresponding enum variables in the enum list are passed as argument. [DIAG_CPU_SELF_TEST_SUBSET](#) for the list of options

After the self-test execution, the function restores the original contents of the test RAM regions using inline assembly and re-enables interrupts by restoring the saved interrupt control state. The function then returns the overall diagnostic result.

The CPU instructions and features tested within each Test Subset function are listed below. (a) Test Subset 1: [DIAG_CPU_SELF_TEST_SUBSET](#) DIAG_CPU_SELF_TEST_SUBSET_1

- All of the MOV instructions are tested
- Addressing modes tested: Immediate, File Register, Register Direct and Indirect (with pre-increment, post-increment, pre-decrement, post-decrement, literal offset and register offset)
- SWAP and EXCH instructions are tested
- All bit manipulation, bit test and bit-compare-skip operations are tested
- Single-word and double-word instructions are tested
- Byte MOV instructions are also checked (b) Test Subset 2: [DIAG_CPU_SELF_TEST_SUBSET](#) DIAG_CPU_SELF_TEST_SUBSET_2
- PSV and table accesses from different sections (four locations) of program memory
- All bits of the program memory address bus are toggled
- All bits of the data memory address bus are toggled
- All bits of the X data read and write buses are toggled
- CPU registers tested for read/write operations are: W0-W15, SPLIM, TBLPAG, PSVPAG/DSRPAG
- Read-After-Write (RAW) dependency
- NOP and NOPR instructions (c) Test Subset 3: [DIAG_CPU_SELF_TEST_SUBSET](#) DIAG_CPU_SELF_TEST_SUBSET_3
- All conditional branch and GOTO instructions with alternative conditions
- Program Counter (PC) behavior during above program flow change operations
- All CALL and RETURN operations
- Automatic context save on stack
- All stack and shadow operations
- DISI instruction (d) Test Subset 4: [DIAG_CPU_SELF_TEST_SUBSET](#) DIAG_CPU_SELF_TEST_SUBSET_4

- Branch instructions tested for false condition are NOV, Z, NN, NC and NZ
- All logic instructions: AND, CLR, COM, IOR, NEG, SETM and XOR instructions
- All data rotate and shift instructions
- All compare and compare-skip instructions
- The Z, OV, DC, N and C bits of the STATUS Register are checked for their behavior
- Some Byte mode logic instructions are tested (e) Test Subset 5: [DIAG_CPU_SELF_TEST_SUBSET_DIAG_CPU_SELF_TEST_SUBSET_5](#)
- The following arithmetic instructions are tested: ADD, ADDC, SUB, SUBB, SUBBR, INC, INC2, DEC, DEC2, SE, ZE
- The addressing modes tested here are: Immediate, File Register, Register Direct
- Byte instructions are also checked
- Branch instructions tested for True condition are: GT, GTU, LE, LEU, NC, NN, OV
- Branch instructions tested for False condition are: C, GE, GEU, LT, LTU, N, NOV
- Divide Unsigned Double (DIV.UD) instruction (f) Test Subset 6: [DIAG_CPU_SELF_TEST_SUBSET_DIAG_CPU_SELF_TEST_SUBSET_6](#)
- All MUL instruction variants
- All DIV instruction variants except DIVF and DIV.UD
- REPEAT loop
- Math error trap generation (divide-by-zero error) (g) Test Subset 7: [DIAG_CPU_SELF_TEST_SUBSET_DIAG_CPU_SELF_TEST_SUBSET_7](#)
- All DSP accumulator operations, including different bit states of the individual bits of both accumulators
- All DSP multiplier-based instructions
- All DSP MAC Register Indirect Addressing modes
- All DSP shift instructions
- Math error trap generation due to accumulator-related events (accumulator overflow and catastrophic overflow)
- CORCON bit behavior (h) Test Subset 8: [DIAG_CPU_SELF_TEST_SUBSET_DIAG_CPU_SELF_TEST_SUBSET_8](#)
- Modulo Addressing (both Byte and Word modes)
- Bit-Reversed Addressing
- DO loop
- Fractional Divide (DIVF) instruction

Parameters:

in	executionMode	Specifies which self-test subset(s) to execute. This can be: • DIAG_CPU_SELF_TEST_ALL: Execute all test subsets. • DIAG_CPU_SELF_TEST_SUBSET_INVALID: Invalid subset (returns error). • Any valid subset index: Execute only the specified subset.
----	---------------	--

Returns:

uint32_t Returns the diagnostic result:

- DIAG_PASS if all executed test subsets pass.
- DIAG_FAIL if any executed test subset fails.
- DIAG_INVALID_PARAM if an invalid subset is specified.

Note:

The function uses inline assembly to save and restore RAM regions critical to the self-test. It also relies on external diagnostic functions and status variables to manage and report the results of each test subset. The function disables and restores interrupts to ensure test integrity.

Figure 9-5. CPU Self Test Sequence Diagram

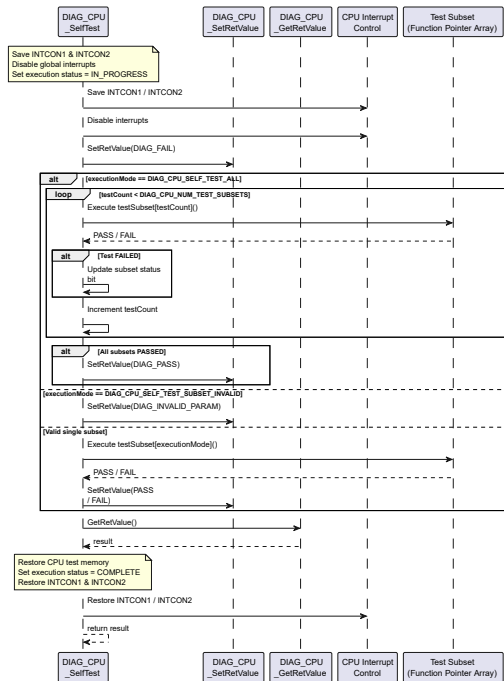
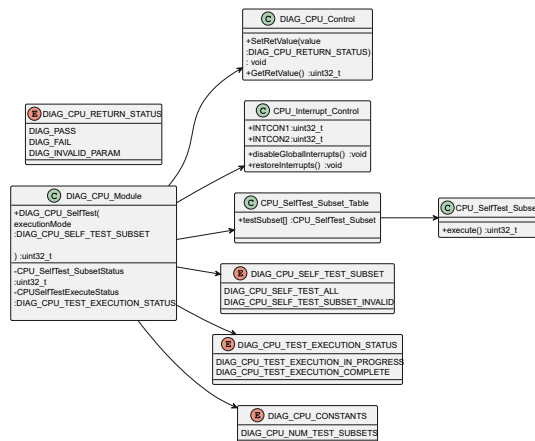


Figure 9-6. CPU Self Test Class Diagram



9.2.1.4. DIAG_CPU_SelfTest_GetStatus()

uint32_t DIAG_CPU_SelfTest_GetStatus (void)

Retrieves the most recent pass/fail status of CPU self-test subsets.

This function returns the status word that indicates the result of the most recently executed CPU self-test subsets. Each bit in the returned value corresponds to a specific test subset:

- A bit value of '1' indicates that the corresponding test subset has failed.
- A bit value of '0' indicates that the corresponding test subset has passed.

This function is typically used after running the CPU self-test routine to determine which (if any) test subsets have failed.

Returns:

uint32_t A status word where each bit represents the pass/fail status of a test subset:

- '1' = Fail
- '0' = Pass

Note:

The mapping of bits to test subsets is as follows. The Least significant bit indicates the subset0, the next bit subset1 and so on.

9.2.1.5. DIAG_CPU_CheckFlagMathError()

void DIAG_CPU_CheckFlagMathError (void(*)() userMathError)

Handles math error conditions during CPU self-test or delegates to user-defined handler.

This function checks the current status of the CPU self-test execution. If a math error occurs while the self-test is in progress, it calls the internal diagnostic math error handler (DIAG_CPU_MathError). If the self-test is not in progress, it calls a user-provided math error handler function, allowing the application to handle math errors according to its own logic.

The function is marked with __attribute__((optimize(0))) to disable compiler optimizations, ensuring predictable behavior for diagnostic and coverage analysis purposes.

Parameters:

in	userMathError	Pointer to a user-defined function that handles math errors when the self-test is not in progress. This function should take no arguments and return void.
----	---------------	--

Returns:

void

Note:

This function is typically used as a wrapper for math error handling, ensuring that errors are processed appropriately depending on whether the CPU self-test is running. If the self-test is running, only the internal handler is called to maintain test integrity. Otherwise, the user-defined handler is invoked.

9.2.1.6. FAULTINJ_CPU_RegisterResetStateCheck()

void FAULTINJ_CPU_RegisterResetStateCheck (uint32_t * faultInjAddress, uint16_t faultInjValue)

Software Requirement Reference ID : SW_CPU_REGISTER_RESET_STATE_CHECK_01.

This API injects the fault for the CPU_REGISTER_RESET_STATE_CHECK diagnostic. Since this diagnostic does not perform any writes to the control/status register being tested (unless it is found to contain an incorrect reset state), this fault-injection API function simply writes an incorrect default value to a specific register, thereby causing the diagnostic API to return a failure for that register. DMA and interrupts are not required for this fault-injection method.

Parameters:

faultInjAddress	Pointer to the register in which the fault is to be injected.
faultInjValue	The faulty value written to this register.

Returns:

void

9.3. Macros

9.3.1. Definition Documentation

9.3.1.1. DIAG_CPU_TEST_EXECUTION_IN_PROGRESS

```
#define DIAG_CPU_TEST_EXECUTION_IN_PROGRESS 0x5555U
```

Macro indicating that a CPU self-test is currently in progress.

This macro defines a constant value (0x5555U) used to represent the state where a CPU self-test routine is actively executing. It can be used to check or set the execution status in diagnostic routines.

9.3.1.2. DIAG_CPU_TEST_EXECUTION_COMPLETE

```
#define DIAG_CPU_TEST_EXECUTION_COMPLETE 0xAAAAU
```

Macro indicating that a CPU self-test has completed.

This macro defines a constant value (0xAAAAU) used to represent the state where a CPU self-test routine has finished execution. It can be used to check or set the execution status in diagnostic routines.

9.4. Enums

9.4.1. Enumeration Type Documentation

9.4.1.1. DIAG_CPU_CONTROL_REGISTER_TEST_FLAG_INDEX

```
enum DIAG_CPU_CONTROL_REGISTER_TEST_FLAG_INDEX
```

Enumeration of flag indices for CPU control register diagnostic tests.

This enumeration defines symbolic indices for various CPU control and status registers used in diagnostic test routines. Each enumerator corresponds to a specific CPU register or special function register, and is typically used to index into status arrays or bitfields that track the pass/fail status of individual register tests.

The indices cover general-purpose working registers (W0-W15), accumulator registers, special function registers, and various control/status registers relevant to CPU diagnostics.

9.4.1.2. DIAG_CPU_SELF_TEST_SUBSET

```
enum DIAG_CPU_SELF_TEST_SUBSET
```

Enumeration of CPU self-test subset identifiers.

This enumeration defines the possible subsets of CPU self-tests that can be executed individually or collectively. Each enumerator represents a specific group of diagnostic tests targeting different CPU components or functionalities. The enumeration also includes options to execute all subsets or to indicate an invalid subset selection.

9.5. Global Variables

9.5.1. Variable Documentation

9.5.1.1. DIAG_CPU_TestResult

volatile uint32_t DIAG_CPU_TestResult[extern]

Global variable holding the result of the most recent CPU diagnostic test.

This external variable is used to store the result of the latest CPU diagnostic operation. It can be accessed by multiple sub APIs to check the outcome of CPU self-tests or other diagnostic routines.

Note:

The values stored in this variable (e.g., DIAG_PASS, DIAG_FAIL) are defined in

See also:

diag_cpu_self_test.c.

9.5.1.2. DIAG_CPU_MathErrorFlag

volatile uint16_t DIAG_CPU_MathErrorFlag[extern]

Global flag indicating the occurrence of a CPU math error.

This external, volatile variable is set when a math error (such as overflow, underflow, or divide-by-zero) is detected during CPU operation or self-test routines. It can be checked by diagnostic or application code to determine if a math error has occurred.

Note:

The variable is declared as volatile because it may be modified by interrupt service routines or hardware, and should not be optimized out by the compiler.

9.5.1.3. diagCpuControlRegStatusFlags

volatile uint64_t diagCpuControlRegStatusFlags[extern]

Status flags for CPU control register diagnostic tests.

This external, volatile 64-bit variable is used as a bitfield to record the pass/fail status of individual CPU control register tests. Each bit corresponds to a specific register or test, where a '1' typically indicates a failure and a '0' indicates a pass.

Note:

The mapping of bits to specific registers or tests is defined elsewhere in the codebase. The variable is declared as volatile because it may be updated by different parts of the program, including interrupt service routines.

9.6. Source Code Reference

The following table provides a list of APIs and their location in the source code:

MODULE	DIAGNOSTIC API	SOURCE CODE REFERENCE
CPU	• DIAG_CPU_ControlRegTest	diag_cpu_controlreg_test.c
	• DIAG_CPU_ControlNonMemRegTest	
CPU	• DIAG_CPU_RegResetStateTest	diag_cpu_regresetstate_test.c
CPU	• DIAG_CPU_SelfTest	diag_cpu_self_test.c
	• DIAG_CPU_SelfTest_GetStatus	
CPU	• FAULTINJ_CPU_RegisterResetStateCheck	faultinj_cpu.c

10. CRC

10.1. Overview

This document describes the diagnostics API for the CRC module. The safety requirements described for CRC diagnostics in the Safety Manual are implemented in the following APIs.

- [DIAG_CRC_FunctionalTest](#)
- [DIAG_CRC_FunctionalTest_GetStatus](#)

Constraints/Limitations:

- Nil

Integration Rules:

- The [DIAG_CRC_FunctionalTest](#) API must be called on start-up.
- The [DIAG_CRC_FunctionalTest_GetStatus](#) API can be called by the application after executing the [DIAG_CRC_FunctionalTest](#) API function.

10.2. Functions

10.2.1. Function Documentation

10.2.1.1. DIAG_CRC_FunctionalTest()

uint32_t DIAG_CRC_FunctionalTest (void)

Software Requirement Reference ID : SW_CRC_FUNCTIONAL_TEST_01 This diagnostic API performs CRC hardware functional test by checking the CRC results for each CRC type starting from polynomial width of 1 till 32 in a loop. For each polynomial width, two polynomials are used one starting from zero and one starting with one. For example: for CRC-8 polynomial1 = 0x55 and polynomial2 = 0xAA. For each of the polynomial, CRC is computed and compared against the corresponding expected value. The return value is a PASS/FAIL where pass indicates that all the CRC types have passed the test while fail indicates that one or more have failed the test. On diagnostic fail, [DIAG_CRC_FunctionalTest_GetStatus\(\)](#) API may be used to check which CRC test has failed during the most recent execution.

Parameters:

void

Returns:

DIAG_PASS
DIAG_FAIL

Code Snippet:

```
volatile uint32_t DIAG_CRC_retValue = DIAG_FAIL;  
  
DIAG_CRC_retValue = DIAG_CRC_FunctionalTest();
```

Figure 10-1. CRC Functional Test Sequence Diagram

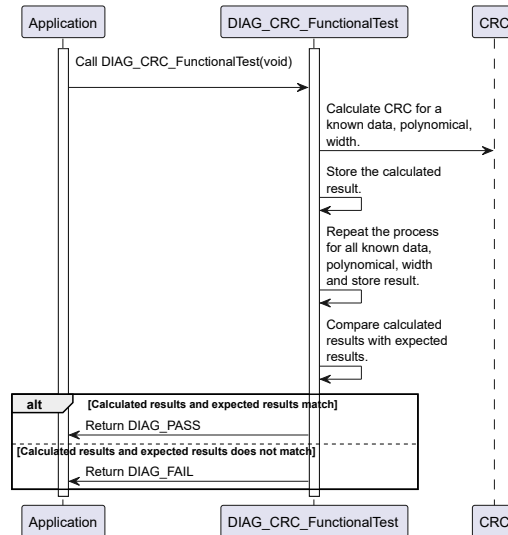
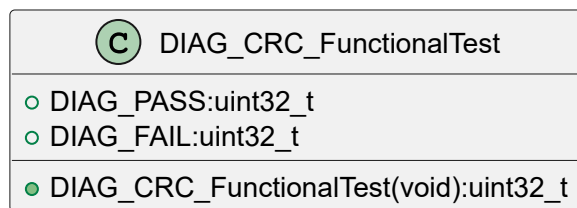


Figure 10-2. CRC Functional Test Class Diagram



10.2.1.2. DIAG_CRC_FunctionalTest_GetStatus()

uint32_t DIAG_CRC_FunctionalTest_GetStatus (void)

Software Requirement Reference ID : SW_CRC_FUNCTIONAL_TEST_01 This API function returns a 32-bit unsigned integer which indicates which CRC test has failed during the most recent execution. Refer [DIAG_CRC_FunctionalTest\(\)](#).

Parameters:

void

Returns:

Variable containing individual PASS/FAIL flags for all the CRC tests.

```

volatile uint32_t DIAG_CRC_retValue = DIAG_FAIL;
volatile uint32_t DIAG_CRC_FunctionalTest_StatusFlag = 0x00000000;

DIAG_CRC_retValue = DIAG_CRC_FunctionalTest();
if (DIAG_CRC_retValue == DIAG_FAIL)
{
    DIAG_CRC_FunctionalTest_StatusFlag = DIAG_CRC_FunctionalTest_GetStatus();
}
  
```

10.2.1.3. FAULTINJ_CRC_SetFaultInjFunctional()

void FAULTINJ_CRC_SetFaultInjFunctional (DIAG_CRC_FAULTINJ_SCENARIOS setFaultInj)

Software Requirement Reference ID : SW_CRC_FUNCTIONAL_TEST_01 Implements the fault injection for functional diagnostics.

Parameters:

setFaultInj	- Fault mode
-------------	--------------

Returns:

void

10.3. Macros

10.3.1. Definition Documentation

10.3.1.1. DIAG_CRC_DATAWIDTH

```
#define DIAG_CRC_DATAWIDTH 8U
```

Specifies the width of data used

10.3.1.2. DIAG_CRC_DATALENGTH

```
#define DIAG_CRC_DATALENGTH 8U
```

Specifies the length of data or number of bytes of data used

10.3.1.3. DIAG_CRC_DATA_ARRAY_SIZE

```
#define DIAG_CRC_DATA_ARRAY_SIZE 8U
```

Specifies the size of data array

10.3.1.4. DIAG_CRC_NUM_OF_POLYNOMIALS

```
#define DIAG_CRC_NUM_OF_POLYNOMIALS 33U
```

Specifies the loop count for CRC diagnostics

10.3.1.5. DIAG_CRC_STRUCT_SIZE

```
#define DIAG_CRC_STRUCT_SIZE 32U
```

Specifies the size of structure array for defining polynomials and expected values

10.4. Enums

10.4.1. Enumeration Type Documentation

10.4.1.1. DIAG_CRC_FAULTINJ_SCENARIOS

```
enum DIAG_CRC_FAULTINJ_SCENARIOS
```

Defines the fault injection scenarios for crc diagnostics.

10.5. Data Structure Documentation

10.5.1. DIAG_CRC_POLY_EXPECTED_VALUES Struct Reference

10.5.1.1. Detailed Description

This structure defines the polynomials and the corresponding expected values for each polynomial width from 1-32

10.5.1.1.1. Data Fields

- uint32_t [poly](#)
- uint64_t [expectedVal1](#)
- uint64_t [expectedVal2](#)

10.5.1.2. Field Documentation

10.5.1.2.1. poly

DIAG_CRC_POLY_EXPECTED_VALUES::poly

Polynomial value

10.5.1.2.2. expectedVal1

DIAG_CRC_POLY_EXPECTED_VALUES::expectedVal1

Expected CRC value of first polynomial

10.5.1.2.3. expectedVal2

DIAG_CRC_POLY_EXPECTED_VALUES::expectedVal2

Expected CRC value of second polynomial (second polynomial obtained by ~polynomial1)

10.6. Source Code Reference

The following table provides a list of APIs and their location in the source code:

MODULE	DIAGNOSTIC API	SOURCE CODE REFERENCE
CRC	• DIAG_CRC_FunctionalTest_GetStatus	diag_crc_functional_test.c
	• DIAG_CRC_FunctionalTest	
CRC	• FAULTINJ_CRC_SetFaultInjFunctional	faultinj_crc.c

11. FLASH

11.1. Overview

This document describes the diagnostics for the FLASH module. The safety requirements described for Flash diagnostics in the Safety Manual are implemented in the following APIs.

- [DIAG_FLASH_SingleDoubleErrorDetectionTest](#)
- [DIAG_FLASH_IntegrityReadPractice](#)
- [DIAG_FLASH_WriteVerifyPractice](#)
- [DIAG_FLASH_CRCCalculate](#)
- [DIAG_FLASH_CRCPractice](#)

Constraints/Limitations:

- Nil

Integration Rules:

- Nil

11.2. Functions

11.2.1. Function Documentation

11.2.1.1. **DIAG_FLASH_SingleDoubleErrorDetectionTest()**

uint32_t DIAG_FLASH_SingleDoubleErrorDetectionTest (uint32_t * location)

Software Requirement Reference ID :

SW_FLASH_ECC_SINGLE_DOUBLE_ERROR_DETECTION_TEST_01 The diagnostic API is aimed to verify the functionality of the hardware detecting a single or a double-bit error. The faults are induced using the ECC fault mechanics. On successful detection the diagnostics returns a pass otherwise fail.

Parameters:

*location	- Pointer to the Flash location
-----------	---------------------------------

Returns:

DIAG_PASS
DIAG_FAIL
DIAG_INVALID_PARAM

Figure 11-1. FLASH Single Double Error Detection Test

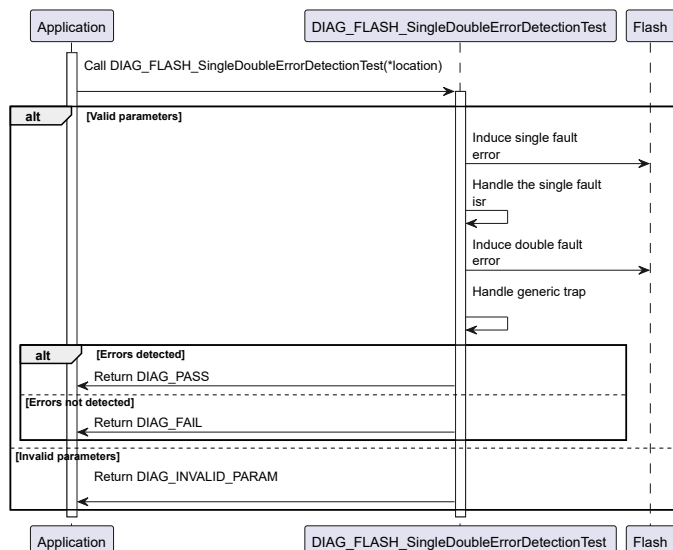
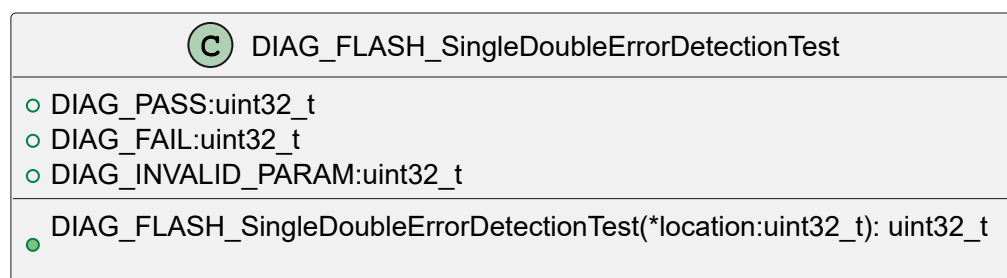


Figure 11-2. FLASH Single Double Error Detection Test Class Diagram



11.2.1.2. DIAG_FLASH_IntegrityReadPractice()

uint32_t DIAG_FLASH_IntegrityReadPractice (uint32_t * location)

Software Requirement Reference ID : SW_FLASH_INTEGRITY_READ_PRACTICE_01 The diagnostic API is aimed to verify the data integrity. The flash location which is of interest is accessed using a read operation to check for data integrity. If any errors are detected by the ECC the diagnostics returns a fail and if no errors detected the diagnostics returns a pass.

Parameters:

*location	- Pointer to the Flash location
-----------	---------------------------------

Returns:

DIAG_PASS
DIAG_FAIL
DIAG_INVALID_PARAM

Figure 11-3. FLASH Integrity Read Practice

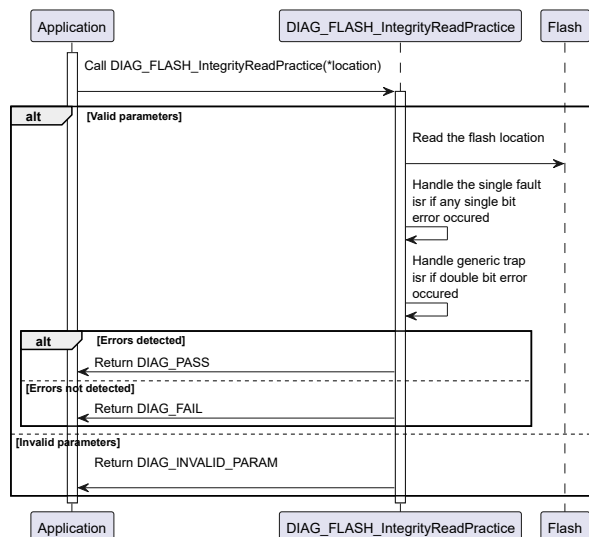
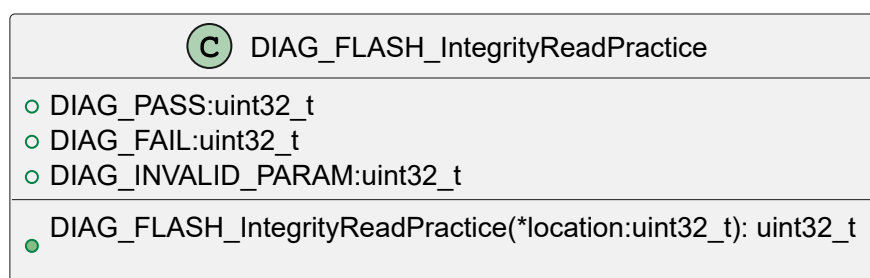


Figure 11-4. FLASH Integrity Read Practice Class Diagram



11.2.1.3. DIAG_FLASH_WriteVerifyPractice()

uint32_t DIAG_FLASH_WriteVerifyPractice (uint32_t * location, DIAG_FLASH_DATA_WRITE dataToAuthenticate)

Software Requirement Reference ID : SW_FLASH_WRITE_VERIFY_PRACTICE_01 The diagnostic API is aimed to verify the data authenticity. The flash location which is of interest is accessed using a read operation and compared with the user data for authenticity. The diagnostics return pass on data match otherwise returns fail.

Parameters:

*location	- Pointer to the Flash location
dataToAuthenticate	- Data for comparison

Returns:

DIAG_PASS
DIAG_FAIL
DIAG_INVALID_PARAM

Figure 11-5. FLASH Write Verify Practice

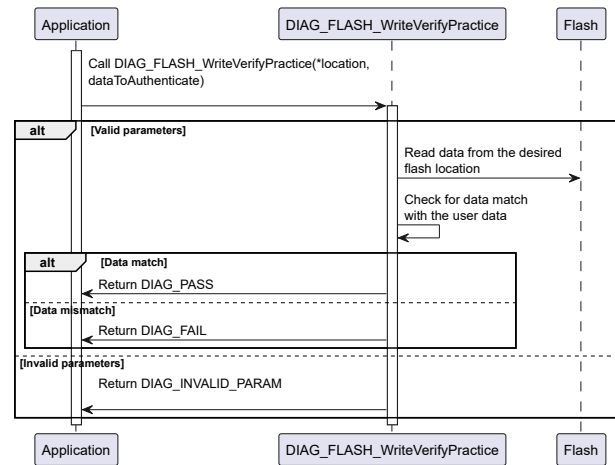
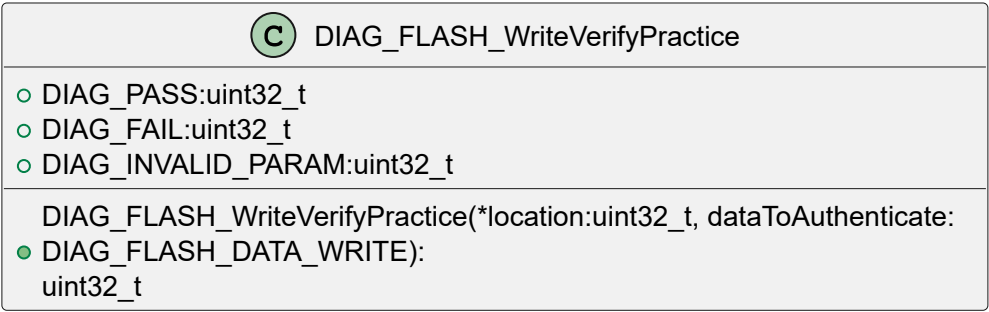


Figure 11-6. FLASH Write Verify Practice Class Diagram



11.2.1.4. DIAG_FLASH_CRCCalculate()

uint32_t DIAG_FLASH_CRCCalculate (uint32_t * flashBlockStart, uint32_t * flashBlockEnd)

Software Requirement Reference ID : SW_FLASH_MEMORY_CRC_PRACTICE_01The API calculates the CRC value for the flash block using the NVM CRC.

Parameters:

*flashBlockStart	- Pointer to starting address of the flash block for testing
*flashBlockEnd	- Pointer to end address of the flash block for testing

Returns:

Calculated CRC for the flash block

11.2.1.5. DIAG_FLASH_CRCPractice()

uint32_t DIAG_FLASH_CRCPractice (uint32_t * flashBlockStart, uint32_t * flashBlockEnd, uint32_t previousCRC)

Software Requirement Reference ID : SW_FLASH_MEMORY_CRC_PRACTICE_01 The diagnostic API calculates the CRC value for the flash block using the NVM CRC and compares with previous calculated CRC to check for authenticity. The diagnostics shall return pass on success full authentication otherwise returns fail.

Parameters:

*flashBlockStart	- Pointer to starting address of the flash block for testing
------------------	--

*flashBlockEnd	- Pointer to end address of the flash block for testing
previousCRC	- Previously calculated CRC value for the same block

Returns:

DIAG_PASS
DIAG_FAIL
DIAG_INVALID_PARAM

Figure 11-7. FLASH CRC Practice

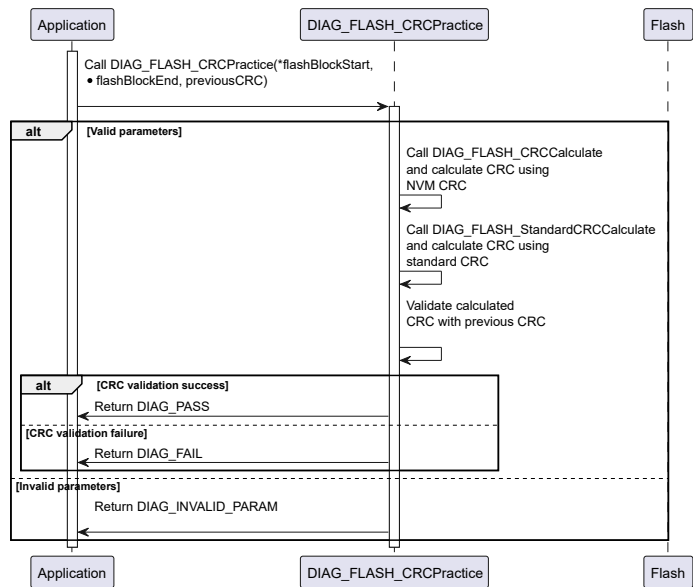
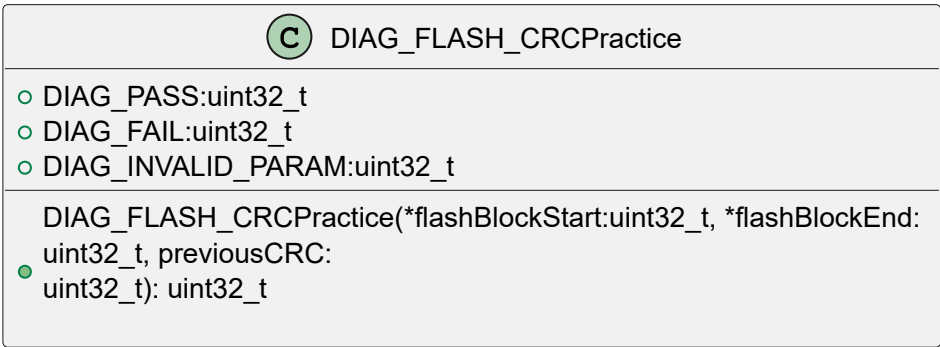


Figure 11-8. FLASH CRC Practice Class Diagram



11.2.1.6. FAULTINJ_FLASH_SetTestModeBitError()

void FAULTINJ_FLASH_SetTestModeBitError (DIAG_FLASH_FAULTINJ_SCENARIOS setFaultInj)

Software Requirement Reference ID :

SW_FLASH_ECC_SINGLE_DOUBLE_ERROR_DETECTION_TEST_01Implements the fault injection for flash ecc single double error detection test diagnostics.

Parameters:

setFaultInj	- Fault mode
-------------	--------------

Returns:

void

11.2.1.7. FAULTINJ_FLASH_SetTestModeCrcError()

void FAULTINJ_FLASH_SetTestModeCrcError (DIAG_FLASH_FAULTINJ_SCENARIOS setFaultInj)

Software Requirement Reference ID : SW_FLASH_MEMORY_CRC_PRACTICE_01 Implements the fault injection for flash memory crc practice diagnostics.

Parameters:

setFaultInj - Fault mode

Returns:

void

11.3. Enums**11.3.1. Enumeration Type Documentation****11.3.1.1. DIAG_FLASH_FAULTINJ_SCENARIOS**

enum DIAG_FLASH_FAULTINJ_SCENARIOS

Defines the fault injection scenarios for flash diagnostics.

11.4. Data Structure Documentation**11.4.1. DIAG_FLASH_DATA_WRITE Struct Reference****11.4.1.1. Detailed Description**

Maintains the necessary structure to handle flash write and read

11.4.1.1.1. Data Fields

- uint32_t [data](#) [4]

11.4.1.2. Field Documentation**11.4.1.2.1. data**

DIAG_FLASH_DATA_WRITE::data

Array to handle flash word data

11.5. Source Code Reference

The following table provides a list of APIs and their location in the source code:

MODULE	DIAGNOSTIC API	SOURCE CODE REFERENCE
FLASH	• DIAG_FLASH_SetTestModeBitError • DIAG_FLASH_SingleDoubleErrorDetectionTest • DIAG_FLASH_DoubleErrorDetectionTest • DIAG_FLASH_SingleErrorDetectionTest	diag_flash_biterror_test.c

Source Code Reference (continued)

MODULE	DIAGNOSTIC API	SOURCE CODE REFERENCE
FLASH	• DIAG_FLASH_STD_CRC32_POLYNOMIAL • DIAG_FLASH_CRCPractice • DIAG_FLASH_StandardCRCCalculate • DIAG_FLASH_CRCCalculate • DIAG_FLASH_ReverseBitOrder • DIAG_FLASH_SetTestModeCrcError • DIAG_FLASH_STD_CRC32_SEED_VALUE	diag_flash_crc_test.c
FLASH	• DIAG_FLASH_IntegrityReadPractice	diag_flash_integrity_practice.c
FLASH	• DIAG_FLASH_WriteVerifyPractice	diag_flash_writeverify_test.c
FLASH	• FAULTINJ_FLASH_SetTestModeCrcError • FAULTINJ_FLASH_SetTestModeBitError	faultinj_flash.c

12. GPIO

12.1. Overview

This document describes the diagnostics for the GPIO module. The safety requirements described for GPIO diagnostics in the Safety Manual are implemented by the following APIs.

- [DIAG_GPIO_ActivityCheck](#)
- [DIAG_GPIO_InputPractice](#)
- [DIAG_GPIO_InterruptGenTest](#)
- [DIAG_GPIO_OutputTest](#)
- [DIAG_GPIO_PpsOutputConnectionTest](#)
- [DIAG_GPIO_IntegrityMonitorTest](#)

Constraints/Limitations:

- Based on the family of devices and the pin count, the number and placement of the GPIOs may differ. The user would need this information and configure a set of pins suitably before running the diagnostics.
- The first five GPIO diagnostics listed above are Hardware connection based test and therefore requires physical connections on the Hardware which is under test as per the corresponding hardware requirements.

Integration Rules:

- To have a different set of pins to be tested by these diagnostics, the relevant pin configurations and connections are to be made and the tests have to be run again.
- The GPIO module diagnostics uses the delay routines defined in `diag_delay.c`. The users have to pass the system clock frequency value to the API `DIAG_SetClockFreq()` and call this API in their main application. If the API is not called, then this may result in improper functioning of the GPIO module diagnostics.

12.2. Functions

12.2.1. Function Documentation

12.2.1.1. DIAG_GPIO_InputPractice()

`uint32_t DIAG_GPIO_InputPractice (uint32_t * prlInputPort, uint16_t prlInputPin, uint32_t * secInputPort, uint16_t secInputPin)`

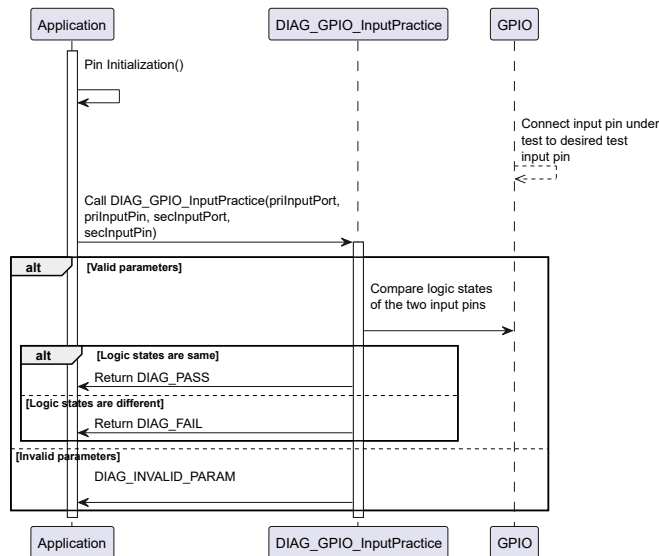
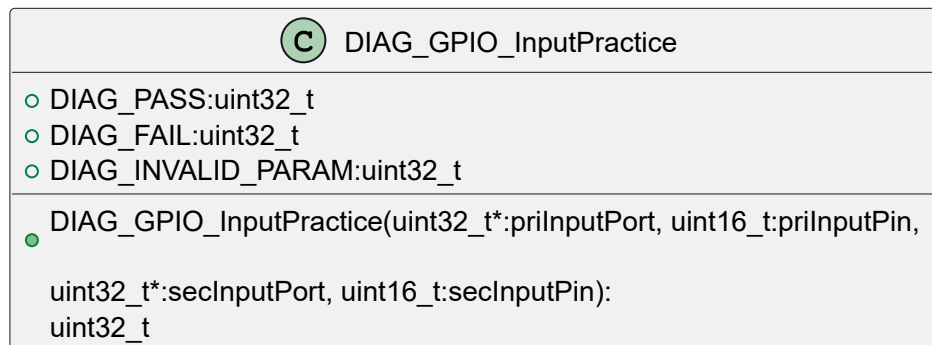
Software Requirement Reference ID : SW_GPIO_PORTS_INPUT_PRACTICE_01 External connections between two input configured pins ensures that both of them receive the same input. This diagnostic compares the state of these two pins (PORTx bit) and reports the result. The Hardware requirement HW_IO_INPUT_SECONDARY_IO_INPUT has to be met for this diagnostic to function.

Parameters:

<code>prlInputPort</code>	- The PORT register to which the primary input pin belongs to
<code>prlInputPin</code>	- Pin number on the <code>prlInputPort</code> that is physically accessed
<code>secInputPort</code>	- The PORT register to which the secondary input pin belongs to
<code>secInputPin</code>	- Pin number on the <code>secInputPort</code> that is physically accessed

Returns:

DIAG_PASS/DIAG_FAIL/DIAG_INVALID_PARAM

Figure 12-1. GPIO Input Practice Sequence diagram**Figure 12-2.** GPIO Input Practice Class diagram**12.2.1.2. DIAG_GPIO_OutputTest()**

uint32_t DIAG_GPIO_OutputTest (uint32_t * outputPort, uint16_t outputPinNum, uint32_t * inputPort, uint16_t inputPinNum)

Software Requirement Reference ID : SW_GPIO_PORTS_OUTPUT_TEST_01 Implements the comparison of the LATx and PORTx bits of the ports and reports the result of this comparison. The Hardware requirement HW_IO_OUTPUT_IO_INPUT has to be met for this diagnostic to function.

Parameters:

outputPort	- The PORT register to which the output pin belongs to
outputPinNum	- Pin number on the outputPort that is physically accessed
inputPort	- The PORT register to which the input pin belongs to
inputPinNum	- Pin number on the inputPort that is physically connected to outputPinNum

Returns:

DIAG_PASS/DIAG_FAIL/DIAG_INVALID_PARAM

Figure 12-3. GPIO Output Test Sequence diagram

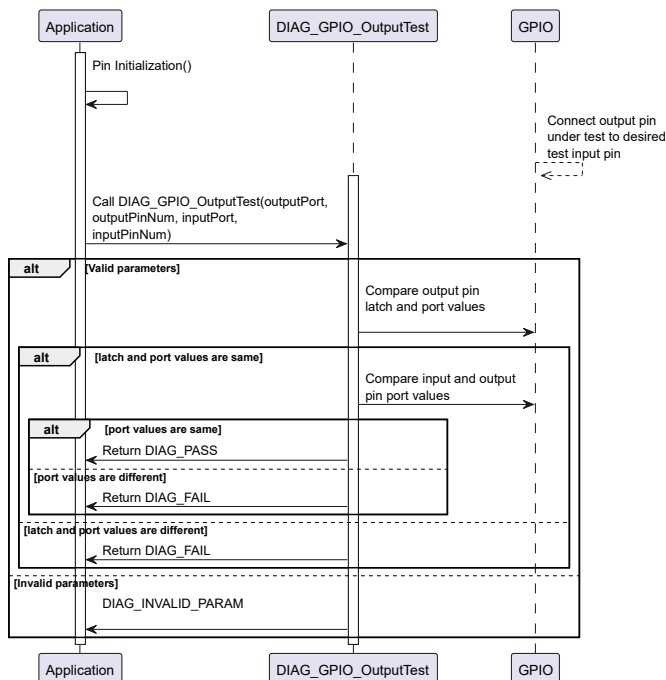
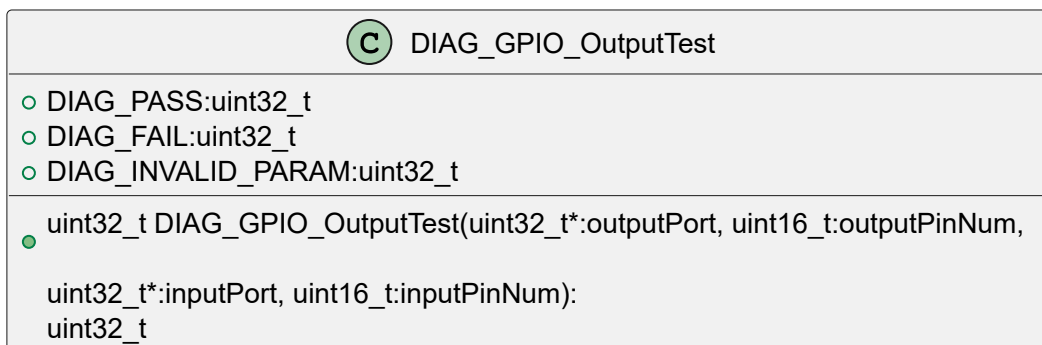


Figure 12-4. GPIO Output Test Class diagram



12.2.1.3. DIAG_GPIO_ActivityCheck()

uint32_t DIAG_GPIO_ActivityCheck (uint32_t * inputPort, uint16_t inputPinNum, void(*)() outputActivityFunc, uint16_t timeout)

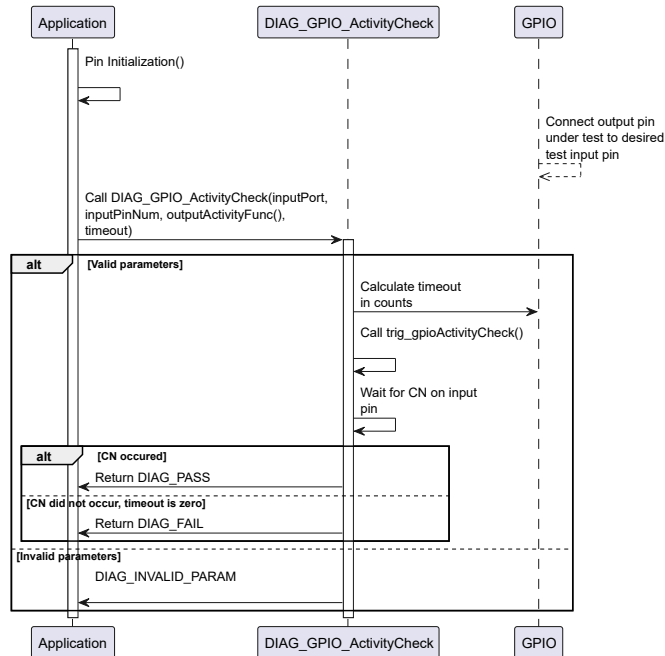
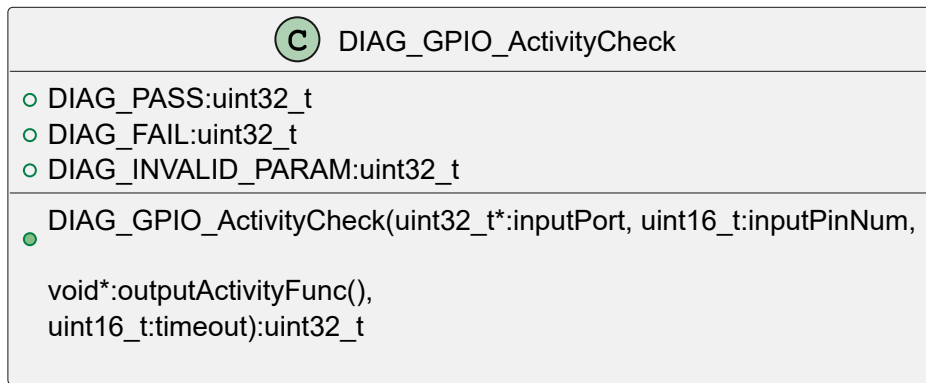
Software Requirement Reference ID : SW_GPIO_ACTIVITY_CHECK_01The diagnostic API shall accept the port and pin number of the I/O output pin to be tested and a reference I/O input pin. The activity on the output pin shall be connected to the input pin externally.

Parameters:

inputPort	- The PORT register to which the input pin belongs to
inputPinNum	- Pin number on the inputPort that is physically connected to the output pin under test.
outputActivityFunc()	- Function pointer, where user shall handle the output pin activity
timeout	- the diagnostic shall wait for timeout amount of time for the CN flag to be set, this is in terms microseconds.

Returns:

DIAG_PASS/DIAG_FAIL/DIAG_INVALID_PARAM

Figure 12-5. GPIO Activity Check Sequence diagram**Figure 12-6.** GPIO Activity Check Class diagram**12.2.1.4. DIAG_GPIO_InterruptGenTest()**

uint32_t DIAG_GPIO_InterruptGenTest (uint32_t * inputPort, uint16_t inputPinNum, uint32_t * outputPort, uint16_t outputPinNum)

Software Requirement Reference ID : SW_GPIO_PORTS_INTERRUPT_GENERATION_TEST_01 The diagnostic API shall accept the port address and pin number of input pin to be tested and a reference output pin for testing. The two pins shall be connected externally. Any transition on the output pin should generate a change notification ISR on the input pin.

Parameters:

inputPort	- The PORT register to which the input pin belongs to
inputPinNum	- Pin number on the inputPort that is physically connected to outputPinNum

outputPort	- The PORT register to which the output pin belongs to
outputPinNum	- Pin number on the outputPort that is physically accessed

Returns:

DIAG_PASS/DIAG_FAIL/DIAG_INVALID_PARAM

Figure 12-7. GPIO Interrupt Generation Test Sequence diagram

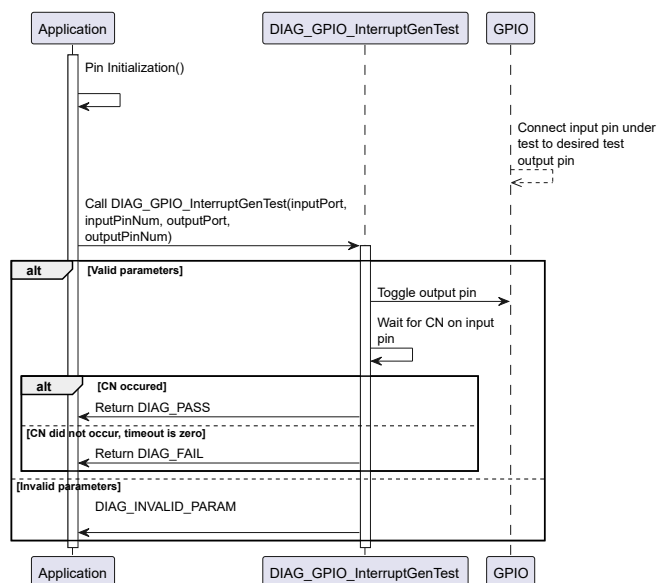
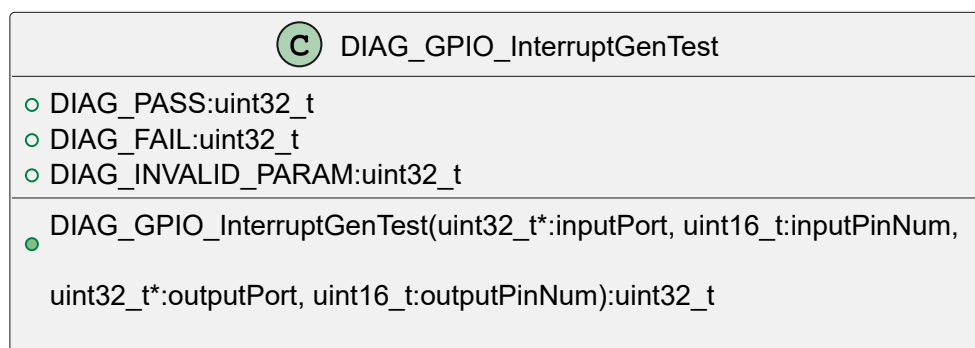


Figure 12-8. GPIO Interrupt Generation Test Class diagram



12.2.1.5. DIAG_GPIO_PpsOutputConnectionTest()

uint32_t DIAG_GPIO_PpsOutputConnectionTest (uint32_t * outputPort, uint16_t outputPinNum, uint32_t * inputPort, uint16_t inputPinNum, void(*)() outputPpsFunc, uint16_t timeout)

Software Requirement Reference ID : SW_GPIO_PPS_OUTPUT_CONNECTION_TEST_01The diagnostic shall require the user to map the output pin under test to the peripheral of interest. The output pin under test shall also be connected to an input test pin. Any activity through the peripheral should be reflected on the input test pin to verify the PPS of the output pin under test.

Parameters:

outputPort	- The PORT register to which the output pin belongs to
outputPinNum	- Pin number on the outputPort that is physically accessed

inputPort	- The PORT register to which the input pin belongs to
inputPinNum	- Pin number on the inputPort that is physically connected to outputPinNum
outputPpsFunc()	- Function pointer that shall trigger peripheral events on the pin
timeout	- the diagnostic shall wait for timeout amount of time before comparing the logic states, this is in terms microseconds

Returns:

DIAG_PASS/DIAG_FAIL/DIAG_INVALID_PARAM

Figure 12-9. GPIO PPS Output Connection Test Sequence diagram

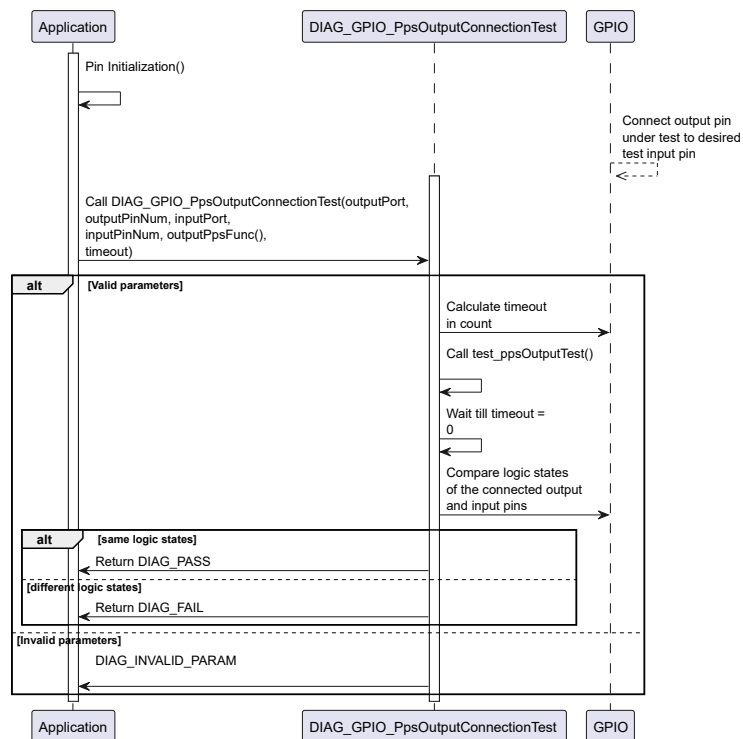
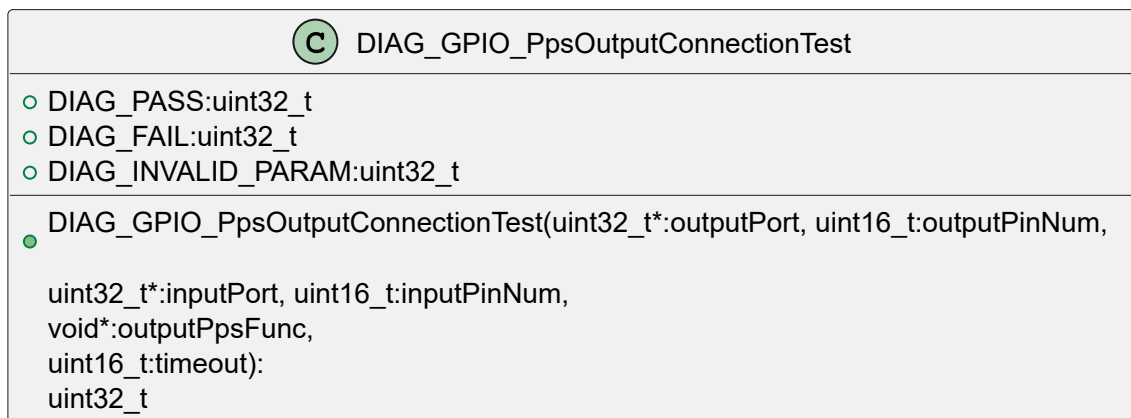


Figure 12-10. GPIO PPS Output Connection Test Class diagram



12.2.1.6. DIAG_GPIO_IntegrityMonitorTest()

uint32_t DIAG_GPIO_IntegrityMonitorTest (void)

Software Requirement Reference ID : SW_IO_MONITOR_TEST_01The diagnostic shall check the functionality of the module by using the OKINJ and FLTINJ, self-test feature of IOIM module.

Returns:

DIAG_PASS/DIAG_FAIL

Figure 12-11. GPIO Integrity Monitor Sequence diagram

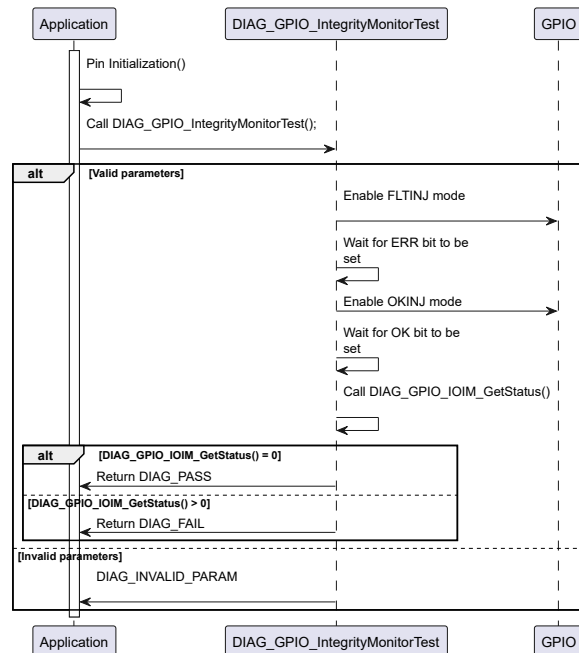
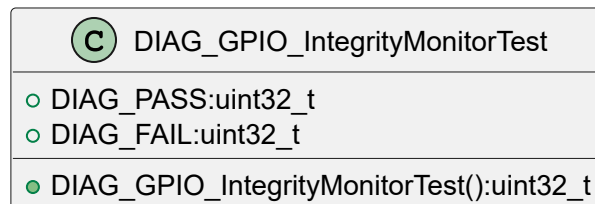


Figure 12-12. GPIO Integrity Monitor Class diagram



12.2.1.7. DIAG_GPIO_IOIM_GetStatus()

uint32_t DIAG_GPIO_IOIM_GetStatus (void)

The API shall return the status variable of [DIAG_GPIO_IntegrityMonitorTest](#) diagnostic.

Returns:

DIAG_GPIO_IoimInstResult

12.3. Macros

12.3.1. Definition Documentation

12.3.1.1. PORTA_PTR

```
#define PORTA_PTR &PORTA
```

This macro automatically maps the PORTA address of the device from the device header.

12.3.1.2. PORTB_PTR

```
#define PORTB_PTR &PORTB
```

This macro automatically maps the PORTB address of the device from the device header.

12.3.1.3. PORTC_PTR

```
#define PORTC_PTR &PORTC
```

This macro automatically maps the PORTC address of the device from the device header.

12.3.1.4. PORTD_PTR

```
#define PORTD_PTR &PORTD
```

This macro automatically maps the PORTD address of the device from the device header.

12.3.1.5. DIAG_GPIO_IOIM_INSTANCES

```
#define DIAG_GPIO_IOIM_INSTANCES 4U
```

This macro indicates the number of instances of IO integrity monitor module.

12.4. Source Code Reference

The following table provides a list of APIs and their location in the source code:

MODULE	DIAGNOSTIC API	SOURCE CODE REFERENCE
GPIO	• DIAG_GPIO_ActivityCheck	diag_gpio_activitycheck.c
GPIO	• DIAG_GPIO_InputPractice	diag_gpio_inputpractice.c
GPIO	• DIAG_GPIO_InterruptGenTest	diag_gpio_interruptgentest.c
GPIO	• DIAG_GPIO_IOIM_GetStatus • DIAG_GPIO_IntegrityMonitorTest • DIAG_GPIO_IOIM_SetStatus	diag_gpio_iomonitor.c
GPIO	• DIAG_GPIO_OutputTest	diag_gpio_outputtest.c
GPIO	• DIAG_GPIO_PpsOutputConnectionTest	diag_gpio_ppsoutputtest.c

13. INTERRUPT

13.1. Overview

This document describes the diagnostics API for the Interrupt module. The safety requirements described for Interrupt diagnostics in the Safety Manual are implemented in the following APIs.

- [DIAG_INTERRUPT_FrequencyCheck](#)
- [DIAG_INTERRUPT_ServicingTest](#)
- [DIAG_INTERRUPT_IsrClearedCheck](#)
- [DIAG_INTERRUPT_HardTrapTest](#)
- [DIAG_INTERRUPT_ExternalInputTest](#)

Constraints/Limitations:

- For the [DIAG_INTERRUPT_ExternalInputTest](#) diagnostics API, it is expected that the application passes valid *interruptPol* to the API. The acceptable values are as defined in [DIAG_INTERRUPT_POLARITY](#)

Integration Rules:

The interrupt module diagnostics use the Interrupt Service Routines that are also used by the application. As the ISR is a common code shared between the module diagnostics and the application code, some APIs provided in the interrupt diagnostics module need to be used in the Application ISR routine. Two such APIs are [DIAG_INTERRUPT_UpdateISRNotifications](#) and [DIAG_INTERRUPT_IsInterruptServicingCheckRunning](#). Consider the following code snippet

```
void __attribute__ ( ( interrupt, no_auto_psv ) ) _CNAInterrupt(void)
{
    uint8_t testCounter = 0x00U;
    DIAG_INTERRUPT_UpdateISRNotifications(INTERRUPT_CNA);
    if(! (DIAG_INTERRUPT_IsInterruptServicingCheckRunning()))
    {
        //The application code is written here.
        testCounter += 1;
    }
    CNFABits.CNFA2 = 0; //Clear flag for Pin - RA2
    IFS0bits.CNAIF = 0;
}
```

Here the *_CNAInterrupt(void)* is the ISR which will service the change notification interrupt. The [DIAG_INTERRUPT_IsInterruptServicingCheckRunning](#) is used to prevent the application code from running when the [DIAG_INTERRUPT_ServicingTest](#) API is called at startup. Similarly the [DIAG_INTERRUPT_UpdateISRNotifications](#) API is used to update the interrupt count which would be used in the [DIAG_INTERRUPT_FrequencyCheck](#).

13.2. Functions

13.2.1. Function Documentation

13.2.1.1. DIAG_INTERRUPT_ServicingTest()

uint32_t DIAG_INTERRUPT_ServicingTest (uint32_t intrReqFlagRegArr[])

Software Requirement Reference ID : SW_INTERRUPT_SERVICING_TEST_01 Services the interrupt test by checking the interrupt request flag registers.

This function performs the servicing of the interrupt test by iterating through the interrupt request flag registers. It checks if the interrupts supported by the device match the requested interrupts and calls other functions to check and get the status of the interrupt flag registers.

Parameters:

intrReqFlagRegArr[] Array of interrupt request flag registers.

Returns:

uint32_t The result of the interrupt servicing test.

- DIAG_PASS: The test passed.
- DIAG_FAIL: The test failed.

The function implementation details:

- Initializes the return value to a failure state.
- Sets the interrupt servicing check running status to true.
- Iterates through the interrupt request flag registers.
- Checks if the requested interrupts are supported by the device.
- If the requested interrupts are not supported, sets the temporary return value to an invalid parameter state and breaks the loop.
- If the requested interrupts are non-zero, calls the function to check the IFS register.
- Calls the function to get the status of the interrupt flag registers.
- Gets the final return value.
- Sets the interrupt servicing check running status to false.
- Returns the final result of the interrupt servicing test.

Figure 13-1. Interrupt Servicing Test Sequence Diagram

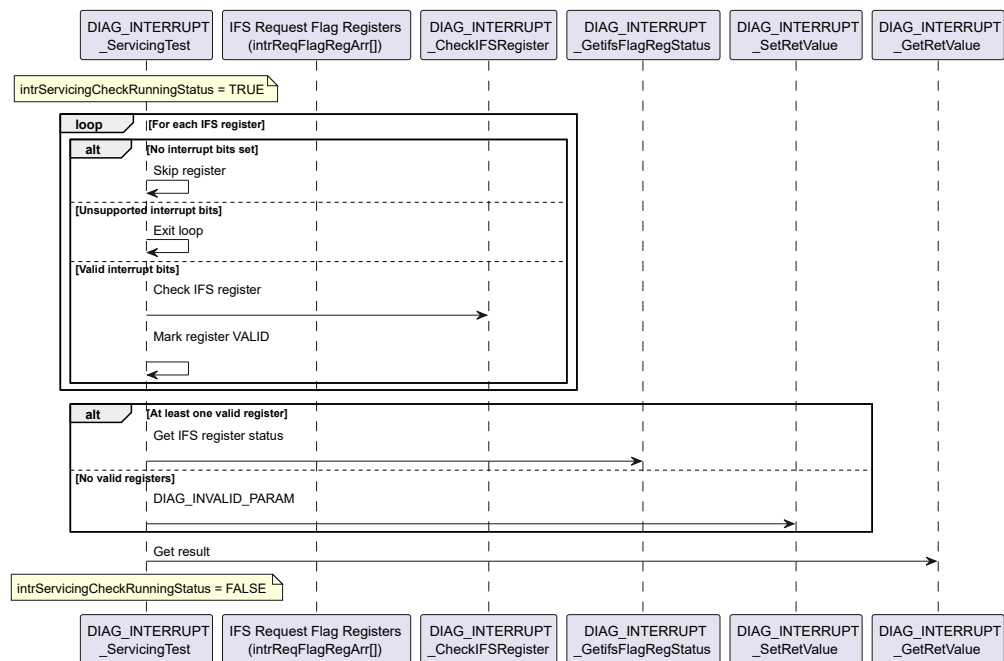
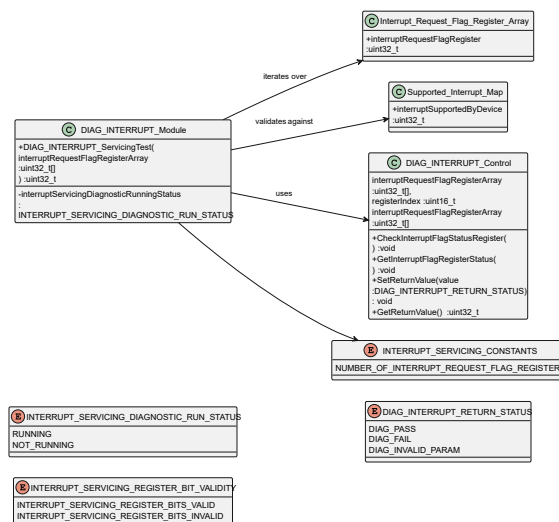


Figure 13-2. Interrupt Servicing Test Activity Diagram



13.2.1.2. DIAG_INTERRUPT_HardTrapTest()

uint32_t DIAG_INTERRUPT_HardTrapTest (void)

Software Requirement Reference ID : SW_HARD_TRAP_TEST_01

Performs a hard trap diagnostic test on the interrupt control register.

This function induces a software-generated hard trap and verifies if the interrupt control register responds correctly. It sets a software trap bit, waits for the bit to be cleared by the ISR, and checks the result.

Returns:

uint32_t Status value indicating the result of the hard trap test.

- DIAG_PASS: The test passed.
- DIAG_FAIL: The test failed.

The function implementation details:

- Initializes the return value and timeout variable.
- Sets a pointer to the interrupt control register.
- Sets the diagInduceHardTrap flag to indicate the diagnostic test is running.
- Sets the software trap bit in the interrupt control register.
- Waits for the software trap bit to be cleared or for the timeout to occur.
- Checks if the loop exited because the software trap bit was cleared by the ISR.
- Sets the return value to DIAG_PASS if the bit was cleared, otherwise sets it to DIAG_FAIL.
- Retrieves the final return value using DIAG_INTERRUPT_GetRetVal.
- Returns the final status value.

Figure 13-3. Interrupt Hard Trap test sequence diagram

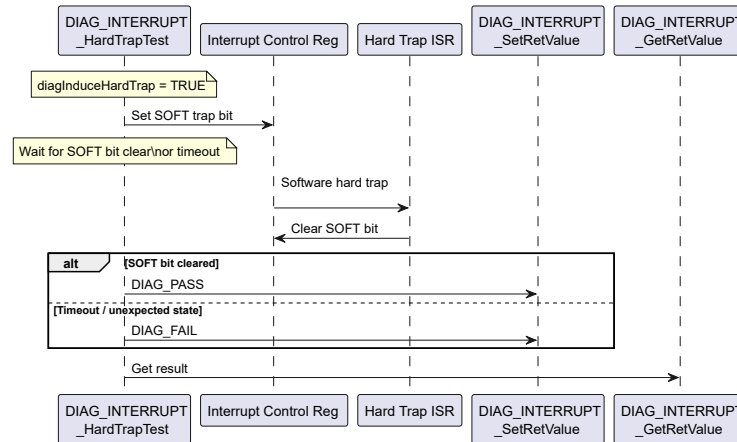
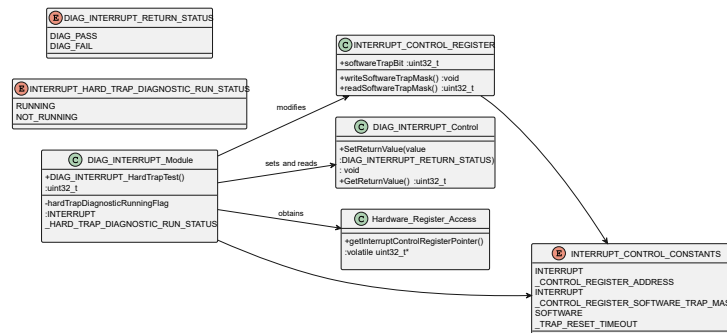


Figure 13-4. Interrupt Hard trap test activity diagram



13.2.1.3. DIAG_INTERRUPT_resetDiagInducedTrapStatus()

void DIAG_INTERRUPT_resetDiagInducedTrapStatus (void)

Resets the diagnostic induced trap status.

This function sets the diagInduceHardTrap variable to INTR_HARDTRAP_DIAG_RUN_FALSE, effectively resetting the diagnostic induced trap status.

13.2.1.4. DIAG_INTERRUPT_isTrapInducedByDiag()

DIAG_INTERRUPT_HARDTRAP_RUN_STATUS DIAG_INTERRUPT_isTrapInducedByDiag (void)

Checks if the hard trap was induced by diagnostics. This function returns the status indicating whether the hard trap was induced by diagnostics.

Returns:

The status of the diagnostic induced hard trap.

- DIAG_PASS: The test passed.
- DIAG_FAIL: The test failed.

13.2.1.5. DIAG_INTERRUPT_ifsFlagClearedCheckFromIsr()

void DIAG_INTERRUPT_ifsFlagClearedCheckFromIsr (uint16_t irqNum)

Checks if the IFS flag has been cleared from the ISR for the specified interrupt number.

This function verifies if the interrupt flag status (IFS) bit has been cleared for a given interrupt number from within the interrupt service routine (ISR). It performs several checks and updates relevant status flags based on the results.

Parameters:

in	irqNum	The interrupt number to be checked.
----	--------	-------------------------------------

The function implementation details:

- Initializes the register index and bit position based on the interrupt number.
- Retrieves the interrupt control status register pointer and the IFS and IEC register pointers.
- Calculates the register index and bit position for the given interrupt number.
- Checks if the ISR cleared check is running.
- Compares the current interrupt vector number with the expected value.
- If the vector number does not match, sets the ISR cleared check status to not complete.
- If the vector number matches, checks if the corresponding IFS bit is cleared.
- If the IFS bit is cleared, sets the IFSBitCleared flag to 1.
- If the IFS bit is not cleared, sets the IFSBitCleared flag to 0 and clears the corresponding IEC bit.
- Sets the ISR cleared check status to complete.
- Resets the ISR cleared check flag to prevent repeated checks.

13.2.1.6. DIAG_INTERRUPT_IsrClearedCheck()

uint32_t DIAG_INTERRUPT_IsrClearedCheck (uint16_t irqNum)

Software Requirement Reference ID : SW_ISR_CLEARED_CHECK_01 Checks if the ISR has been cleared for the specified interrupt number.

This function checks if the interrupt service routine (ISR) has been cleared for a given interrupt number. It sets the ISR cleared check status to running, then enters a loop to wait for the ISR cleared check to complete or for a timeout to occur. If the timeout occurs, it sets the return value to DIAG_INVALID_PARAM. If the check completes, it sets the return value to DIAG_PASS if the interrupt flag status bit is cleared, or DIAG_FAIL otherwise.

Parameters:

in	irqNum	The interrupt number to be checked.
----	--------	-------------------------------------

Returns:

uint32_t Status value indicating the result of the ISR cleared check.
• DIAG_PASS: The test passed. • DIAG_FAIL: The test failed.

The function implementation details:

- Initializes the timeout value to ISR_CLEARED_CHECK_TIMEOUT.
- Sets the ISR cleared check status to running.
- Enters a loop to wait for the ISR cleared check to complete or for the timeout to occur.
- If the timeout occurs, sets the return value to DIAG_INVALID_PARAM.
- If the check completes, sets the return value to DIAG_PASS if the interrupt flag status bit is cleared, or DIAG_FAIL otherwise.
- Retrieves the final return value using DIAG_INTERRUPT_GetRetVal.
- Returns the final status value.

Figure 13-5. Interrupt Isr Cleared Check Sequence Diagram

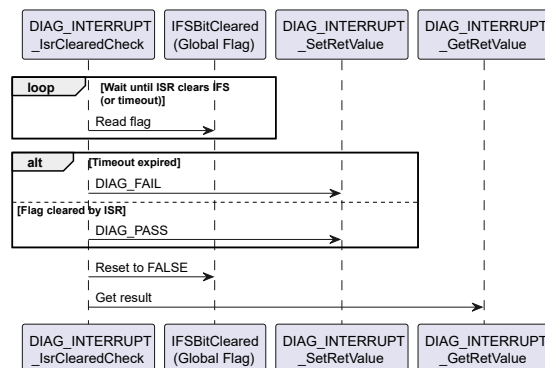
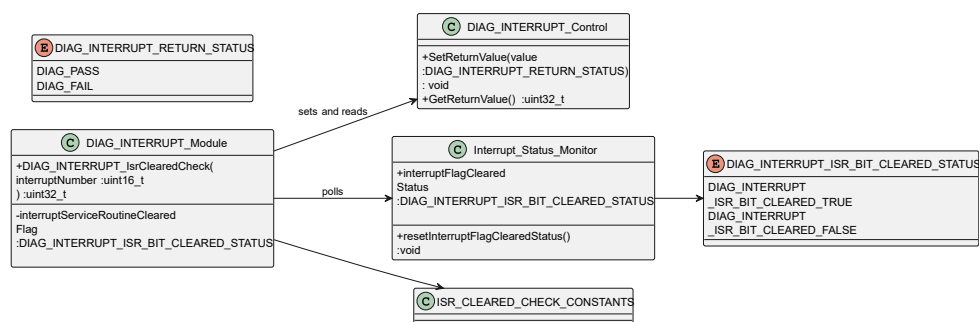


Figure 13-6. Interrupt Isr Cleared Check Activity Diagram



13.2.1.7. DIAG_INTERRUPT_FrequencyCheck()

uint32_t DIAG_INTERRUPT_FrequencyCheck (DIAG_INTERRUPT_IRQ_MODEL * modulePtr, uint32_t modulesForFreqCheck)

Software Requirement Reference ID : SW_INTERRUPT_FREQUENCY_CHECK_01 Performs a frequency check diagnostic test on interrupt modules.

This function checks the interrupt frequency for a given set of modules. It iterates through the module interrupt request (IRQ) structure to verify the interrupt counter status for each module involved in the frequency test. The function returns a status value indicating the result of the test.

Parameters:

in	modulePtr	Pointer to the structure containing module IRQ information.
in	modulesForFreqCheck	Number of modules to be checked for interrupt frequency.

Returns:

uint32_t Status value indicating the result of the frequency check test.

- DIAG_PASS: The test passed.
- DIAG_FAIL: The test failed.

The function implementation details:

- Initializes the return value.
- Iterates through the module IRQ structure to check the interrupt counter status for each module.
- Skips modules that have been removed from the structure.
- Decrements the individual module interrupt frequency test counter.

- Calls DIAG_INTERRUPT_CheckModuleInterruptCounter to check the interrupt counter for each module.
- Resets the module index and checks if the API should return pass or fail.
- Sets the return value to DIAG_FAIL if any module fails the frequency test, otherwise sets it to DIAG_PASS.
- Retrieves the final return value using DIAG_INTERRUPT_GetRetVal.
- Returns the final status value.

Figure 13-7. Interrupt Frequency Check Sequence Diagram

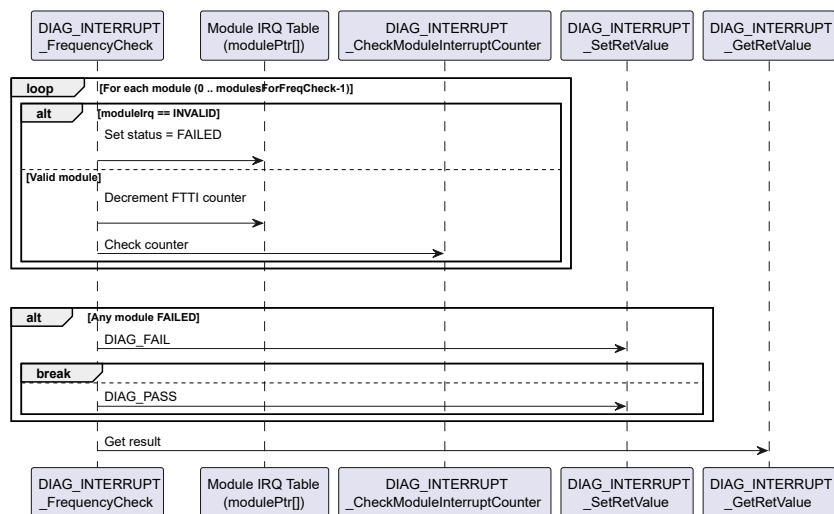
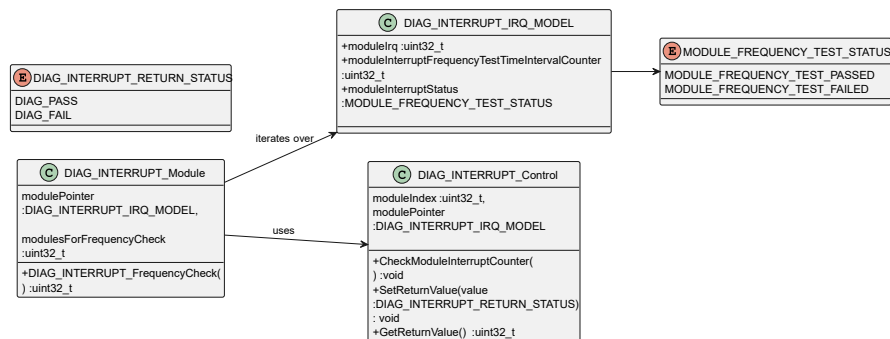


Figure 13-8. Interrupt Frequency Check Activity Diagram



13.2.1.8. DIAG_INTERRUPT_GetFreqTestResults()

void DIAG_INTERRUPT_GetFreqTestResults (uint32_t * failCount, DIAG_INTERRUPT_MODULE_STATUS * IrqDataTable, DIAG_INTERRUPT_IRQ_MODEL * modulePtr, uint32_t size)

Retrieves the frequency test results for diagnostic interrupts.

This function updates the interrupt status, IRQ number, and interrupt counter for each module being monitored. It also increments the fail count for modules that have failed the frequency test.

Parameters:

out	failCount	Pointer to the variable that will store the count of failed modules.
out	IrqDataTable	Pointer to the array that will store the interrupt status data for each module.
in	modulePtr	Pointer to the array of modules being monitored.

in	size	The number of modules in the modulePtr array.
----	------	---

13.2.1.9. DIAG_INTERRUPT_UpdateISRNotifications()

void DIAG_INTERRUPT_UpdateISRNotifications (uint32_t irqNum, DIAG_INTERRUPT_IRQ_MODEL * modulePtr, uint32_t modsize)

Updates the ISR notifications for a specified interrupt.

This function updates the interrupt service routine (ISR) notifications for a given interrupt number. It checks if the interrupt frequency test is running, scans through the interrupt module structure to find the module index, and updates the module interrupt counter if the module interrupt status is not failed.

Parameters:

in	irqNum	The interrupt number to be updated.
in	modulePtr	Pointer to the structure containing module interrupt information.
in	modsize	The size of the module interrupt structure.

The function implementation details:

- Retrieves the current diagnostic status of the interrupt frequency test.
- If the frequency test is running, it scans through the module structure to find the module index.
- If the module index is found, it updates the module interrupt counter if the module interrupt status is not failed.
- If the module index is not found, it sets the index to INVALID_MODULE_INDEX.
- The module interrupt model structure is updated only if the module interrupt status is MODULE_FREQ_TEST_PASSED or MODULE_FREQ_TEST_IN_PROGRESS.
- If the module interrupt status is MODULE_FREQ_TEST_FAILED, the module interrupt counter is not updated.

13.2.1.10. DIAG_INTERRUPT_StartFrequencyTest()

void DIAG_INTERRUPT_StartFrequencyTest (void)

Starts the frequency test for diagnostic interrupts.

This function sets the intrFreqRunFlag to INTR_FREQ_DIAG_RUN_TRUE, indicating that the frequency test for diagnostic interrupts has started.

13.2.1.11. DIAG_INTERRUPT_StopFrequencyTest()

void DIAG_INTERRUPT_StopFrequencyTest (void)

Stops the frequency test for diagnostic interrupts.

This function sets the intrFreqRunFlag to INTR_FREQ_DIAG_RUN_FALSE, indicating that the frequency test for diagnostic interrupts has stopped.

13.2.1.12. DIAG_INTERRUPT_IsInterruptServicingCheckRunning()

DIAG_INTERRUPT_SERVICING_RUN_STATUS DIAG_INTERRUPT_IsInterruptServicingCheckRunning (void)

Checks if the interrupt servicing check is currently running. This function returns the current status of the interrupt servicing check.

Returns:

The status indicating whether the interrupt servicing check is running.

- DIAG_PASS: The test passed.
- DIAG_FAIL: The test failed.

13.2.1.13. DIAG_INTERRUPT_ExternalInputTest()

uint32_t DIAG_INTERRUPT_ExternalInputTest (DIAG_INTERRUPT_EXTERNAL_INTR_NUM interruptNum, DIAG_INTERRUPT_POLARITY interruptPol, void(*)() trigForInterrupt)

Software Requirement Reference ID : SW_EXTERNAL_INTERRUPT_TEST_01 Performs an external input test on the specified interrupt.

This function tests an external interrupt by triggering it and checking the result. It sets the test status to in-progress, verifies the trigger function pointer, and then calls the function to trigger and test the external interrupt.

Parameters:

in	interruptNum	The external interrupt number to be tested.
in	interruptPol	The polarity of the external interrupt.
in	trigForInterrupt	Pointer to the function that triggers the interrupt.

Returns:

uint32_t Status value indicating the result of the external input test.

- DIAG_PASS: The test passed.
- DIAG_FAIL: The test failed.

The function implementation details:

- Sets the external interrupt test status to in-progress.
- Sets the running status flag to true.
- Checks if the trigger function pointer is NULL.
- If the trigger function pointer is NULL, sets the return value to DIAG_INVALID_PARAM.
- If the trigger function pointer is not NULL, calls DIAG_INTERRUPT_TriggerAndTestExternalInterrupts with the interrupt number, polarity, and trigger function.
- Retrieves the final return value using DIAG_INTERRUPT_GetRetVal.
- Sets the running status flag to false.
- Returns the final status value.

Figure 13-9. Interrupt external input test sequence diagram

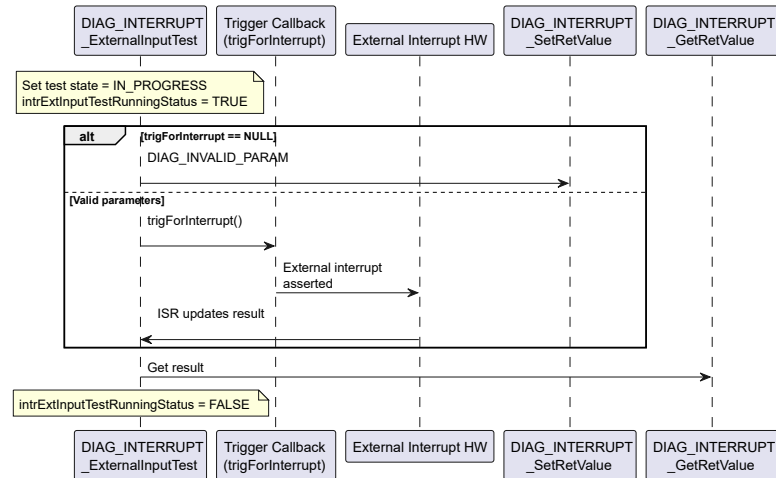
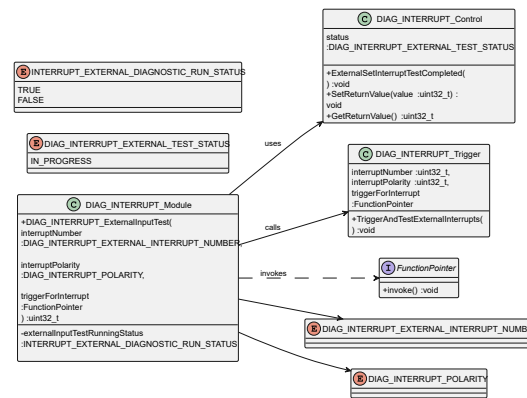


Figure 13-10. Interrupt external input test class diagram



13.2.1.14. DIAG_INTERRUPT_ExternalSetIntrTestCompleted()

void DIAG_INTERRUPT_ExternalSetIntrTestCompleted (uint32_t extIntrTestStatus)

Sets the status of the external interrupt test. This function updates the status of the external interrupt test by setting the extIntrTestCompleted variable to the provided status.

Parameters:

in	extIntrTestStatus	The status to set for the external interrupt test.
----	-------------------	--

13.2.1.15. DIAG_INTERRUPT_IsExtIntrTestRunning()

DIAG_INTERRUPT_EXT_INTR_RUN_STATUS DIAG_INTERRUPT_IsExtIntrTestRunning (void)

Checks if the external interrupt test is running. This function returns the current status of the external interrupt test.

Returns:

The status indicating whether the external interrupt test is running.

- DIAG_PASS: The test passed.
- DIAG_FAIL: The test failed.

13.2.1.16. FAULTINJ_INTERRUPT_ServicingCheck()

void FAULTINJ_INTERRUPT_ServicingCheck (uint32_t irqNumber)

Software Requirement Reference ID : SW_INTERRUPT_SERVICING_CHECK_01 This API injects the fault for the interrupt servicing check diagnostic API. The interrupt priority of the requested irqNumber is made 0 to ensure that the fault is injected.

Parameters:

irqNumber	is the interrupt request number for which the fault needs to be injected.
-----------	---

Returns:

void

Code Snippet:

```
uint16_t TEST_INTERRUPT_ServicingCheck(uint8_t testOption)
{
    uint16_t retval;
    The application HAS TO fill the entire array including the registers for which it does
    not intend
    to perform the interrupt servicing check. For registers where none of the interrupt bits
    need
    to be tested, the applicaiton has to fill 0x0000
    uint16_t InterruptFlagStatusReg[12] = {0x7FDE, 0xFBEB, 0xFB7F,
                                           0x803F, 0x387B, 0xFFFE,
                                           0xFFFF, 0xFFF8, 0x3FFE,
                                           0x2364, 0xFF00, 0x601F};

    Update the runDiagnostics so that the ISRs dont update the ModuleInterruptModel
    structure accidentally
    switch (testOption)
    {
        ...
        ...
        ...
        case TEST_SERVICECHECK_WITH_FAULTINJECT:
            FAULTINJ_INTERRUPT_ServicingCheck(INTERRUPT_CCT1);
            retval = DIAG_INTERRUPT_ServicingCheck(InterruptFlagStatusReg);
            break;
    }
    return retval;
}
```

13.2.1.17. FAULTINJ_INTERRUPT_ExternalInterrupt()

void FAULTINJ_INTERRUPT_ExternalInterrupt (DIAG_INTERRUPT_EXTERNAL_INTR_NUM
interruptNum)

Software Requirement Reference ID : SW_EXTERNAL_INTERRUPT_TEST_01 This API injects the fault for the external interrupt input API diagnostic. The interrupt priority of the requested irqNumber is made 0 to ensure that the fault is injected.

Parameters:

interruptNum	is one of the external interrupt pins for where the fault needs to be injected.
--------------	---

Returns:

void

Code Snippet:

```
uint16_t TEST_INTERRUPT_ExternalInputTest(uint8_t testOption)
{
    uint16_t testresult;

    switch(testOption)
    {
        ...
        ...
        ...
        case TEST_INT0_FAULT_INJECT:
            Clear the interrupt flag initially
            IFS0bits.INT0IF = 0;
            Inject the fault
            FAULTINJ_INTERRUPT_ExternalInterrupt(EXTERNAL_INTERRUPT_0);
            testresult =
            DIAG_INTERRUPT_ExternalInputTest(EXTERNAL_INTERRUPT_0, INTERRUPT_ON_POSITIVE_EDGE, triggerInt0InterruptOnPosEdge);
            break;
    }
}
```

13.3. Macros

13.3.1. Definition Documentation

13.3.1.1. DIAG_INTERRUPT_IVTCIF

#define DIAG_INTERRUPT_IVTCIF 0U

Interrupt Vector Table Configuration Interrupt Flag.

13.3.1.2. DIAG_INTERRUPT_CPUFPUIF

#define DIAG_INTERRUPT_CPUFPUIF 1U

CPU Floating Point Unit Interrupt Flag.

13.3.1.3. DIAG_INTERRUPT_XRAMECCIF

#define DIAG_INTERRUPT_XRAMECCIF 2U

XRAM ECC Interrupt Flag.

13.3.1.4. DIAG_INTERRUPT_YRAMECCIF

#define DIAG_INTERRUPT_YRAMECCIF 3U

YRAM ECC Interrupt Flag.

13.3.1.5. DIAG_INTERRUPT_PBERRIF

#define DIAG_INTERRUPT_PBERRIF 4U

Peripheral Bus Error Interrupt Flag.

13.3.1.6. DIAG_INTERRUPT_NVMECCIF

#define DIAG_INTERRUPT_NVMECCIF 5U

NVM ECC Interrupt Flag.

13.3.1.7. DIAG_INTERRUPT_NVMIF

#define DIAG_INTERRUPT_NVMIF 6U

NVM Interrupt Flag.

13.3.1.8. DIAG_INTERRUPT_NVMCRCIF

#define DIAG_INTERRUPT_NVMCRCIF 7U

NVM CRC Interrupt Flag.

13.3.1.9. DIAG_INTERRUPT_CLKFAILIF

#define DIAG_INTERRUPT_CLKFAILIF 9U

Clock Failure Interrupt Flag.

13.3.1.10. DIAG_INTERRUPT_CLKERRIF

#define DIAG_INTERRUPT_CLKERRIF 10U

Clock Error Interrupt Flag.

13.3.1.11. DIAG_INTERRUPT_C1FAIL1IF

#define DIAG_INTERRUPT_C1FAIL1IF 11U

CAN1 Failure 1 Interrupt Flag.

13.3.1.12. DIAG_INTERRUPT_C1WARNIF

#define DIAG_INTERRUPT_C1WARNIF 12U

CAN1 Warning Interrupt Flag.

13.3.1.13. DIAG_INTERRUPT_C1MONIF

#define DIAG_INTERRUPT_C1MONIF 13U

CAN1 Monitor Interrupt Flag.

13.3.1.14. DIAG_INTERRUPT_C1RDYIF

#define DIAG_INTERRUPT_C1RDYIF 14U

CAN1 Ready Interrupt Flag.

13.3.1.15. DIAG_INTERRUPT_C2FAILIF

#define DIAG_INTERRUPT_C2FAILIF 15U

CAN2 Failure Interrupt Flag.

13.3.1.16. DIAG_INTERRUPT_C2WARNIF

#define DIAG_INTERRUPT_C2WARNIF 16U

CAN2 Warning Interrupt Flag.

13.3.1.17. DIAG_INTERRUPT_C2MONIF

#define DIAG_INTERRUPT_C2MONIF 17U

CAN2 Monitor Interrupt Flag.

13.3.1.18. DIAG_INTERRUPT_C2RDYIF

#define DIAG_INTERRUPT_C2RDYIF 18U

CAN2 Ready Interrupt Flag.

13.3.1.19. DIAG_INTERRUPT_C3FAILIF

#define DIAG_INTERRUPT_C3FAILIF 19U

CAN3 Failure Interrupt Flag.

13.3.1.20. DIAG_INTERRUPT_C3WARNIF

#define DIAG_INTERRUPT_C3WARNIF 20U

CAN3 Warning Interrupt Flag.

13.3.1.21. DIAG_INTERRUPT_C3MONIF

#define DIAG_INTERRUPT_C3MONIF 21U

CAN3 Monitor Interrupt Flag.

13.3.1.22. DIAG_INTERRUPT_C3RDYIF

#define DIAG_INTERRUPT_C3RDYIF 22U

CAN3 Ready Interrupt Flag.

13.3.1.23. DIAG_INTERRUPT_C4FAILIF

#define DIAG_INTERRUPT_C4FAILIF 23U

CAN4 Failure Interrupt Flag.

13.3.1.24. DIAG_INTERRUPT_C4WARNIF

#define DIAG_INTERRUPT_C4WARNIF 24U

CAN4 Warning Interrupt Flag.

13.3.1.25. DIAG_INTERRUPT_C4MONIF

#define DIAG_INTERRUPT_C4MONIF 25U

CAN4 Monitor Interrupt Flag.

13.3.1.26. DIAG_INTERRUPT_C4RDYIF

#define DIAG_INTERRUPT_C4RDYIF 26U

CAN4 Ready Interrupt Flag.

13.3.1.27. DIAG_INTERRUPT_WDTIF

#define DIAG_INTERRUPT_WDTIF 28U

Watchdog Timer Interrupt Flag.

13.3.1.28. DIAG_INTERRUPT_INT0IF

#define DIAG_INTERRUPT_INT0IF 33U

External Interrupt 0 Flag.

13.3.1.29. DIAG_INTERRUPT_INT1IF

#define DIAG_INTERRUPT_INT1IF 34U

External Interrupt 1 Flag.

13.3.1.30. DIAG_INTERRUPT_INT2IF

#define DIAG_INTERRUPT_INT2IF 35U

External Interrupt 2 Flag.

13.3.1.31. DIAG_INTERRUPT_INT3IF

#define DIAG_INTERRUPT_INT3IF 36U

External Interrupt 3 Flag.

13.3.1.32. DIAG_INTERRUPT_INT4IF

#define DIAG_INTERRUPT_INT4IF 37U

External Interrupt 4 Flag.

13.3.1.33. DIAG_INTERRUPT_PEVTAIF

#define DIAG_INTERRUPT_PEVTAIF 38U

Peripheral Event A Interrupt Flag.

13.3.1.34. DIAG_INTERRUPT_PVBTBIF

#define DIAG_INTERRUPT_PVBTBIF 39U

Peripheral Event B Interrupt Flag.

13.3.1.35. DIAG_INTERRUPT_PEVTCIF

#define DIAG_INTERRUPT_PEVTCIF 40U

Peripheral Event C Interrupt Flag.

13.3.1.36. DIAG_INTERRUPT_PEVTDIF

#define DIAG_INTERRUPT_PEVTDIF 41U

Peripheral Event D Interrupt Flag.

13.3.1.37. DIAG_INTERRUPT_PEVTEIF

#define DIAG_INTERRUPT_PEVTEIF 42U

Peripheral Event E Interrupt Flag.

13.3.1.38. DIAG_INTERRUPT_PEVTFIF

#define DIAG_INTERRUPT_PEVTFIF 43U

Peripheral Event F Interrupt Flag.

13.3.1.39. DIAG_INTERRUPT_PWM1IF

#define DIAG_INTERRUPT_PWM1IF 44U

PWM1 Interrupt Flag.

13.3.1.40. DIAG_INTERRUPT_PWM2IF

#define DIAG_INTERRUPT_PWM2IF 45U

PWM2 Interrupt Flag.

13.3.1.41. DIAG_INTERRUPT_PWM3IF

#define DIAG_INTERRUPT_PWM3IF 46U

PWM3 Interrupt Flag.

13.3.1.42. DIAG_INTERRUPT_PWM4IF

#define DIAG_INTERRUPT_PWM4IF 47U

PWM4 Interrupt Flag.

13.3.1.43. DIAG_INTERRUPT_T1IF

#define DIAG_INTERRUPT_T1IF 48U

Timer 1 Interrupt Flag.

13.3.1.44. DIAG_INTERRUPT_CCT1IF

#define DIAG_INTERRUPT_CCT1IF 49U

Capture/Compare Timer 1 Interrupt Flag.

13.3.1.45. DIAG_INTERRUPT_CCP1IF

#define DIAG_INTERRUPT_CCP1IF 50U

Capture/Compare 1 Interrupt Flag.

13.3.1.46. DIAG_INTERRUPT_CCT2IF

#define DIAG_INTERRUPT_CCT2IF 51U

Capture/Compare Timer 2 Interrupt Flag.

13.3.1.47. DIAG_INTERRUPT_CCP2IF

#define DIAG_INTERRUPT_CCP2IF 52U

Capture/Compare 2 Interrupt Flag.

13.3.1.48. DIAG_INTERRUPT_CCT3IF

#define DIAG_INTERRUPT_CCT3IF 53U

Capture/Compare Timer 3 Interrupt Flag.

13.3.1.49. DIAG_INTERRUPT_CCP3IF

#define DIAG_INTERRUPT_CCP3IF 54U

Capture/Compare 3 Interrupt Flag.

13.3.1.50. DIAG_INTERRUPT_CCT4IF

#define DIAG_INTERRUPT_CCT4IF 55U

Capture/Compare Timer 4 Interrupt Flag.

13.3.1.51. DIAG_INTERRUPT_CCP4IF

#define DIAG_INTERRUPT_CCP4IF 56U

Capture/Compare 4 Interrupt Flag.

13.3.1.52. DIAG_INTERRUPT_SPI1RXIF

#define DIAG_INTERRUPT_SPI1RXIF 63U

SPI1 Receive Interrupt Flag.

13.3.1.53. DIAG_INTERRUPT_SPI1TXIF

#define DIAG_INTERRUPT_SPI1TXIF 64U

SPI1 Transmit Interrupt Flag.

13.3.1.54. DIAG_INTERRUPT_SPI1EIF

#define DIAG_INTERRUPT_SPI1EIF 65U

SPI1 Error Interrupt Flag.

13.3.1.55. DIAG_INTERRUPT_SPI2RXIF

#define DIAG_INTERRUPT_SPI2RXIF 66U

SPI2 Receive Interrupt Flag.

13.3.1.56. DIAG_INTERRUPT_SPI2TXIF

#define DIAG_INTERRUPT_SPI2TXIF 67U

SPI2 Transmit Interrupt Flag.

13.3.1.57. DIAG_INTERRUPT_SPI2EIF

#define DIAG_INTERRUPT_SPI2EIF 68U

SPI2 Error Interrupt Flag.

13.3.1.58. DIAG_INTERRUPT_SPI3RXIF

#define DIAG_INTERRUPT_SPI3RXIF 69U

SPI3 Receive Interrupt Flag.

13.3.1.59. DIAG_INTERRUPT_SPI3TXIF

#define DIAG_INTERRUPT_SPI3TXIF 70U

SPI3 Transmit Interrupt Flag.

13.3.1.60. DIAG_INTERRUPT_SPI3EIF

#define DIAG_INTERRUPT_SPI3EIF 71U

SPI3 Error Interrupt Flag.

13.3.1.61. DIAG_INTERRUPT_DMA0IF

#define DIAG_INTERRUPT_DMA0IF 72U

DMA Channel 0 Interrupt Flag.

13.3.1.62. DIAG_INTERRUPT_DMA1IF

#define DIAG_INTERRUPT_DMA1IF 73U

DMA Channel 1 Interrupt Flag.

13.3.1.63. DIAG_INTERRUPT_DMA2IF

#define DIAG_INTERRUPT_DMA2IF 74U

DMA Channel 2 Interrupt Flag.

13.3.1.64. DIAG_INTERRUPT_DMA3IF

#define DIAG_INTERRUPT_DMA3IF 75U

DMA Channel 3 Interrupt Flag.

13.3.1.65. DIAG_INTERRUPT_CMP1IF

#define DIAG_INTERRUPT_CMP1IF 76U

Comparator 1 Interrupt Flag.

13.3.1.66. DIAG_INTERRUPT_CMP2IF

#define DIAG_INTERRUPT_CMP2IF 77U

Comparator 2 Interrupt Flag.

13.3.1.67. DIAG_INTERRUPT_CMP3IF

#define DIAG_INTERRUPT_CMP3IF 78U

Comparator 3 Interrupt Flag.

13.3.1.68. DIAG_INTERRUPT_I2C1EIF

#define DIAG_INTERRUPT_I2C1EIF 80U

I2C1 Error Interrupt Flag.

13.3.1.69. DIAG_INTERRUPT_I2C1IF

#define DIAG_INTERRUPT_I2C1IF 81U

I2C1 Interrupt Flag.

13.3.1.70. DIAG_INTERRUPT_I2C1RXIF

#define DIAG_INTERRUPT_I2C1RXIF 82U

I2C1 Receive Interrupt Flag.

13.3.1.71. DIAG_INTERRUPT_I2C1TXIF

#define DIAG_INTERRUPT_I2C1TXIF 83U

I2C1 Transmit Interrupt Flag.

13.3.1.72. DIAG_INTERRUPT_I2C2EIF

#define DIAG_INTERRUPT_I2C2EIF 84U

I2C2 Error Interrupt Flag.

13.3.1.73. DIAG_INTERRUPT_I2C2IF

#define DIAG_INTERRUPT_I2C2IF 85U

I2C2 Interrupt Flag.

13.3.1.74. DIAG_INTERRUPT_I2C2RXIF

#define DIAG_INTERRUPT_I2C2RXIF 86U

I2C2 Receive Interrupt Flag.

13.3.1.75. DIAG_INTERRUPT_I2C2TXIF

#define DIAG_INTERRUPT_I2C2TXIF 87U

I2C2 Transmit Interrupt Flag.

13.3.1.76. DIAG_INTERRUPT_U1RXIF

#define DIAG_INTERRUPT_U1RXIF 89U

UART1 Receive Interrupt Flag.

13.3.1.77. DIAG_INTERRUPT_U1TXIF

#define DIAG_INTERRUPT_U1TXIF 90U

UART1 Transmit Interrupt Flag.

13.3.1.78. DIAG_INTERRUPT_U1EIF

#define DIAG_INTERRUPT_U1EIF 91U

UART1 Error Interrupt Flag.

13.3.1.79. DIAG_INTERRUPT_U1EVTIF

#define DIAG_INTERRUPT_U1EVTIF 92U

UART1 Event Interrupt Flag.

13.3.1.80. DIAG_INTERRUPT_U2RXIF

#define DIAG_INTERRUPT_U2RXIF 93U

UART2 Receive Interrupt Flag.

13.3.1.81. DIAG_INTERRUPT_U2TXIF

#define DIAG_INTERRUPT_U2TXIF 94U

UART2 Transmit Interrupt Flag.

13.3.1.82. DIAG_INTERRUPT_U2EIF

#define DIAG_INTERRUPT_U2EIF 95U

UART2 Error Interrupt Flag.

13.3.1.83. DIAG_INTERRUPT_U2EVTIF

#define DIAG_INTERRUPT_U2EVTIF 96U

UART2 Event Interrupt Flag.

13.3.1.84. DIAG_INTERRUPT_U3RXIF

#define DIAG_INTERRUPT_U3RXIF 97U

UART3 Receive Interrupt Flag.

13.3.1.85. DIAG_INTERRUPT_U3TXIF

#define DIAG_INTERRUPT_U3TXIF 98U

UART3 Transmit Interrupt Flag.

13.3.1.86. DIAG_INTERRUPT_U3EIF

#define DIAG_INTERRUPT_U3EIF 99U

UART3 Error Interrupt Flag.

13.3.1.87. DIAG_INTERRUPT_U3EVTIF

#define DIAG_INTERRUPT_U3EVTIF 100U

UART3 Event Interrupt Flag.

13.3.1.88. DIAG_INTERRUPT_SENT1IF

#define DIAG_INTERRUPT_SENT1IF 105U

SENT1 Interrupt Flag.

13.3.1.89. DIAG_INTERRUPT_SENT1EIF

#define DIAG_INTERRUPT_SENT1EIF 106U

SENT1 Error Interrupt Flag.

13.3.1.90. DIAG_INTERRUPT_SENT2IF

#define DIAG_INTERRUPT_SENT2IF 107U

SENT2 Interrupt Flag.

13.3.1.91. DIAG_INTERRUPT_SENT2EIF

#define DIAG_INTERRUPT_SENT2EIF 108U

SENT2 Error Interrupt Flag.

13.3.1.92. DIAG_INTERRUPT_DMA4IF

#define DIAG_INTERRUPT_DMA4IF 109U

DMA Channel 4 Interrupt Flag.

13.3.1.93. DIAG_INTERRUPT_DMA5IF

#define DIAG_INTERRUPT_DMA5IF 110U

DMA Channel 5 Interrupt Flag.

13.3.1.94. DIAG_INTERRUPT_CNAIF

#define DIAG_INTERRUPT_CNAIF 113U

Change Notification A Interrupt Flag.

13.3.1.95. DIAG_INTERRUPT_CNBIF

#define DIAG_INTERRUPT_CNBIF 114U

Change Notification B Interrupt Flag.

13.3.1.96. DIAG_INTERRUPT_CNCIF

#define DIAG_INTERRUPT_CNCIF 115U

Change Notification C Interrupt Flag.

13.3.1.97. DIAG_INTERRUPT_CNDIF

#define DIAG_INTERRUPT_CNDIF 116U

Change Notification D Interrupt Flag.

13.3.1.98. DIAG_INTERRUPT_QEI1IF

#define DIAG_INTERRUPT_QEI1IF 125U

Interrupt flag for QE1.

13.3.1.99. DIAG_INTERRUPT_BISS1EIF

#define DIAG_INTERRUPT_BISS1EIF 129U

Interrupt flag for BISS1E.

13.3.1.100. DIAG_INTERRUPT_BISS1IF

#define DIAG_INTERRUPT_BISS1IF 130U

Interrupt flag for BISS1.

13.3.1.101. DIAG_INTERRUPT_CRCIF

#define DIAG_INTERRUPT_CRCIF 131U

Interrupt flag for CRC.

13.3.1.102. DIAG_INTERRUPT_PTGSTEPF

#define DIAG_INTERRUPT_PTGSTEPF 134U

Interrupt flag for PTGSTEP.

13.3.1.103. DIAG_INTERRUPT_PTGWDTIF

#define DIAG_INTERRUPT_PTGWDTIF 135U

Interrupt flag for PTGWDT.

13.3.1.104. DIAG_INTERRUPT_PTGOIF

#define DIAG_INTERRUPT_PTGOIF 136U

Interrupt flag for PTG0.

13.3.1.105. DIAG_INTERRUPT_PTG1IF

#define DIAG_INTERRUPT_PTG1IF 137U

Interrupt flag for PTG1.

13.3.1.106. DIAG_INTERRUPT_PTG2IF

#define DIAG_INTERRUPT_PTG2IF 138U

Interrupt flag for PTG2.

13.3.1.107. DIAG_INTERRUPT_PTG3IF

#define DIAG_INTERRUPT_PTG3IF 139U

Interrupt flag for PTG3.

13.3.1.108. DIAG_INTERRUPT_AD1CH0IF

#define DIAG_INTERRUPT_AD1CH0IF 146U

Interrupt flag for AD1CH0.

13.3.1.109. DIAG_INTERRUPT_AD1CMP0IF

#define DIAG_INTERRUPT_AD1CMP0IF 147U

Interrupt flag for AD1CMP0.

13.3.1.110. DIAG_INTERRUPT_AD1CH1IF

#define DIAG_INTERRUPT_AD1CH1IF 148U

Interrupt flag for AD1CH1.

13.3.1.111. DIAG_INTERRUPT_AD1CMP1IF

#define DIAG_INTERRUPT_AD1CMP1IF 149U

Interrupt flag for AD1CMP1.

13.3.1.112. DIAG_INTERRUPT_AD1CH2IF

#define DIAG_INTERRUPT_AD1CH2IF 150U

Interrupt flag for AD1CH2.

13.3.1.113. DIAG_INTERRUPT_AD1CMP2IF

#define DIAG_INTERRUPT_AD1CMP2IF 151U

Interrupt flag for AD1CMP2.

13.3.1.114. DIAG_INTERRUPT_AD1CH3IF

#define DIAG_INTERRUPT_AD1CH3IF 152U

Interrupt flag for AD1CH3.

13.3.1.115. DIAG_INTERRUPT_AD1CMP3IF

#define DIAG_INTERRUPT_AD1CMP3IF 153U

Interrupt flag for AD1CMP3.

13.3.1.116. DIAG_INTERRUPT_AD1CH4IF

#define DIAG_INTERRUPT_AD1CH4IF 154U

Interrupt flag for AD1CH4.

13.3.1.117. DIAG_INTERRUPT_AD1CMP4IF

#define DIAG_INTERRUPT_AD1CMP4IF 155U

Interrupt flag for AD1CMP4.

13.3.1.118. DIAG_INTERRUPT_AD1CH5IF

#define DIAG_INTERRUPT_AD1CH5IF 156U

Interrupt flag for AD1CH5.

13.3.1.119. DIAG_INTERRUPT_AD1CMP5IF

#define DIAG_INTERRUPT_AD1CMP5IF 157U

Interrupt flag for AD1CMP5.

13.3.1.120. DIAG_INTERRUPT_AD1CH6IF

#define DIAG_INTERRUPT_AD1CH6IF 158U

Interrupt flag for AD1CH6.

13.3.1.121. DIAG_INTERRUPT_AD1CMP6IF

#define DIAG_INTERRUPT_AD1CMP6IF 159U

Interrupt flag for AD1CMP6.

13.3.1.122. DIAG_INTERRUPT_AD1CH7IF

#define DIAG_INTERRUPT_AD1CH7IF 160U

Interrupt flag for AD1CH7.

13.3.1.123. DIAG_INTERRUPT_AD1CMP7IF

#define DIAG_INTERRUPT_AD1CMP7IF 161U

Interrupt flag for AD1CMP7.

13.3.1.124. DIAG_INTERRUPT_AD2CH0IF

#define DIAG_INTERRUPT_AD2CH0IF 188U

Interrupt flag for AD2CH0.

13.3.1.125. DIAG_INTERRUPT_AD2CMP0IF

#define DIAG_INTERRUPT_AD2CMP0IF 189U

Interrupt flag for AD2CMP0.

13.3.1.126. DIAG_INTERRUPT_AD2CH1IF

#define DIAG_INTERRUPT_AD2CH1IF 190U

Interrupt flag for AD2CH1.

13.3.1.127. DIAG_INTERRUPT_AD2CMP1IF

#define DIAG_INTERRUPT_AD2CMP1IF 191U

Interrupt flag for AD2CMP1.

13.3.1.128. DIAG_INTERRUPT_AD2CH2IF

#define DIAG_INTERRUPT_AD2CH2IF 192U

Interrupt flag for AD2CH2.

13.3.1.129. DIAG_INTERRUPT_AD2CMP2IF

#define DIAG_INTERRUPT_AD2CMP2IF 193U

Interrupt flag for AD2CMP2.

13.3.1.130. DIAG_INTERRUPT_AD2CH3IF

#define DIAG_INTERRUPT_AD2CH3IF 194U

Interrupt flag for AD2CH3.

13.3.1.131. DIAG_INTERRUPT_AD2CMP3IF

#define DIAG_INTERRUPT_AD2CMP3IF 195U

Interrupt flag for AD2CMP3.

13.3.1.132. DIAG_INTERRUPT_AD2CH4IF

#define DIAG_INTERRUPT_AD2CH4IF 196U

Interrupt flag for AD2CH4.

13.3.1.133. DIAG_INTERRUPT_AD2CMP4IF

#define DIAG_INTERRUPT_AD2CMP4IF 197U

Interrupt flag for AD2CMP4.

13.3.1.134. DIAG_INTERRUPT_AD2CH5IF

#define DIAG_INTERRUPT_AD2CH5IF 198U

Interrupt flag for AD2CH5.

13.3.1.135. DIAG_INTERRUPT_AD2CMP5IF

#define DIAG_INTERRUPT_AD2CMP5IF 199U

Interrupt flag for AD2CMP5.

13.3.1.136. DIAG_INTERRUPT_AD2CH6IF

#define DIAG_INTERRUPT_AD2CH6IF 200U

Interrupt flag for AD2CH6.

13.3.1.137. DIAG_INTERRUPT_AD2CMP6IF

#define DIAG_INTERRUPT_AD2CMP6IF 201U

Interrupt flag for AD2CMP6.

13.3.1.138. DIAG_INTERRUPT_AD2CH7IF

#define DIAG_INTERRUPT_AD2CH7IF 202U

Interrupt flag for AD2CH7.

13.3.1.139. DIAG_INTERRUPT_AD2CMP7IF

#define DIAG_INTERRUPT_AD2CMP7IF 203U

Interrupt flag for AD2CMP7.

13.3.1.140. DIAG_INTERRUPT_CLC1PIF

#define DIAG_INTERRUPT_CLC1PIF 232U

Interrupt flag for CLC1P.

13.3.1.141. DIAG_INTERRUPT_CLC1NIF

#define DIAG_INTERRUPT_CLC1NIF 233U

Interrupt flag for CLC1N.

13.3.1.142. DIAG_INTERRUPT_CLC2NIF

#define DIAG_INTERRUPT_CLC2NIF 235U

Interrupt flag for CLC2N.

13.3.1.143. DIAG_INTERRUPT_CLC3PIF

#define DIAG_INTERRUPT_CLC3PIF 236U

Interrupt flag for CLC3P.

13.3.1.144. DIAG_INTERRUPT_CLC3NIF

#define DIAG_INTERRUPT_CLC3NIF 237U

Interrupt flag for CLC3N.

13.3.1.145. DIAG_INTERRUPT_CLC4PIF

#define DIAG_INTERRUPT_CLC4PIF 238U

Interrupt flag for CLC4P.

13.3.1.146. DIAG_INTERRUPT_CLC4NIF

#define DIAG_INTERRUPT_CLC4NIF 239U

Interrupt flag for CLC4N.

13.3.1.147. DIAG_INTERRUPT_IOM1IF

#define DIAG_INTERRUPT_IOM1IF 275U

Interrupt flag for IOM1.

13.3.1.148. DIAG_INTERRUPT_IOM2IF

#define DIAG_INTERRUPT_IOM2IF 276U

Interrupt flag for IOM2.

13.3.1.149. DIAG_INTERRUPT_IOM3IF

#define DIAG_INTERRUPT_IOM3IF 277U

Interrupt flag for IOM3.

13.3.1.150. DIAG_INTERRUPT_IOM4IF

#define DIAG_INTERRUPT_IOM4IF 278U

Interrupt flag for IOM4.

13.3.1.151. DIAG_INVALID_MODULE_ID

#define DIAG_INVALID_MODULE_ID 0xFFFFU

Invalid module ID.

13.3.1.152. DIAG_INVALID_MODULE_INDEX

#define DIAG_INVALID_MODULE_INDEX (DIAG_INVALID_MODULE_ID - 1U)

Invalid module index.

13.3.1.153. DIAG_INTERRUPT_EXTR_TEST_COMPLETED

#define DIAG_INTERRUPT_EXTR_TEST_COMPLETED 0xAAAAAAAAU

Diagnostic interrupt flag indicating that the external test has been completed.

13.3.1.154. DIAG_INTERRUPT_EXTR_TEST_IN_PROGRESS

#define DIAG_INTERRUPT_EXTR_TEST_IN_PROGRESS 0x55555555U

Diagnostic interrupt flag indicating that the external test is in progress.

13.3.1.155. DIAG_INTERRUPT_ISR_BIT_CLEARED_TRUE

#define DIAG_INTERRUPT_ISR_BIT_CLEARED_TRUE 0x77777777U

Diagnostic interrupt isr bit clear true.

13.3.1.156. DIAG_INTERRUPT_ISR_BIT_CLEARED_FALSE

#define DIAG_INTERRUPT_ISR_BIT_CLEARED_FALSE 0x88888888U

Diagnostic interrupt isr bit clear false.

13.3.1.157. INTERRUPT_PRIORITY_MASK_ZERO

#define INTERRUPT_PRIORITY_MASK_ZERO 0xFFF8U

Defines the macro for the interrupt priority mask. Each interrupt has an eight level priority. The value of interrupt priority is set to 0, to inject the fault for the interrupt tests.

13.3.1.158. INTERRUPT_PRIORITY_MASK_ONE

#define INTERRUPT_PRIORITY_MASK_ONE 0xFF8FU

Defines the macro for the interrupt priority mask. Each interrupt has an eight level priority. The value of interrupt priority is set to 0, to inject the fault for the interrupt tests.

13.3.1.159. INTERRUPT_PRIORITY_MASK_TWO

#define INTERRUPT_PRIORITY_MASK_TWO 0xF8FFU

Defines the macro for the interrupt priority mask. Each interrupt has an eight level priority. The value of interrupt priority is set to 0, to inject the fault for the interrupt tests.

13.3.1.160. INTERRUPT_PRIORITY_MASK_THREE

#define INTERRUPT_PRIORITY_MASK_THREE 0x8FFFU

Defines the macro for the interrupt priority mask. Each interrupt has an eight level priority. The value of interrupt priority is set to 0, to inject the fault for the interrupt tests.

13.3.1.161. INTERRUPT_IPC_BIT_POS_OFFSET

#define INTERRUPT_IPC_BIT_POS_OFFSET 4U

Macro defined to set the bit position of the interrupt priority of the irqnumber in the IPC register.

13.4. Enums

13.4.1. Enumeration Type Documentation

13.4.1.1. DIAG_INTERRUPT_SERVICING_RUN_STATUS

enum DIAG_INTERRUPT_SERVICING_RUN_STATUS

Enumeration for diagnostic interrupt servicing run status.

This enumeration defines the possible states for diagnostic interrupt servicing.

13.4.1.2. DIAG_INTERRUPT_HARDTRAP_RUN_STATUS

enum DIAG_INTERRUPT_HARDTRAP_RUN_STATUS

Enumeration for the diagnostic interrupt hard trap run status.

This enumeration defines the possible states for the diagnostic interrupt hard trap run status.

13.4.1.3. DIAG_INTERRUPT_ISRCLEAREDCHECK_RUN_STATUS

enum DIAG_INTERRUPT_ISRCLEAREDCHECK_RUN_STATUS

Enumeration for interrupt ISR cleared check run status.

This enumeration defines the possible statuses for checking if the ISR has been cleared.

13.4.1.4. DIAG_INTERRUPT_ISR_FLAG_CHECK_STATUS

enum DIAG_INTERRUPT_ISR_FLAG_CHECK_STATUS

Enumeration for interrupt ISR flag check status.

This enum defines the possible statuses for the interrupt ISR flag check.

13.4.1.5. DIAG_INTERRUPT_EXTERNAL_INTR_NUM

enum DIAG_INTERRUPT_EXTERNAL_INTR_NUM

Enumeration for external interrupt numbers.

This enum defines the possible external interrupt numbers.

13.4.1.6. DIAG_INTERRUPT_POLARITY

enum DIAG_INTERRUPT_POLARITY

Enumeration for interrupt polarity.

This enum defines the possible polarities for interrupts.

13.4.1.7. DIAG_INTERRUPT_FREQ_RUN_STATUS

enum DIAG_INTERRUPT_FREQ_RUN_STATUS

Enumeration for interrupt frequency diagnostic run status.

This enum defines the possible statuses for the interrupt frequency diagnostic run.

13.4.1.8. DIAG_INTERRUPT_EXT_INTR_RUN_STATUS

enum DIAG_INTERRUPT_EXT_INTR_RUN_STATUS

Enumeration for external interrupt diagnostic run status.

This enum defines the possible statuses for the external interrupt diagnostic run.

13.4.1.9. DIAG_INTERRUPT_IRQ_STATUS

enum DIAG_INTERRUPT_IRQ_STATUS

Enumeration for interrupt IRQ status.

This enum defines the possible statuses for the interrupt IRQ.

13.5. Data Structure Documentation

13.5.1. DIAG_INTERRUPT_IRQ_MODEL Struct Reference

Structure for interrupt IRQ model.

13.5.1.1. Detailed Description

Structure for interrupt IRQ model.

This structure defines the model for interrupt IRQ, including various counters and status indicators.

13.5.1.1.1. Data Fields

- uint32_t [moduleIrq](#)
- int32_t [moduleInterruptCounter](#)
- int32_t [moduleExpectedInterruptCount](#)
- int32_t [moduleInterruptToleranceBand](#)
- uint32_t [moduleInterruptFTTI](#)
- uint32_t [moduleInterruptFTTICounter](#)
- [DIAG_INTERRUPT_IRQ_STATUS](#) [moduleIntrStatus](#)

13.5.1.2. Field Documentation

13.5.1.2.1. moduleIrq

DIAG_INTERRUPT_IRQ_MODEL::moduleIrq

The IRQ number for the module.

13.5.1.2.2. moduleInterruptCounter

DIAG_INTERRUPT_IRQ_MODEL::moduleInterruptCounter

Counter for the number of interrupts that have occurred.

13.5.1.2.3. moduleExpectedInterruptCount

DIAG_INTERRUPT_IRQ_MODEL::moduleExpectedInterruptCount

Expected number of interrupts.

13.5.1.2.4. moduleInterruptToleranceBand

DIAG_INTERRUPT_IRQ_MODEL::moduleInterruptToleranceBand

Tolerance band for the interrupt count.

13.5.1.2.5. moduleInterruptFTTI

DIAG_INTERRUPT_IRQ_MODEL::moduleInterruptFTTI

First Time To Interrupt (FTTI) value.

13.5.1.2.6. moduleInterruptFTTICounter

DIAG_INTERRUPT_IRQ_MODEL::moduleInterruptFTTICounter

Counter for the FTTI.

13.5.1.2.7. moduleIntrStatus

DIAG_INTERRUPT_IRQ_MODEL::moduleIntrStatus

Status of the interrupt IRQ.

13.5.2. DIAG_INTERRUPT_MODULE_STATUS Struct Reference

Structure for interrupt module status.

13.5.2.1. Detailed Description

Structure for interrupt module status.

This structure defines the status of an interrupt module, including the IRQ number, interrupt status, and interrupt counter.

13.5.2.1.1. Data Fields

- uint32_t [IrqNumber](#)

- [DIAG_INTERRUPT_IRQ_STATUS interruptStatus](#)
- [int32_t intrCounter](#)

13.5.2.2. Field Documentation

13.5.2.2.1. IrqNumber

DIAG_INTERRUPT_MODULE_STATUS::IrqNumber

The IRQ number for the interrupt module.

13.5.2.2.2. interruptStatus

DIAG_INTERRUPT_MODULE_STATUS::interruptStatus

The status of the interrupt.

13.5.2.2.3. intrCounter

DIAG_INTERRUPT_MODULE_STATUS::intrCounter

Counter for the number of interrupts.

13.6. Source Code Reference

The following table provides a list of APIs and their location in the source code:

MODULE	DIAGNOSTIC API	SOURCE CODE REFERENCE
INTERRUPT	• DIAG_INTERRUPT_ExternalGetIntrTestStatus • DIAG_INTERRUPT_TriggerAndTestExternalInterrupts • DIAG_INTERRUPT_ExternalSetIntrTestCompleted • DIAG_INTERRUPT_IsExtIntrTestRunning • DIAG_INTERRUPT_ExternalInputTest	diag_interrupt_external_input.c
INTERRUPT	• DIAG_INTERRUPT_StartFrequencyTest • DIAG_INTERRUPT_UpdateISRNotifications • DIAG_INTERRUPT_FrequencyCheck • DIAG_INTERRUPT_CheckModuleInterruptCounter • DIAG_INTERRUPT_IsInterruptFreqTestRunning • DIAG_INTERRUPT_GetFreqTestResults • DIAG_INTERRUPT_StopFrequencyTest	diag_interrupt_frequency_check.c
INTERRUPT	• DIAG_INTERRUPT_isTrapInducedByDiag • DIAG_INTERRUPT_HardTrapTest • DIAG_INTERRUPT_resetDiagInducedTrapStatus	diag_interrupt_hardtrap_test.c
INTERRUPT	• DIAG_INTERRUPT_IsFlagClearedCheckFromIsr • DIAG_INTERRUPT_IsrClearedCheck	diag_interrupt_isrcleared_check.c
INTERRUPT	• DIAG_INTERRUPT_IsInterruptServicingCheckRunning • DIAG_INTERRUPT_ServicingTest • DIAG_INTERRUPT_CheckIFSRegister • DIAG_INTERRUPT_GetifsFlagRegStatus	diag_interrupt_servicing_test.c
INTERRUPT	• FAULTINJ_INTERRUPT_ExternalInterrupt • FAULTINJ_INTERRUPT_ServicingCheck	faultinj_interrupt.c

14. PC

14.1. Overview

This document describes the diagnostics API for the PC module. The safety requirements described for PC diagnostics in the Safety Manual are implemented in the following APIs.

- [DIAG_PC_ProgramCounterTest](#)

Constraints/Limitations:

- Nil

Integration Rules:

- Nil

14.2. Functions

14.2.1. Function Documentation

14.2.1.1. DIAG_PC_ProgramCounterTest()

uint32_t DIAG_PC_ProgramCounterTest (void)

Software Requirement Reference ID : SW_PROGRAM_COUNTER_TEST

This diagnostic API verifies the program counter as it progresses through the code. The API repeatedly calls three internal helper functions to indicate that those functions have been touched, a counter variable is updated and the updated value is verified at the end of the API for the correct value to indicate a PASS/FAIL.

Parameters:

void

Returns:

DIAG_PASS
DIAG_FAIL

Figure 14-1. PC Program Counter Test Sequence Diagram

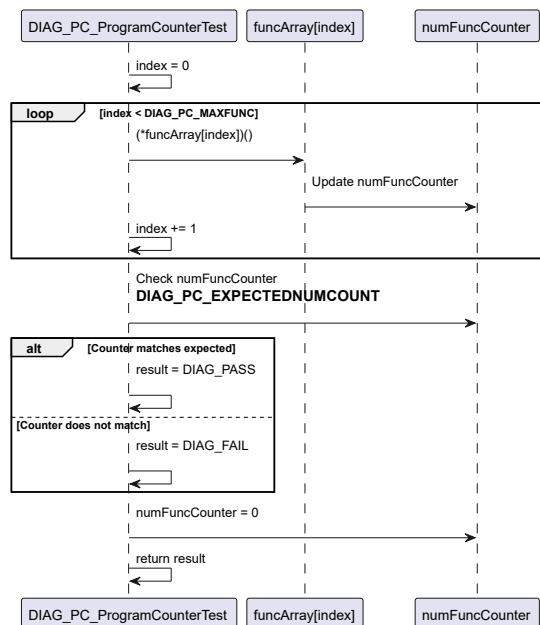
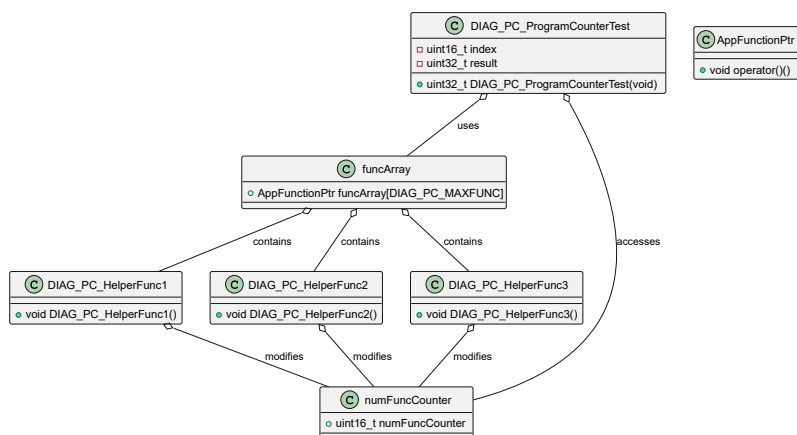


Figure 14-2. PC Program Counter Test Class Diagram



14.3. Source Code Reference

The following table provides a list of APIs and their location in the source code:

MODULE	DIAGNOSTIC API	SOURCE CODE REFERENCE
PC	- DIAG_PC_HelperFunc3 - DIAG_PC_HelperFunc1 - DIAG_PC_ProgramCounterTest - DIAG_PC_HelperFunc2	diag_pc_programcountertest.c

15. SRAM

15.1. Overview

This document describes the diagnostics API for the SRAM module. The safety requirements described for SRAM module diagnostics in the Safety Manual are implemented in the following APIs.

- [DIAG_SRAM_SetSramSingleBitIsrEntryStatus](#)
- [DIAG_SRAM_SetSramDoubleBitTrapEntryStatus](#)
- [DIAG_SRAM_SingleDoubleErrorDetectionTest](#)
- [DIAG_SRAM_ReplicationWrite](#)
- [DIAG_SRAM_IsBackedUpDataValid](#)

Constraints/Limitations:

None

Integration Rules:

None

15.2. Functions

15.2.1. Function Documentation

15.2.1.1. DIAG_SRAM_SetSramSingleBitIsrEntryStatus()

void DIAG_SRAM_SetSramSingleBitIsrEntryStatus (uint32_t status)

Sets the entry status for the SRAM single-bit error ISR.

This function updates the status variable that tracks the entry state of the single-bit error interrupt service routine (ISR) for SRAM.

Parameters:

in	status	Status value to set (e.g., DIAG_SRAM_SINGLE_BIT_ERROR_ISR_WAITING or DIAG_SRAM_SINGLE_BIT_ERROR_ISR_TRIGGERED).
----	--------	---

15.2.1.2. DIAG_SRAM_SetSramDoubleBitTrapEntryStatus()

void DIAG_SRAM_SetSramDoubleBitTrapEntryStatus (uint32_t status)

Sets the entry status for the SRAM double-bit error trap.

This function updates the status variable that tracks the entry state of the double-bit error trap for SRAM.

Parameters:

in	status	Status value to set (e.g., DIAG_SRAM_DOUBLE_BIT_TRAP_WAITING or DIAG_SRAM_DOUBLE_BIT_TRAP_ENTERED).
----	--------	---

15.2.1.3. DIAG_SRAM_SingleDoubleErrorDetectionTest()

uint32_t DIAG_SRAM_SingleDoubleErrorDetectionTest (uint8_t errorType, uint32_t * sramFaultAddressPtr, uint8_t faultBitValue)

Software Requirement Reference ID : SW_SRAM_ECC_EVENT_INTERRUPT_TEST_01 **Software Requirement Reference ID : SW_SRAM_ECC_SINGLE_DOUBLE_ERROR_DETECTION_TEST_01** Test and handle an ECC event interrupt in SRAM.

This function simulates or handles an ECC event interrupt for SRAM by invoking the appropriate error handler based on the error type and fault bit value.

Parameters:

in	errorType	Type of ECC error (e.g., single or double bit).
in	sramFaultAddressPtr	Pointer to the address where the fault occurred.
in	faultBitValue	The value of the fault bit.

Returns:

Returns a status code indicating the result of the operation. The value is set by DIAG_SRAM_SetRetVal() and retrieved by DIAG_SRAM_GetRetVal().

Figure 15-1. SRAM Single Double Error Detection Sequence Diagram

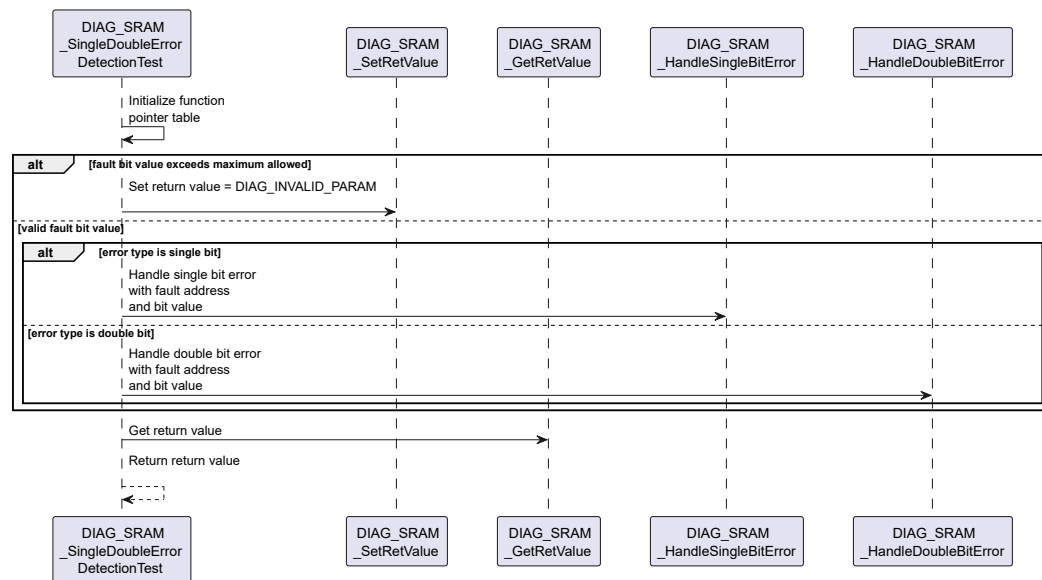
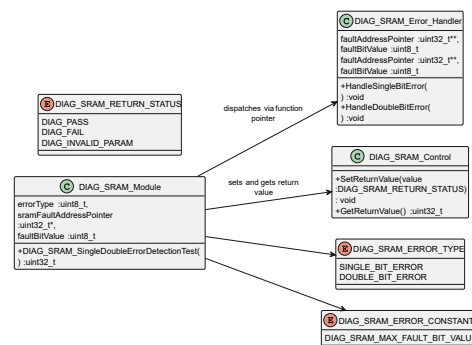


Figure 15-2. SRAM Single Double Error Detection Activity Diagram



15.2.1.4. DIAG_SRAM_ReplicationWrite()

uint32_t DIAG_SRAM_ReplicationWrite (const uint32_t srcAddress[], uint32_t backUpAddress[], uint8_t size)

Software Requirement Reference ID : SW_SRAM_REPLICATION_PRACTICE_01Replicates and inverts a block of SRAM data to a backup location.

This function copies a block of data from the source address to the backup address, inverting each 32-bit word during the copy. It performs checks for null pointers, valid size, and overlapping memory regions before proceeding.

Parameters:

in	srcAddress	Pointer to the source SRAM memory block.
out	backUpAddress	Pointer to the destination (backup) SRAM memory block.
in	size	Number of 32-bit words to replicate and invert.

Return values:

DIAG_PASS	Replication and inversion successful.
DIAG_FAIL	Source and backup regions overlap.
DIAG_INVALID_PARAM	Invalid parameters (null pointers or invalid size).

Returns:

Status code indicating the result of the operation, as set by DIAG_SRAM_SetRetVal() and retrieved by DIAG_SRAM_GetRetVal().

Figure 15-3. SRAM Replication write Sequence Diagram

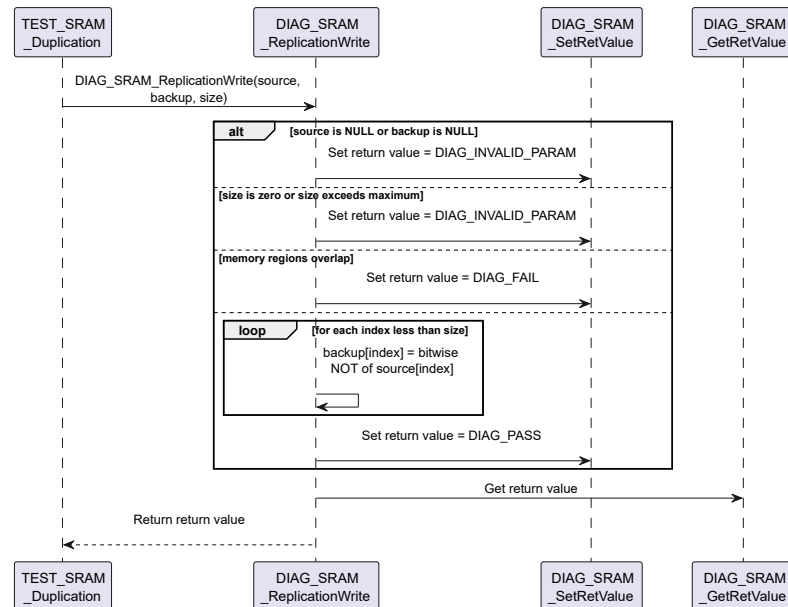
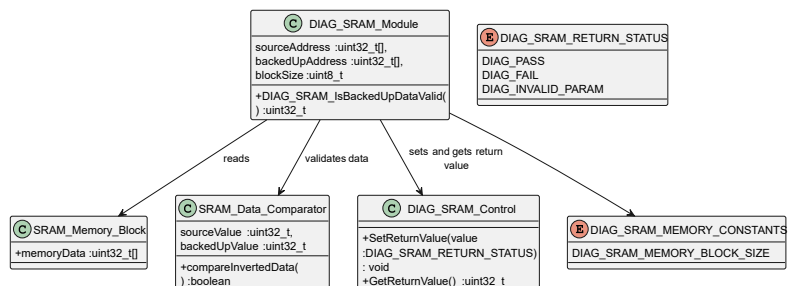


Figure 15-4. SRAM Replication Write Class Diagram



15.2.1.5. DIAG_SRAM_IsBackedUpDataValid()

uint32_t DIAG_SRAM_IsBackedUpDataValid (const uint32_t srcAddress[], const uint32_t backedUpAddress[], uint8_t size)

Validates that the backed-up SRAM data is the bitwise inverse of the source data.

This function checks whether each 32-bit word in the backup memory block is the bitwise inverse of the corresponding word in the source memory block. It performs parameter validation and returns a status code indicating the result.

Parameters:

in	srcAddress	Pointer to the source SRAM memory block.
in	backedUpAddress	Pointer to the backed-up (inverted) SRAM memory block.
in	size	Number of 32-bit words to validate.

Return values:

DIAG_PASS	All backed-up data is a valid bitwise inverse of the source data.
DIAG_FAIL	At least one word in the backup is not the bitwise inverse of the source.
DIAG_INVALID_PARAM	Invalid parameters (null pointers or invalid size).

Returns:

Status code indicating the result of the validation, as set by DIAG_SRAM_SetRetValue() and retrieved by DIAG_SRAM_GetRetValue().

Figure 15-5. SRAM Backed Up Data Valid Sequence Diagram

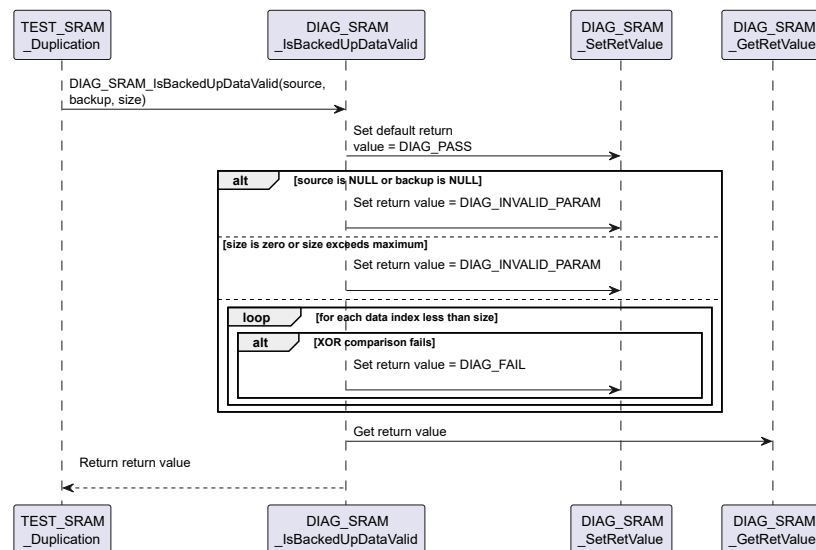
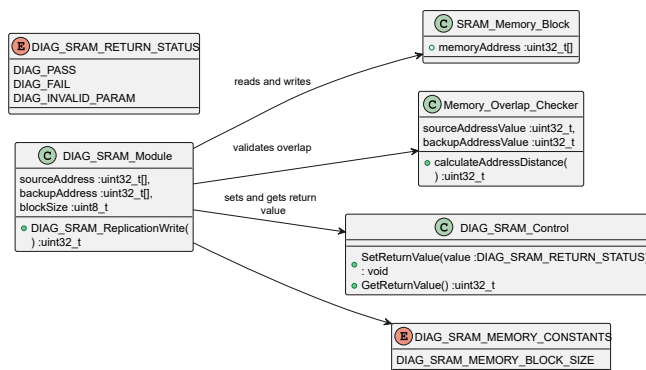


Figure 15-6. SRAM Backed Up Data Valid Class Diagram



15.2.1.6. FAULTINJ_SRAM_Duplication()

void FAULTINJ_SRAM_Duplication (uint32_t * address, uint32_t faultbitpos)

Software Requirement Reference ID : SW_SRAM_DUPLICATION_01 This API injects the fault for the SRAM duplication diagnostic functions.

Parameters:

address	the SRAM location where the fault is injected.
faultbitpos	the bit position in the SRAM word where the fault is injected.

Returns:

void

Code Snippet:

```

uint16_t TEST_SRAM_Duplication(uint8_t testOption)
{
    uint16_t result = 0x00U;
    switch(testOption)
    {
        ...
        case INJECT_FAULT_TRUE:
            result = DIAG_SRAM_DuplicateWrite((uint8_t*)dupsrc, (uint8_t*)dupdest,
            sizeof(dupsrc));
            FAULTINJ_SRAM_Duplication((uint8_t*)dupsrc, 5U);
            result = DIAG_SRAM_IsDuplicateCheckValid((uint8_t*)dupsrc, (uint8_t*)dupdest,
            sizeof(dupsrc));
            break;
    }
}

```

15.3. Macros

15.3.1. Definition Documentation

15.3.1.1. DIAG_SRAM_SINGLE_BIT_ERROR_ISR_WAITING

#define DIAG_SRAM_SINGLE_BIT_ERROR_ISR_WAITING 0x55555555U

Status code indicating the ISR is waiting for a single-bit error event in SRAM.

15.3.1.2. DIAG_SRAM_SINGLE_BIT_ERROR_ISR_TRIGGERED

#define DIAG_SRAM_SINGLE_BIT_ERROR_ISR_TRIGGERED 0xAAAAAAAAU

Status code indicating the ISR has been triggered by a single-bit error event in SRAM.

15.3.1.3. DIAG_SRAM_DOUBLE_BIT_TRAP_WAITING

```
#define DIAG_SRAM_DOUBLE_BIT_TRAP_WAITING 0xCCCCCCCCU
```

Status code indicating the system is waiting for a double-bit error trap event in SRAM.

15.3.1.4. DIAG_SRAM_DOUBLE_BIT_TRAP_ENTERED

```
#define DIAG_SRAM_DOUBLE_BIT_TRAP_ENTERED 0xBBBBBBBBU
```

Status code indicating the system has entered the double-bit error trap in SRAM.

15.4. Enums

15.4.1. Enumeration Type Documentation

15.4.1.1. DIAG_SRAM_eSramErrorEventType

```
enum DIAG_SRAM_eSramErrorEventType
```

Enumerates the types of SRAM ECC error events.

This enumeration defines the possible error event types that can occur in SRAM, such as single-bit and double-bit error events.

15.5. Source Code Reference

The following table provides a list of APIs and their location in the source code:

MODULE	DIAGNOSTIC API	SOURCE CODE REFERENCE
SRAM	- DIAG_SRAM_SetSramSingleBitSramEntryStatus - DIAG_SRAM_SetSramDoubleBitTrapEntryStatus - DIAG_SRAM_SingleDoubleErrorDetectionTest - DIAG_SRAM_HandleSingleBitError - DIAG_SRAM_HandleDoubleBitError	diag_sram_ecc_event.c
SRAM	- DIAG_SRAM_IsBackedUpDataValid - DIAG_SRAM_ReplicationWrite	diag_sram_replication.c
SRAM	FAULTINJ_SRAM_Duplication	faultinj_sram.c

16. TIMER

16.1. Overview

This document describes the diagnostics for the Timer module. The safety requirements described for Timer diagnostics in the Safety Manual are implemented by the following APIs.

- [DIAG_TIMER_FunctionalTest](#)
- [DIAG_TIMER_LinearityTest](#)

Constraints/Limitations:

- Some microcontrollers have one instance of the Timer module and maybe a few CCP (Capture/Compare/Timer) instances. The diagnostic functions therefore operate on one instance of a Timer (or CCP as a timer) to detect Functional and Linearity issues.

Integration Rules:

- Nil

16.2. Functions

16.2.1. Function Documentation

16.2.1.1. DIAG_TIMER_FunctionalTest()

uint32_t DIAG_TIMER_FunctionalTest (DIAG_UTILITY_TIMER_INSTANCE tmrId, uint32_t tolerance)

Software Requirement Reference ID : SW_TIMER_FUNCTIONAL_TEST_02 Implements the Timer functional diagnostic to check the operation of a given timer instance. The diagnostic indicates when the Timer runs at a frequency different from the one intended and when timer interrupts are not received

Parameters:

tmrId	- Timer instance used by the timer diagnostic
tolerance	- tolerance value to determine the band within which the timer interrupt count must fall for the test to pass. The value represents by how many counts, the timer is allowed to deviate.

Returns:

DIAG_PASS
DIAG_FAIL
DIAG_INVALID_PARAM

Figure 16-1. TIMER Functional Test

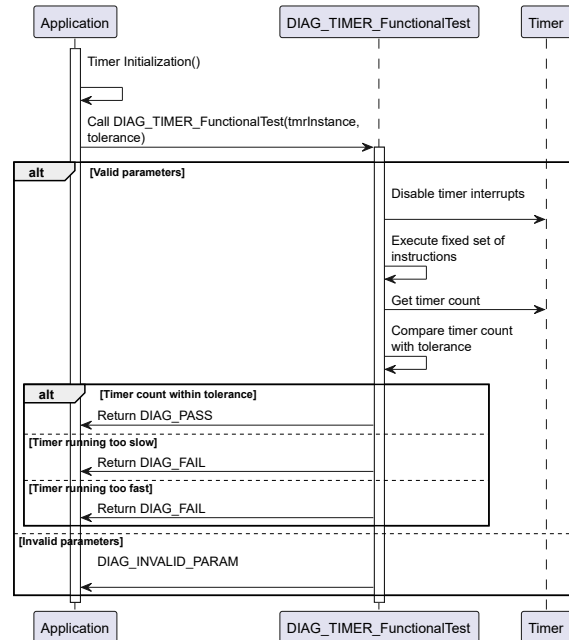
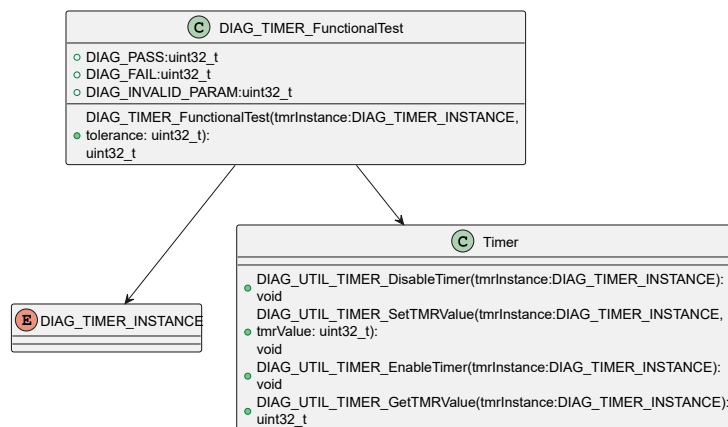


Figure 16-2. TIMER Functional Test Class Diagram



16.2.1.2. DIAG_TIMER_LinearityTest()

uint32_t DIAG_TIMER_LinearityTest (DIAG_UTILITY_TIMER_INSTANCE tmrId, uint32_t tolerance)

Software Requirement Reference ID : SW_TIMER_LINEARITY_TEST_01 Inaccuracies, if any, in the TMRx count register are diagnosed by this function. The previously recorded TMRx difference value is used as a reference to check if any errors have crept into the Timer period base.

Parameters:

tmrId	- Timer instance used by the timer diagnostic
tolerance	- Tolerance is in counts which specifies, maximum allowable deviation between two measured timer counts during a diagnostic test.

Returns:

DIAG_PASS
DIAG_FAIL
DIAG_INVALID_PARAM

Figure 16-3. TIMER Linearity Test

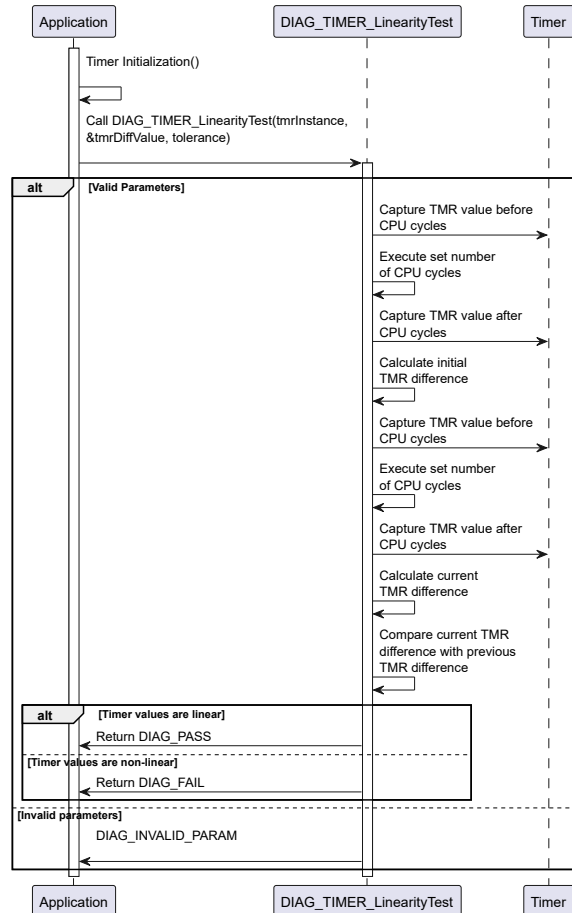
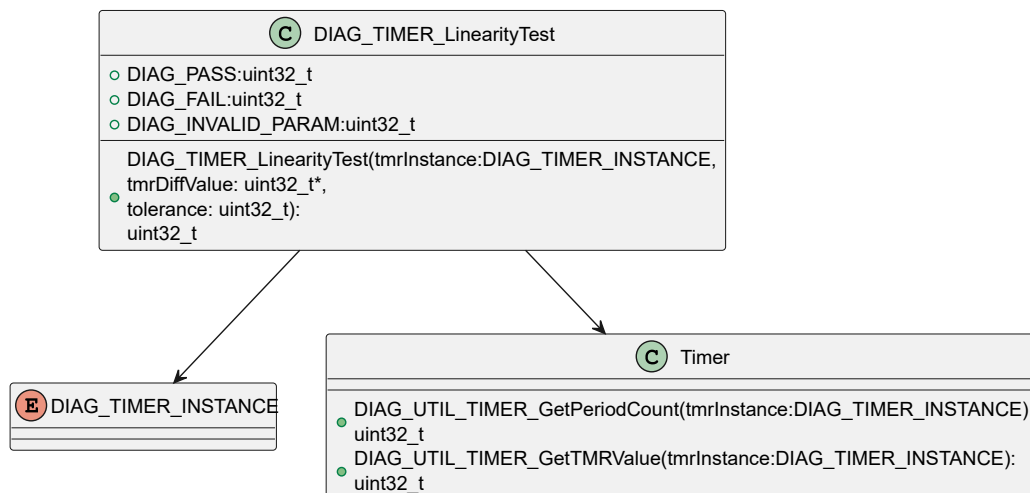


Figure 16-4. TIMER Linearity Test Class Diagram



16.2.1.3. DIAG_TIMER_GetResults()

DIAG_TIMER_FUNCTIONAL_RESULTS DIAG_TIMER_GetResults (void)

Software Requirement Reference ID : SW_TIMER_FUNCTIONAL_TEST_02The function returns the result of the timer functional diagnostics.

Parameters:

Nil

Returns:

DIAG_TIMER_FUNCTIONAL_RESULTS

16.2.1.4. FAULTINJ_TIMER_SetFaultInjFunctional()

void FAULTINJ_TIMER_SetFaultInjFunctional (DIAG_TIMER_FAULTINJ_SCENARIOS setFaultInj)

Software Requirement Reference ID : SW_TIMER_FUNCTIONAL_TEST_02Implements the fault injection for functional diagnostics.

Parameters:

setFaultInj - Fault mode

Returns:

void

16.2.1.5. FAULTINJ_TIMER_SetFaultInjLinearity()

void FAULTINJ_TIMER_SetFaultInjLinearity (DIAG_TIMER_FAULTINJ_SCENARIOS setFaultInj)

Software Requirement Reference ID : SW_TIMER_LINEARITY_TEST_01Implements the fault injection for linearity diagnostics.

Parameters:

setFaultInj - Fault mode

Returns:

void

16.3. Enums

16.3.1. Enumeration Type Documentation

16.3.1.1. DIAG_TIMER_FUNCTIONAL_RESULTS

enum DIAG_TIMER_FUNCTIONAL_RESULTS

Defines the functional diagnostic results.

16.3.1.2. DIAG_TIMER_FAULTINJ_SCENARIOS

enum DIAG_TIMER_FAULTINJ_SCENARIOS

Defines the fault injection scenarios for timer diagnostics.

16.4. Source Code Reference

The following table provides a list of APIs and their location in the source code:

MODULE	DIAGNOSTIC API	SOURCE CODE REFERENCE
TIMER	• DIAG_TIMER_FunctionalRestoreValues • DIAG_TIMER_FunctionalTest • DIAG_TIMER_GetResults • DIAG_TIMER_FunctionalStoreValues • DIAG_TIMER_SetTestModeFunctional • DIAG_TIMER_SetResults • DIAG_TIMER_FunctionalSetValues	diag_timer_functional_test.c
TIMER	• DIAG_TIMER_LinearitySetValues • DIAG_TIMER_SetTestModeLinearity • DIAG_TIMER_GetCountElapsed • DIAG_TIMER_GetDeviations • DIAG_TIMER_LinearityTest • DIAG_TIMER_LinearityRestoreValues • DIAG_TIMER_LinearityStoreValues	diag_timer_linearity_test.c
TIMER	• FAULTINJ_TIMER_SetFaultInjFunctional • FAULTINJ_TIMER_SetFaultInjLinearity	faultinj_timer.c

17. COMMON

17.1. Overview

This document contains the API descriptions for the common utility functions.

17.2. Functions

17.2.1. Function Documentation

17.2.1.1. DIAG_SetClockFreq()

void DIAG_SetClockFreq (uint32_t clockfreq)

Sets the system clock frequency for delay calculations.

Parameters:

clockfreq	System clock frequency in Hz.
-----------	-------------------------------

Stores the provided clock frequency and calculates the number of instruction cycles required for 1 microsecond and 1 millisecond delays. These values are used by the delay functions to generate accurate timing.

17.2.1.2. DIAG_delayMs()

void DIAG_delayMs (uint32_t msdelay)

Delays execution for a specified number of milliseconds.

Parameters:

msdelay	Number of milliseconds to delay.
---------	----------------------------------

Multiplies the millisecond delay value by the pre-calculated instruction count for one millisecond, then calls [DIAG_delayInstrCount\(\)](#) to perform the actual delay using a repeat-nop loop.

17.2.1.3. DIAG_delayUs()

void DIAG_delayUs (uint32_t usdelay)

Delays execution for a specified number of microseconds.

Parameters:

usdelay	Number of microseconds to delay.
---------	----------------------------------

Multiplies the microsecond delay value by the pre-calculated instruction count for one microsecond, then calls [DIAG_delayInstrCount\(\)](#) to perform the actual delay using a repeat-nop loop.

17.2.1.4. DIAG_delayInstrCount()

void DIAG_delayInstrCount (uint32_t count)

Delays execution for a specified number of instruction cycles.

Parameters:

count	Number of instruction cycles to delay.
-------	--

Uses an inline assembly loop to execute a "repeat-nop" instruction sequence for the specified number of cycles, providing a precise delay based on the system clock.

17.2.1.5. DIAG_GetClockFreq()

uint32_t DIAG_GetClockFreq (void)

Gets the current system clock frequency.

Returns:

Current system clock frequency in Hz.

Returns the value stored by [DIAG_SetClockFreq\(\)](#), representing the clock frequency used for delay calculations.

17.2.1.6. DIAG_UTIL_ADC_RedundantInputsPractice()

uint32_t DIAG_UTIL_ADC_RedundantInputsPractice (DIAG_ADC_CHANNELS adcFirstChannelNumber, DIAG_ADC_CHANNELS adcSecondChannelNumber, uint32_t tolerance)

**** DIAG_UTIL_ADC_RedundantInputsPractice ****Implements the function which returns comparison of results from two ADC channels

Parameters:

adcFirstChannelNumber	- First Channel Number
adcSecondChannelNumber	- Second Channel Number
tolerance	- Maximum allowable difference (deviation) between the ADC results of two channels

Returns:

[DIAG_PASS](#)
[DIAG_FAIL](#)
[DIAG_INVALID_PARAM](#)

17.2.1.7. DIAG_UTIL_DAC_GenerateDC()

void DIAG_UTIL_DAC_GenerateDC (uint16_t outValue)

**** DIAG_UTIL_DAC_GenerateDC ****This function is used to change the DC output voltage generated by DAC.

Parameters:

outValue	- Value corresponding to the required DC voltage.
----------	---

Returns:

void

17.2.1.8. DIAG_UTIL_DAC_ConfigureDC()

void DIAG_UTIL_DAC_ConfigureDC (void)

**** DIAG_UTIL_DAC_ConfigureDC ****Implements the function which configures DAC to generate DC steady signal.

Parameters:

void

Returns:

void

17.2.1.9. DIAG_UTIL_TIMER_GetPeriodCount()

uint32_t DIAG_UTIL_TIMER_GetPeriodCount (DIAG_UTILITY_TIMER_INSTANCE tmrInstance)

**** DIAG_UTIL_TIMER_GetPeriodCount ****Implements the function which returns value of the period count set for the timer instance specified

Parameters:

tmrInstance	- Timer ID from the instances available on the device
-------------	---

Returns:

- Period count value set for the timer module

17.2.1.10. DIAG_UTIL_TIMER_GetTMRValue()

uint32_t DIAG_UTIL_TIMER_GetTMRValue (DIAG_UTILITY_TIMER_INSTANCE tmrInstance)

**** DIAG_UTIL_TIMER_GetTMRValue ****Implements the function which returns value of the TMRx register for the timer instance specified

Parameters:

tmrInstance	- Timer ID from the instances available on the device
-------------	---

Returns:

- TMRx value for the timer instance specified

17.2.1.11. DIAG_UTIL_TIMER_GetPrescaler()

uint32_t DIAG_UTIL_TIMER_GetPrescaler (DIAG_UTILITY_TIMER_INSTANCE tmrInstance)

**** DIAG_UTIL_TIMER_GetPrescaler ****Implements the function which returns value of the prescaler settings for the timer instance specified

Parameters:

tmrInstance	- Timer ID from the instances available on the device
-------------	---

Returns:

- Prescaler value for the timer instance specified
--

17.2.1.12. DIAG_UTIL_TIMER_SetPeriodCount()

void DIAG_UTIL_TIMER_SetPeriodCount (DIAG_UTILITY_TIMER_INSTANCE tmrInstance, uint32_t prVal)

**** DIAG_UTIL_TIMER_SetPeriodCount ****Implements the function which sets the period count for the timer instance specified

Parameters:

tmrInstance	- Timer ID from the instances available on the device
prVal	- Value of the period to be set for the timer instance specified

Returns:

void

17.2.1.13. DIAG_UTIL_TIMER_SetPrescaler()

void DIAG_UTIL_TIMER_SetPrescaler (DIAG_UTILITY_TIMER_INSTANCE tmrInstance, uint32_t preScaleSetting)

**** DIAG_UTIL_TIMER_SetPrescaler ****Implements the function which sets the prescaler for the timer instance specified

Parameters:

tmrInstance	- Timer ID from the instances available on the device
preScaleSetting	- One of the available prescaler settings for the timer

Returns:

void

17.2.1.14. DIAG_UTIL_TIMER_SetTMRValue()

void DIAG_UTIL_TIMER_SetTMRValue (DIAG_UTILITY_TIMER_INSTANCE tmrInstance, uint32_t tmrVal)

**** DIAG_UTIL_TIMER_SetTMRValue ****Implements the function which sets the TMRx value for the timer instance specified

Parameters:

tmrInstance	- Timer ID from the instances available on the device
tmrVal	- Value to be set on the TMRx register

Returns:

void

17.2.1.15. DIAG_UTIL_TIMER_EnableTimer()

void DIAG_UTIL_TIMER_EnableTimer (DIAG_UTILITY_TIMER_INSTANCE tmrInstance)

**** DIAG_UTIL_TIMER_EnableTimer ****Based on the Timer ID provided, this function enables that particular timer

Parameters:

tmrInstance	- Timer ID from the instances available on the device
-------------	---

Returns:

void

17.2.1.16. DIAG_UTIL_TIMER_DisableTimer()

void DIAG_UTIL_TIMER_DisableTimer (DIAG_UTILITY_TIMER_INSTANCE tmrInstance)

**** DIAG_UTIL_TIMER_DisableTimer ****Based on the Timer ID provided, this function disables that particular timer

Parameters:

tmrInstance	- Timer ID from the instances available on the device
-------------	---

Returns:

void

17.2.1.17. DIAG_UTIL_TIMER_DisableInterrupt()

void DIAG_UTIL_TIMER_DisableInterrupt (DIAG_UTILITY_TIMER_INSTANCE tmrInstance)

**** DIAG_UTIL_TIMER_DisableTimer ****Based on the Timer ID provided, this function disables that particular timer interrupt

Parameters:

tmrInstance	- Timer ID from the instances available on the device
-------------	---

Returns:

void

17.2.1.18. DIAG_UTIL_TIMER_EnableInterrupt()

void DIAG_UTIL_TIMER_EnableInterrupt (DIAG_UTILITY_TIMER_INSTANCE tmrInstance)

**** DIAG_UTIL_TIMER_EnableInterrupt ****Based on the Timer ID provided, this function enables that particular timer interrupt.

Parameters:

tmrInstance	- Timer ID from the instances available on the device
-------------	---

Returns:

void

17.2.1.19. DIAG_UTIL_TIMER_GetInterruptEnableStatus()

uint32_t DIAG_UTIL_TIMER_GetInterruptEnableStatus (DIAG_UTILITY_TIMER_INSTANCE tmrInstance)

**** DIAG_UTIL_TIMER_GetInterruptEnableStatus ****Based on the Timer ID provided, this function returns the interrupt status of that particular timer.

Parameters:

tmrInstance	- Timer ID from the instances available on the device
-------------	---

Returns:

- true if interrupt is enabled, false otherwise.
--

17.2.1.20. DIAG_UTIL_TIMER_GetInterruptFlagStatus()

uint32_t DIAG_UTIL_TIMER_GetInterruptFlagStatus (DIAG_UTILITY_TIMER_INSTANCE tmrInstance)

**** DIAG_UTIL_TIMER_GetInterruptFlagStatus ****Based on the Timer ID provided, this function returns the interrupt flag status of that particular timer.

Parameters:

tmrInstance	- Timer ID from the instances available on the device
-------------	---

Returns:

- true if interrupt flag is set, false otherwise.

17.3. Macros

17.3.1. Definition Documentation

17.3.1.1. DIAG_DELAY_MICROSECOND_DIVIDER

#define DIAG_DELAY_MICROSECOND_DIVIDER 1000000U

Divider value to convert clock frequency to microseconds.

Used to calculate the number of clock cycles in one microsecond based on the system clock frequency.

17.3.1.2. DIAG_DELAY_MILLISECOND_DIVIDER

#define DIAG_DELAY_MILLISECOND_DIVIDER 1000U

Divider value to convert clock frequency to milliseconds.

Used to calculate the number of clock cycles in one millisecond based on the system clock frequency.

17.3.1.3. DIAG_UTILITY_TIMER_TOTAL

#define DIAG_UTILITY_TIMER_TOTAL 5U

Total timers available.

17.3.1.4. DIAG_PASS

```
#define DIAG_PASS (uint32_t)(0x50415353) /*ASCII value of PASS*/
```

Indicates diagnostic PASS

17.3.1.5. DIAG_FAIL

```
#define DIAG_FAIL (uint32_t)(0x4641494C) /*ASCII value of FAIL*/
```

Indicates diagnostic FAIL

17.3.1.6. DIAG_INVALID_PARAM

```
#define DIAG_INVALID_PARAM (uint32_t)(0x4E564C44) /*ASCII value of NVLD*/
```

Indicates invalid parameters given to diagnostic API

17.3.1.7. DIAG_SET_CPU_IPL

```
#define DIAG_SET_CPU_IPL( ipl)
```

Value:

```
{ \
    DISIPLbits.DISIPL = ipl; \
    __builtin_nop(); \
} (void) 0
```

Macro to set CPU priority level to ipl.

17.3.1.8. DIAG_SET_AND_SAVE_CPU_IPL

```
#define DIAG_SET_AND_SAVE_CPU_IPL( save_to, ipl)
```

Value:

```
{ \
    save_to = DISIPLbits.DISIPL; \
    DIAG_SET_CPU_IPL(ipl); } (void) 0;
```

Macro to set and save CPU priority level.

17.3.1.9. DIAG_RESTORE_CPU_IPL

```
#define DIAG_RESTORE_CPU_IPL( saved_to) DIAG_SET_CPU_IPL(saved_to)
```

Macro to restore CPU priority level.

17.3.1.10. DIAG_INTERRUPT_PROTECT

```
#define DIAG_INTERRUPT_PROTECT( x)
```

Value:

```
{ \
    char saved_ipl; \
    \
    DIAG_SET_AND_SAVE_CPU_IPL(saved_ipl,7); \
    x; \
    DIAG_RESTORE_CPU_IPL(saved_ipl); } (void) 0;
```

Macro to handle critical section.

17.4. Enums

17.4.1. Enumeration Type Documentation

17.4.1.1. DIAG_UTILITY_TIMER_INSTANCE

```
enum DIAG_UTILITY_TIMER_INSTANCE
```

The Ids which refer to timers available in the device.

17.5. Global Variables

17.5.1. Variable Documentation

17.5.1.1. timerDataLookup

const DIAG_UTILITY_TIMER_DATA_LOOKUP timerDataLookup[DIAG_UTILITY_TIMER_TOTAL][extern]

This structure hold the look up data for all the available timer instances.

17.6. Data Structure Documentation

17.6.1. DIAG_UTILITY_TIMER_DATA_LOOKUP Struct Reference

17.6.1.1. Detailed Description

Maintains the lookup for the necessary addresses for an timer instance.

17.6.1.1.1. Data Fields

- uint32_t [baseAddress](#)
- uint32_t [periodOffsetAddress](#)
- uint32_t [timerOffsetAddress](#)
- uint32_t [prescalerOffsetAddress](#)
- uint32_t [prescalerMask](#)
- uint32_t [prescalerDataPosition](#)
- uint32_t [timerEnableOffsetAddress](#)
- uint32_t [timerEnableMask](#)
- uint32_t [interruptEnableAddress](#)
- uint32_t [interruptEnableMask](#)
- uint32_t [interruptFlagAddress](#)
- uint32_t [interruptFlagMask](#)

17.6.1.2. Field Documentation

17.6.1.2.1. baseAddress

DIAG_UTILITY_TIMER_DATA_LOOKUP::baseAddress

Base address of the timer.

17.6.1.2.2. periodOffsetAddress

DIAG_UTILITY_TIMER_DATA_LOOKUP::periodOffsetAddress

Period offset address of the timer.

17.6.1.2.3. timerOffsetAddress

DIAG_UTILITY_TIMER_DATA_LOOKUP::timerOffsetAddress

TMR offset address of the timer.

17.6.1.2.4. prescalerOffsetAddress

DIAG_UTILITY_TIMER_DATA_LOOKUP::prescalerOffsetAddress

Prescaler offset address of the timer.

17.6.1.2.5. prescalerMask

DIAG_UTILITY_TIMER_DATA_LOOKUP::prescalerMask

Prescaler mask of the timer.

17.6.1.2.6. prescalerDataPosition

DIAG_UTILITY_TIMER_DATA_LOOKUP::prescalerDataPosition

Prescaler data position of the timer.

17.6.1.2.7. timerEnableOffsetAddress

DIAG_UTILITY_TIMER_DATA_LOOKUP::timerEnableOffsetAddress

Enable offset address of the timer.

17.6.1.2.8. timerEnableMask

DIAG_UTILITY_TIMER_DATA_LOOKUP::timerEnableMask

Enable mask of the timer.

17.6.1.2.9. interruptEnableAddress

DIAG_UTILITY_TIMER_DATA_LOOKUP::interruptEnableAddress

Interrupt enable address of the timer.

17.6.1.2.10. interruptEnableMask

DIAG_UTILITY_TIMER_DATA_LOOKUP::interruptEnableMask

Interrupt enable mask of the timer.

17.6.1.2.11. interruptFlagAddress

DIAG_UTILITY_TIMER_DATA_LOOKUP::interruptFlagAddress

Interrupt flag address of the timer.

17.6.1.2.12. interruptFlagMask

DIAG_UTILITY_TIMER_DATA_LOOKUP::interruptFlagMask

Interrupt flag mask of the timer.

17.7. Source Code Reference

The following table provides a list of APIs and their location in the source code:

MODULE	DIAGNOSTIC API	SOURCE CODE REFERENCE
COMMON	- DIAG_delayMs - DIAG_GetClockFreq - DIAG_delayUs - DIAG_SetClockFreq - DIAG_delayInstrCount	diag_delay.c
COMMON	DIAG_UTIL_ADC_RedundantInputsPractice	diag_utility_adc.c
COMMON	- DIAG_UTIL_DAC_ConfigureDC - DIAG_UTIL_DAC_GenerateDC	diag_utility_dac.c

Source Code Reference (continued)

MODULE	DIAGNOSTIC API	SOURCE CODE REFERENCE
COMMON	• DIAG_UTIL_TIMER_GetPeriodCount • DIAG_UTIL_TIMER_GetPrescaler • DIAG_UTIL_TIMER_GetTMRValue • DIAG_UTIL_TIMER_GetInterruptFlagStatus • DIAG_UTIL_TIMER_GetInterruptEnableStatus • DIAG_UTIL_TIMER_DisableInterrupt • DIAG_UTIL_TIMER_SetPrescaler • DIAG_UTIL_TIMER_EnableInterrupt • DIAG_UTIL_TIMER_SetTMRValue • DIAG_UTIL_TIMER_SetPeriodCount • DIAG_UTIL_TIMER_EnableTimer • DIAG_UTIL_TIMER_DisableTimer	diag_utility_timer.c

18. Microchip Information

18.1. Trademarks

The “Microchip” name and logo, the “M” logo, and other names, logos, and brands are registered and unregistered trademarks of Microchip Technology Incorporated or its affiliates and/or subsidiaries in the United States and/or other countries (“Microchip Trademarks”). Information regarding Microchip Trademarks can be found at <https://www.microchip.com/en-us/about/legal-information/microchip-trademarks>.

18.2. Legal Notice

This publication and the information herein may be used only with Microchip products, including to design, test, and integrate Microchip products with your application. Use of this information in any other manner violates these terms. Information regarding device applications is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. Contact your local Microchip sales office for additional support or, obtain additional support at <https://www.microchip.com/en-us/support/design-help/client-support-services>.

THIS INFORMATION IS PROVIDED BY MICROCHIP “AS IS”. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE, OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL, OR CONSEQUENTIAL LOSS, DAMAGE, COST, OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP’S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION.

Use of Microchip devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

18.3. Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip products:

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner, within operating specifications, and under normal conditions.
- Microchip values and aggressively protects its intellectual property rights. Attempts to breach the code protection features of Microchip products are strictly prohibited and may violate the Digital Millennium Copyright Act.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is “unbreakable”. Code protection is constantly evolving. Microchip is committed to continuously improving the code protection features of our products