PSOC™ 4 MCU IEC 60730 Class B and IEC 61508 SIL safety software library for ModusToolbox™

About this document

Scope and purpose

This application note describes Infineon's PSOC™ 4 MCU IEC 60730 Class B and IEC 61508 Safety Integrity Level (SIL) safety software library for use with ModusToolbox™ and Peripheral Driver Libraries (PDL). It includes the safety library source code and example projects for PSOC™ 4100S/4100S Max/4100S Plus 256KB/4500S device families with self-check routines to ensure reliable and safe operation.

This document also describes the API functions that are available in the library.

Intended audience

This document is intended for design engineers, technicians, and developers of electronic systems – anyone who uses PSOC™ 4 software libraries for ModusToolbox™.

Keypoints

- Describes integration of IEC 60730 Class B and IEC 61508 SIL self-test routines using PSOC™ 4 safety software library
- Explains diagnostic coverage methods including CPU, memory, clock, I/O, analog, and communication interface tests, etc.
- Demonstrates implementation of periodic and startup self-tests for fault detection and runtime system integrity
- Provides API-based modular test functions for flexible integration into safety-critical home appliance and industrial applications

About this product family

Product family

The PSOC™ 4 portfolio features multiple families of Arm® Cortex®-M0 and Cortex®-M0+ microcontrollers, integrating intelligent analog, programmable digital, and wireless connectivity. The analog coprocessor enhances the portfolio's robust analog capabilities. This combination delivers versatile, high-performance solutions for diverse applications.

Table of contents

About this document . . . 1
About this product family . . . 1
Table of contents . . . 2
1 Introduction . . . 4
2 Overview of IEC 60730-1 Annex H . . . 5
3 Overview of IEC 61508-2 Annex A . . . 7
4 IEC 60730 Class B and IEC 61508 requirements . . . 8
5 Safety library usage . . . 9
6 API functions for PSOC™ 4 MCU . . . 10
6.1 CPU registers test . . . 13
6.2 Program counter test . . . 14
6.3 APSR test . . . 15
6.4 PRIMASK test . . . 15
6.5 CONTROL test . . . 16
6.6 Program flow test . . . 16
6.7 Interrupt handling and execution test . . . 17
6.8 Clock test . . . 19
6.9 Flash (invariable memory) test . . . 21
6.9.1 Fletcher checksum method . . . 22
6.9.2 CRC-32 method . . . 22
6.9.3 Programming steps . . . 22
6.10 SRAM (variable memory) test . . . 23
6.10.1 March C test . . . 23
6.10.2 GALPAT test . . . 23
6.10.3 March C algorithm . . . 23
6.10.4 GALPAT algorithm . . . 24
6.10.5 Test time . . . 25
6.10.6 Stack-free SRAM test (assembly implementation) . . . 25
6.11 Stack overflow test . . . 27
6.12 Digital I/O test . . . 28
6.13 Analog tests . . . 29
6.13.1 Generating reference voltages . . . 29
6.13.2 ADC test . . . 30
6.13.3 Comparator test . . . 30
6.13.4 Op-Amp test . . . 31
6.14 Communications UART test . . . 31
6.15 Communications SPI test . . . 32

Table of contents

6.16 Communications I2C test . . . 33
6.17 Startup configuration registers test . . . 34
6.18 Watchdog test . . . 36
6.19 TCPWM test . . . 37
6.19.1 Timer/counter test . . . 37
6.19.2 PWM test . . . 38
6.19.3 PWM gate kill test . . . 39
6.20 CAPSENSE™ test . . . 39
6.21 Communications UART data transfer protocol example . . . 39
6.21.1 Data delivery control . . . 40
6.21.2 PSOC™ implementation . . . 41
6.22 Communications CAN FD/ CAN test . . . 44
6.23 DMA test . . . 45

7 Test files . . . 46

8 Summary . . . 47

A Appendix A: Set checksum in flash (invariable memory) test . . . 48

B Appendix B: IEC 60730-1 certificates of compliance . . . 50

C Appendix C: Supported part numbers . . . 61

D Appendix D: MISRA compliance . . . 62

Related resources . . . 64

References . . . 65

Revision history . . . 66

Trademarks . . . 67

Disclaimer . . . 68

1 Introduction

Today, most automatic electronic controls for home appliances and industrial products use single-chip microcontroller units (MCUs). Manufacturers develop real-time embedded firmware that is executed in the MCU and provides the hidden intelligence to control home appliances and industrial machines. MCU damage due to overheating, static discharge, overvoltage, or other factors can cause the end product to enter an unknown or unsafe state.

The International Electrotechnical Commission (IEC) 60730-1 safety standard discusses the mechanical, electrical, electronic, environmental endurance, EMC, and abnormal operation of home appliances. IEC 61508 details the requirements for electrical/electronic/programmable electronic (E/E/PE) safety-related systems in industrial designs. The test requirements of both specifications are similar and are addressed in this document and the safety software library.

This document focuses on Annex H of IEC 60730-1, “Requirements for electronic controls”, and Annex A of IEC 61508-2, “Techniques and measures for E/E/PE safety-related systems: control of failures during operation”. These sections detail tests and diagnostic methods that promote the safe operation of embedded control hardware and software for home appliances and industrial machines.

2 Overview of IEC 60730-1 Annex H

Annex H of the IEC 60730-1 standard classifies appliance software into the following categories:

- Class A control functions, which are not intended to be relied upon for the safety of the equipment. Examples are humidity controls, lighting controls, timers, and switches
- Class B control functions, which are intended to prevent the unsafe operation of controlled equipment. Examples are thermal cutoffs and door locks for laundry equipment
- Class C control functions, which are intended to prevent special hazards (such as an explosion caused by the controlled equipment). Examples are automatic burner controls and thermal cutouts for closed, unvented water heater systems

Large appliance products, such as washing machines, dishwashers, dryers, refrigerators, freezers, and cookers/ stoves, tend to fall into Class B. An exception is an appliance that may cause an explosion, such as a gas-fired controlled dryer, which falls into Class C.

The Class B safety software library and the example projects presented in this application note implement the self-test and self-diagnostic methods prescribed in the Class B category. These methods use various measures to detect software-related faults and errors and respond to them. According to the IEC 60730-1 standard, a manufacturer of automatic electronic controls must design its Class B software using one of the following structures:

- Single channel with functional test
- Single channel with periodic self-test
- Dual channel without comparison (see Figure 1)

In the single-channel structure with the functional test, the software is designed using a single CPU to execute the functions as required. The functional test is executed after the application starts to ensure that all the critical features are functioning reliably.

In the single-channel structure with the periodic self-test, the software is designed using a single CPU to execute the functions as required. The periodic tests are embedded in the software, and the self-test occurs periodically while the software is in execution mode. The CPU is expected to check the critical functions of the electronic control regularly, without conflicting with the end application's operation.

In the dual-channel structure without a comparison, the software is designed using two CPUs to execute critical functions. Before executing a critical function, both CPUs are required to share that they have completed their tasks. For example, when a laundry door lock is released, one CPU stops the motor spinning the drum and the other CPU checks the drum speed to verify that it has stopped, as shown in Figure 1.

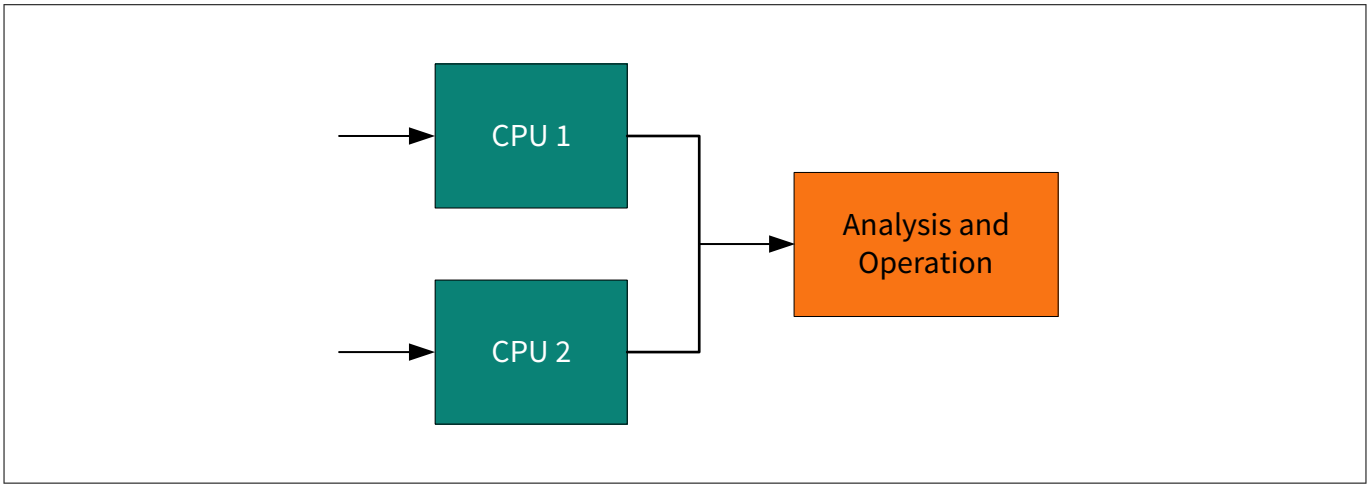


Figure 1 Dual channel without comparison structure

The dual-channel structure implementation is costlier as it requires two CPUs (or MCUs). In addition, it is more complex because two devices are needed to regularly communicate with each other. The single-channel structure with the periodic self-test is the most common implementation.

3 Overview of IEC 61508-2 Annex A

Annex A of IEC 61508-2 defines the maximum diagnostic coverage that may be claimed for relevant techniques and measures in industrial designs. Additional requirements not covered in this library may be applicable to specific industries such as rail, process control, automotive, nuclear, and machinery. For each safety integrity level (SIL), the annex recommends techniques and measures for controlling random hardware, systematic, environmental, and operational failures. More information about architectures and measures is available in Annex B of IEC 61508-6 and Annex A of IEC 61508-7.

To avoid or control such failures when they occur, a number of measures are normally necessary. The requirements in IEC 61508 Annexes A and B are divided into the measures used to avoid failures during the different phases of the E/E/PE system safety lifecycle (Annex B) and those used to control failures during operation (Annex A). The measures to control failures are built-in features of the E/E/PE safety-related systems.

The process starts by evaluating the risk for each hazardous event of the controlled equipment. Typically, diagnostic coverage and safe failure fraction are then determined based on the likelihood of each failure occurring, combined with the consequence of the failure. This weighs the risk in a way that a remote catastrophic failure has a risk similar to a frequent negligible failure, for example. The result of the risk assessment is a target SIL that becomes a requirement for the end system.

The meaning of SIL levels varies based on the frequency of device operation. Most devices are categorized as “high demand” because they are used more than once per year. The probability of a dangerous failure, per hour of use, for the SIL levels at a high level of demand is as follows:

- **SIL 1:** $\geq 10^{-6}$ to $< 10^{-5}$ (1 failure in 11 years)
- **SIL 2:** $\geq 10^{-7}$ to $< 10^{-6}$ (1 failure in 114 years)
- **SIL 3:** $\geq 10^{-8}$ to $< 10^{-7}$ (1 failure in 1,144 years)
- **SIL 4:** $\geq 10^{-9}$ to $< 10^{-8}$ (1 failure in 11,446 years)

4 IEC 60730 Class B and IEC 61508 requirements

According to the IEC 60730-1 Class B Annex H Table H.11.12.7 and the IEC 61508-2 Annex A Tables A.1 to A.14, certain components must be tested, depending on the software classification. Generally, each component offers optional measures to verify or test the corresponding component, providing flexibility for manufacturers.

To comply with Class B IEC 60730 and IEC 61508 for single-channel structures, manufacturers of electronic controls are required to test the components listed in Table 1.

Table 1 Components required to be tested for single-channel structures

Class B IEC 60730 components required to be tested on electronic controls (Table H.11.12.7 in Annex H)	IEC 61508 components required to be tested (Tables A.1–A14 in Annex A)	Fault/error
1.1 CPU registers	A.4, A.10 CPU registers	Stuck at
1.3 CPU program counter	A.4, A.10 Program counter	Stuck at
2. Interrupt handling and execution	A.4 Interrupt handling	No interrupt or too frequent interrupt
3. Clock	A.11 Clock	Wrong frequency
4.1 Invariable memory	A.5 Invariable memory	All single-bit faults
4.2 Variable memory	A.6 Variable memory	DC fault
4.3 Addressing (relevant to variable/ invariable memory)	A.4, A.10 Address calculation	Stuck at
5.1 Internal data path data	A.8 Data paths (internal communication)	Stuck at
5.2 Internal data path addressing (for expanded memory MCU systems only)	–	Wrong address
6.1 External communications data	A.7 I/O units and interface	Hamming distance 3
6.2 External communications addressing	A.7 I/O units and interface	Hamming distance 3
6.3 Timing	–	Wrong point in time/sequence
7.1 I/O periphery	A.7 I/O units and interface	Fault conditions specified in Appendix B, “IEC 60730-1, H.27”
7.2.1 Analog A/D and D/A converters	A.3 Analog signal monitoring	Fault conditions specified in Appendix B, “IEC 60730-1, H.27”
7.2.2 Analog multiplexer	–	Wrong addressing

The user application must determine whether interrupts need to be enabled or disabled during the execution of the Class B Safety Software Library. For example, if an interrupt occurs during execution of the CPU self-test routine, an unexpected change may occur in any register. Therefore, when the interrupt service routine (ISR) is executed, the contents of the register will not match the expected value.

The Class B Safety Software Library example projects show where interrupts need to be disabled and enabled for correct self-testing.

5 Safety library usage

The Safety Software Library described in this application note can be used with PSOC™ 4 MCU devices. The library includes APIs that are designed to maximize application reliability through fault detection. Infineon provides the mtb-stl middleware, which is developed with adherence to Class B standards, and a few code examples in the Infineon GitHub repository.

The STL and Class B certification of the supported devices enable customers to obtain certification for their safety critical-systems. Some self-tests can be applied by only adding an appropriate API function to the `*.c` and `*.h` files from the Class B Safety Software Library.

This application note describes and implements two types of self-test functions:

- Self-test functions to help meet the IEC 60730-1 Class B and IEC 61508-2 standards
  - **CPU registers:** Test for stuck bits
  - **Program counter:** Test for jumps to the correct address
  - **Program flow:** Test for checking correct firmware program flow
  - **Interrupt handling and execution:** Test for proper interrupt calling and periodicity
  - **Clock:** Test for wrong frequency
  - **Flash (invariable memory):** Test for memory corruption
  - **SRAM (variable memory):** Test for stuck bits and proper memory addressing
  - **Stack overflow:** Test for checking stack overflow with the program data memory during program execution
  - **Digital I/O:** Test for pins short
  - **Analog-to-digital converter (ADC):** Test for proper functionality
  - **Comparator:** Test for proper functionality
  - **Communications (UART, SPI, I^2C, CAN/CAN FD):** Test for correct data reception
- Additional self-test functions that PSOC™ 4 MCU can support due to programmable interconnect. Often, the end application also needs these self-tests, even though they are not provided in Appendix B of IEC 60730-1 or IEC 61508-2
  - OpAmp test
  - Startup configuration registers test
  - Watchdog test for chip reset
  - CAPSENSE™ test for proper functionality and external/internal hardware damage

All self-tests can be executed once immediately after device startup and continuously during device operation. Performing the self-test at startup provides an opportunity to determine whether the chip is suitable for operation prior to executing the application firmware. Self-tests executed during normal operation allow continuous damage detection and user-defined corrective actions.

The following sections describe the implementation details for each test and list the APIs required to execute the corresponding tests.

6 API functions for PSOC™ 4 MCU

The following table lists the test functions and return values that integrator must handle.

Table 2 Test functions and return values handled by the integrator

Test file name	Test function	Return values
SelfTest_Analog.c	SelfTest_ADC	0: Test pass 1: Test failed
	SelfTests_Opamp	0: Test pass 1: Test failed
	SelfTests_Comparator	0: Test pass 1: Test failed
SelfTest_Clock.c	SelfTest_Clock	1: Test failed 2: Still testing 3: Test completed 4: Incorrect usage
	SelfTest_Clock_ISR_TIMER	None
SelfTest_ConfigRegisters.c	SelfTests_StartUp_ConfigReg	0: Test pass 1: Test failed
	SelfTests_Save_StartUp_ConfigReg	0: Write in flash is successful 1: Error detected during flash writing
SelfTest_CPU.c	SelfTest_CPU_Registers	0: Test pass 1: Test failed
	SelfTest_PC	0: Test pass 1: Test failed
	SelfTest_PROGRAM_FLOW	0: Test pass 1: Test failed
SelfTest_APSR.c	SelfTest_APSR	0: Test pass 1: Test failed
SelfTest_PRIMASK.c	SelfTest_PRIMASK	0: Test pass 1: Test failed 4: Called from unprivileged Thread mode
SelfTest_CONTROL.c	SelfTest_CONTROL	0: Test pass 1: Test failed 4: Called from Handler mode or called from unprivileged Thread mode

(table continues...)

Table 2 (continued) Test functions and return values handled by the integrator

Test file name	Test function	Return values
SelfTest_Flash.c	SelfTest_FlashCheckSum	1: Test failed 2: Test in progress 3: Test completed
	SelfTest_Flash_init	None
SelfTest_Interrupt.c	SelfTest_Interrupt	0: Test pass 1: Test failed
SelfTest_IO.c	SelfTest_IO	0: Test pass 1: Test failed (Shorts to VCC) 2: Test failed (Shorts to GND)
	SelfTest_IO_GetPinError	The pin number that causes an error
	SelfTest_IO_GetPortError	PORT number that cause an error
	SelfTest_IO_SetPinMask	None
SelfTest_RAM.c	SelfTest_SRAM	0: Test pass 1: Test failed
	SelfTest_SRAM_Stack	0: Test pass 1: Test failed
	SelfTest_SRAM_March_Full	0: Test pass 1: Test failed
	SelfTest_SRAM_March_Runtime	0: Test pass 1: Test failed
SelfTest_Stack.c	SelfTests_Stack_Check_Range	0: Test pass 1: Test failed
	SelfTests_Init_Stack_Range	None
SelfTest_WDT.c	SelfTest_WDT	0: Test pass 1: Test failed
SelfTest_DMAC.c	SelfTest_DMAC	0: Test pass 1: Test failed
SelfTest_Timer_Counter.c	SelfTest_Counter_Timer	0: Test pass 1: Test failed
	SelfTest_Timer_Counter_init	None
SelfTest_PWM.c	SelfTest_PWM	0: Test pass 1: Test failed
	SelfTest_PWM_init	0: Initialization successful 255: Initialization failed

(table continues...)

Table 2 (continued) Test functions and return values handled by the integrator

Test file name	Test function	Return values
SelfTest_PWM_GateKill.c	SelfTest_PWM_GateKill	0: Test pass 1: Test failed
SelfTest_CANFD.c	SelfTest_CANFD	0: Test pass 1: Test failed
SelfTest_CAN.c	SelfTest_CAN	0: Test pass 1: Test failed
SelfTest_UART_SCB.c	SelfTest_UART_SCB	1: Test failed 2: Still testing 3: Test completed OK 4: Error, Tx not empty 5: Error, Rx not empty 6: Error, UART is not enabled
SelfTest_UART_master_message.c	UartMesMaster_Init	None
	UartMesMaster_DataProc	0: If the unit started a message process 1: If not (because the unit is busy or data input is invalid)
	UartMesMaster_State	0: The last message process ended with an error 1: The last message process succeeded 2: The unit is busy with the message process
	UartMesMaster_GetDataSize	Received data size in buffer
SelfTest_UART_slave_message.c	UartMesSlave_Init	None
	UartMesSlave_Respond	0: If the unit starts a response process 1: If not (because of IDLE or RESPOND state)
	UartMesSlave_State	0: The unit to wait for the packet from master 1: The unit to receive the packet with the marker and a respond must be sent 2: The unit is busy with sending a response
	UartMesSlave_GetDataSize	The size of the received data in the buffer

(table continues...)

Table 2 (continued) Test functions and return values handled by the integrator

Test file name	Test function	Return values
	`UartMesSlave_GetDataPtr`	The pointer to the received data buffer
`SelfTest_SPI_SCB.c`	`SelfTest_SPI_SCB`	1: Test failed 2: Test in progress 3: Test completed 4: TX not empty 5: RX not empty
`SelfTest_I2C_SCB.c`	`SelfTest_I2C_SCB`	1: Test failed 2: Still testing 3: Test completed OK

6.1 CPU registers test

PSOC™ 4 with the Arm® Cortex®-M0 or Cortex®-M0+ CPU has 16-bit and 32-bit registers:

- **R0 to R12:** General-purpose registers
- **R13 – Stack pointer (SP):** There are two stack pointers, with only one available at a time. The SP is always 32-bit-word aligned; bits [1:0] are always ignored and considered to be '0'
- **R14 – Link register:** This register stores the return program counter during function calls
- **R15 – Program counter:** This register can be written to control the program flow

The CPU registers test detects stuck-at faults in the CPU registers by using the checkerboard test. This test ensures that the bits in the registers are not stuck at value '0' or '1'. It is a nondestructive test that performs the following major tasks:

1. The contents of the CPU registers to be tested are saved on the stack before executing the routine
2. The registers are tested by successively writing the binary sequences 01010101 followed by 10101010 into the registers, and then reading the values from these registers for verification
3. The test returns an error code if the returned values do not match

The checkerboard method is implemented for all CPU registers except the program counter.

Function

```
uint8       SelfTest_CPU_Registers(void)
```

Returns:

- 0 – Test pass
- 1 – Test failed

Located in:

- `SelfTest_CPU.c`
- `SelfTest_CPU.h`

The function `SelfTest_CPU_Registers` is called to do the CPU test.

If an error is detected, the PSOC™ device should not continue to function because its behavior can be unpredictable and therefore potentially unsafe.

6.2 Program counter test

The PSOC™ 4 CPU program counter R15 register is part of the CPU register set. To test these registers, a checkerboard test is commonly used; the addresses 0x555… and 0xAAA… must be allocated for this test. 0x555… and 0xAAA… represent the checkerboard bit patterns. The address used will depend on the device's flash size and should be the largest address in the flash following the checkerboard pattern while byte-aligned to 4.

The program counter (PC) test implements the functional test defined in Section H.2.16.5 of the IEC 60730 standard. The PC holds the address of the next instruction to be executed. The test performs the following major tasks:

1. The functions that are in the flash memory at different addresses are called. For PSOC™ 4, this can be done using the linker script in the `*.ld` file for GCC, `*.sct` for Arm®, and `*.icf` for IAR. For the GCC compiler, these are located in the custom linker file provided, as follows:

```
NV_CONFIG2 0x15554 :
{
        . = 0x00;
        KEEP (*(PC5555))
} >flash =0

NV_CONFIG3 0x2AAA8 :
{
        . = 0x00;
        KEEP (*(PCAAAA))
} >flash =0
```

In the example project, it is already added to the custom linker .ld file shipped with the example project.

2. The functions return a unique value
3. The returned value is verified using the PC test function
4. If the values match, the PC branches to the correct location, or a WDT triggers a reset because the program execution is out of range

Function

```
uint8       SelfTest_PC(void)
```

Returns:

- 0 – Test pass
- 1 – Test failed

Located in:

- `SelfTest_CPU.c`
- `SelfTest_CPU.h`

The function `SelfTest_PC()` is called to perform the PC test.

***Note**: Linker file for each device family is created based on the default device for the corresponding family. Update the linker file if the flash size is different from the default device.*

6.3 APSR test

The APSR test can:

1. Save the original APSR value or preserve all application-visible state affected by the test
2. Write a known pattern to the writable APSR fields
3. Read APSR back
4. Mask the result to the implemented APSR bits
5. Compare the result with the expected pattern
6. Repeat using the complementary pattern
7. Restore the original APSR state
8. Return `OK_STATUS` only when all comparisons pass

Function

```
uint8 SelfTest_APSR(void)
```

Returns:

`OK_STATUS (0U)` - No error/Test pass

`ERROR_STATUS (1U)` - Error detected/Test failed

Located in:

`SelfTest_APSR.c`

`SelfTest_APSR.h`

The function `SelfTest_APSR()` is called to do the APSR test.

6.4 PRIMASK test

PRIMASK contains the interrupt-mask bit `PM` in bit 0. The test can verify both legal values:

1. Save the current PRIMASK value
2. Write `PM = 0`
3. Read PRIMASK and verify `PM = 0`
4. Write `PM = 1`
5. Read PRIMASK and verify `PM = 1`
6. Restore the original PRIMASK value
7. Return `OK_STATUS` only when all comparisons pass

Function

```
uint8 SelfTest_PRIMASK(void)
```

Returns:

`OK_STATUS (0U)` - No error/Test pass

`ERROR_STATUS (1U)` - Error detected/Test failed

`ERROR_INCORRECT_USAGE_STATUS (4U)` - Called from unprivileged Thread mode

Located in:

`SelfTest_PRIMASK.c`

`SelfTest_PRIMASK.h`

The function `SelfTest_PRIMASK()` is called to do the PRIMASK test.

6.5 CONTROL test

The CONTROL test can verify only the bits selected for testing and supported by the build.

- SPSEL can always be tested
- nPRIV cannot be written or tested. Require privileged Thread-mode entry and preserve its value
- Test FPCA when enabled and SFPA for Secure FPU builds. FP fields may be skipped with SELFTEST_CONTROL_SKIP_FP_BITS where permitted

The test can:

1. Reject Handler-mode and unprivileged calls with ERROR_INCORRECT_USAGE_STATUS
2. Save PRIMASK. Disable interrupts during assembly, then restore it if the test returns
3. Save CONTROL, active SP, PSP, and PSPLIM where supported
4. Use a static test stack for SPSEL. Set PSP and PSPLIM where supported
5. Test cleared and set patterns. Preserve bits outside the test mask
6. Read CONTROL after each write. Mask and compare the selected bits
7. Check the active SP for each SPSEL state. Probe the test stack
8. Restore CONTROL, PSP, and PSPLIM where supported. Verify the saved state
9. Return OK_STATUS on success. Return ERROR_STATUS for a mismatch if restoration succeeds
10. If restoration fails, disable interrupts and loop forever. Recovery requires an external watchdog; no status is returned

Function

```
uint8 SelfTest_CONTROL(void)
```

Returns:

OK_STATUS (0U) - No error/Test pass

ERROR_STATUS (1U) - Error detected/Test failed

ERROR_INCORRECT_USAGE_STATUS (4U) - Called from Handler mode or called from unprivileged Thread mode

Located in:

SelfTest_CONTROL.c

SelfTest_CONTROL.h

The function SelfTest_CONTROL() is called to do the CONTROL test.

6.6 Program flow test

This demonstrates a specific method that can be used to check program execution flow. For every critical execution code block, unique numbers are added to or subtracted from complementary counters before block execution and immediately after execution. These procedures allow you to see if the code block is correctly called from the main program flow and to check if the block is correctly executed.

As long as there are always the same number of exit and entry points, the counter pair will always be complementary after each tested block. See Figure 2.

Any unexpected values should be treated as a program flow execution error.

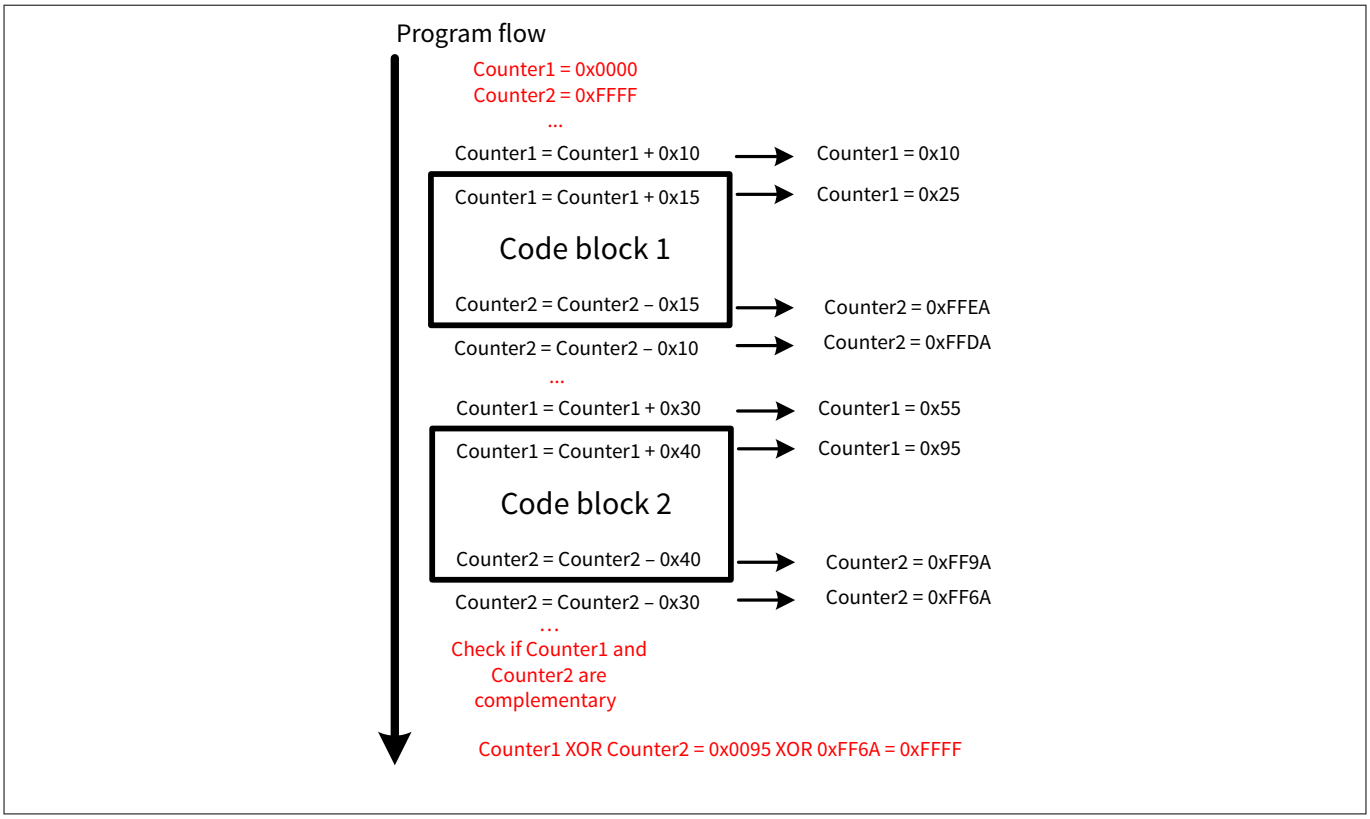


Figure 2 Program flow test

Function

```
uint8       SelfTest_PROGRAM_FLOW(void)
```

Returns:

- 0 – Test pass
- 1 – Test failed

Located in:

- `SelfTest_CPU.c`
- `SelfTest_CPU.h`

The `SelfTest_ PROGRAM_FLOW()` function is called to perform the program flow test.

6.7 Interrupt handling and execution test

The PSOC™ 4 interrupt controllers provide the mechanism for hardware resources to change the program address to a new location independent of the current execution in main code. They also handle the continuation of the interrupted code after completion of the ISR.

The interrupt test implements the independent time-slot monitoring defined in section H.2.18.10.4 of the IEC 60730 standard. It checks whether the number of interrupts that occurred is within the predefined range.

The goal of the interrupt test is to verify that interrupts occur regularly. The test checks the interrupt controller by using the interrupt source driven by the timer UM.

Function

```
uint8       SelfTest_Interrupt(TCPWM_Type *base, uint32_t cntNum)
```

Returns:

- 0 – Test pass
- 1 – Test failed

Parameters:

- `base` – Pointer to a TCPWM instance
- `cntNum` – Counter instance number in the selected TCPWM

Located in:

- `SelfTest_Interrupt.c`
- `SelfTest_Interrupt.h`

***Note**: Global interrupts must be enabled for this test, but all interrupts except isr_1 must be disabled.*

The `SelfTest_Interrupt()` function is called to check the interrupt controller operation. The self-test is parameterized to allow the use of any TCPWM instance. Calling the function starts the timers.

The timer is configured to generate 13 interrupts every ms. `SelfTest_Interrupt_ISR_TIMER` counts the number of interrupts that occurred. If the counted value in `SelfTest_Interrupt_ISR_TIMER` is ≥ 9 and ≤ 15, then the test is passed. The specific number of interrupts to pass this test is application dependent and can be modified as required.

Figure 3 shows the interrupt self-test flowchart.

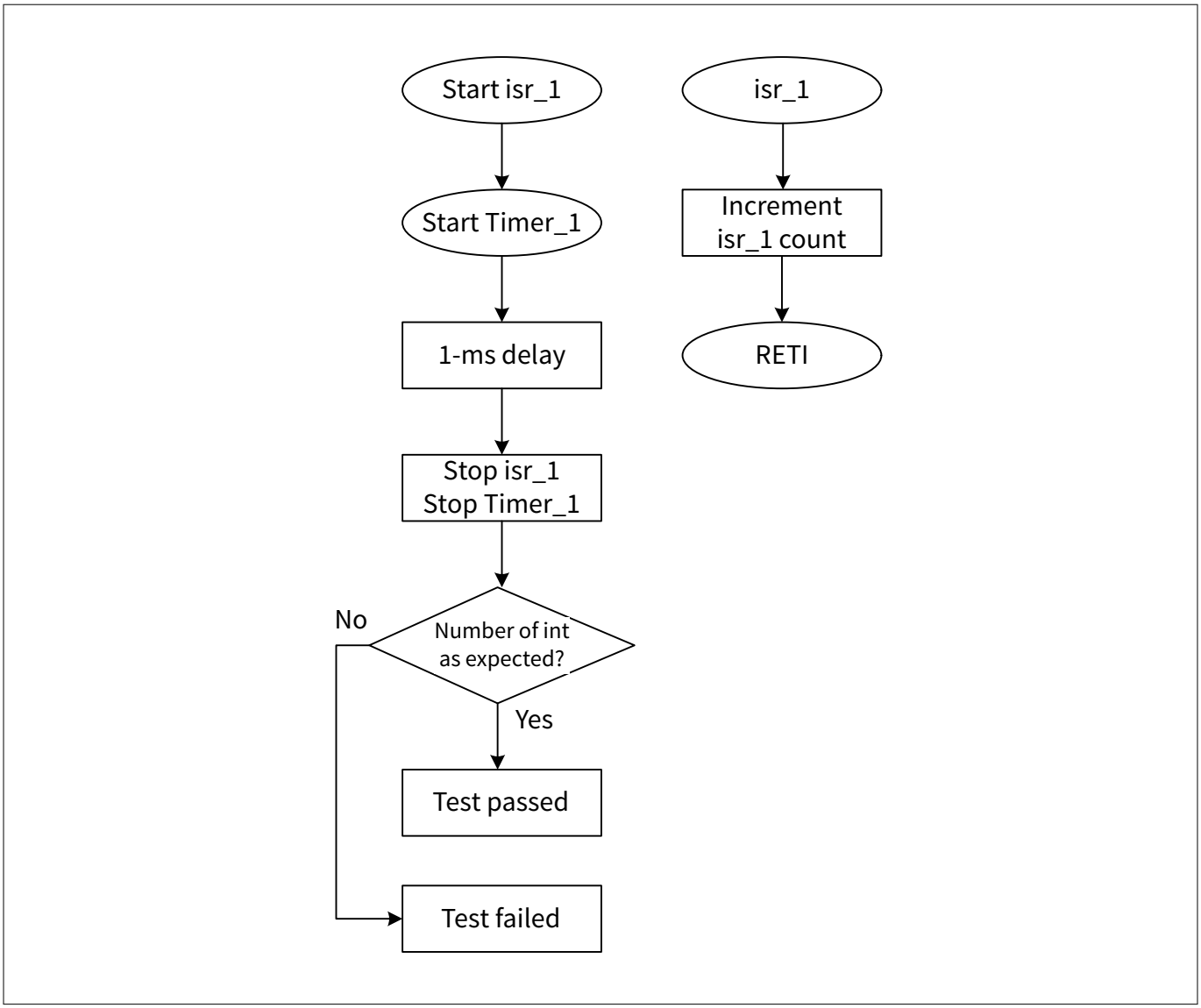


Figure 3 Interrupt self-test flowchart

***Note**: "SelfTest_Interrupt_ISR_TIMER" is renamed as "isr_1" in the diagram above.*

6.8 Clock test

The clock test implements independent time-slot monitoring defined in section H.2.18.10.4 of the IEC 60730 standard. It verifies the reliability of the internal main oscillator (IMO) system clock, specifically, that the system clock is neither too fast nor too slow within the tolerance of the internal low-speed oscillator (ILO). The ILO clock is accurate up to ± 50 percent.

Function

```
uint8       SelfTest_Clock(TCPWM_Type * base, uint32_t cntNum)
```

Returns:

- 1 – Test failed
- 2 – Still testing

- 3 – Test completed
- 4 – Incorrect usage

Parameters:

- `base` – Pointer to a TCPWM instance
- `cntNum` – Counter instance number in the selected TCPWM

Located in:

- `SelfTest_Clock.h`
- `SelfTest_Clock.c`

***Note**:* *The tested clock accuracies are defined in the `SelfTest_Clock.h` file. Clock accuracies may be modified based on end system requirements.*

The clock test uses the 16-bit timer0 integrated into the WDT and clocked by the 40 kHz ILO. The WDT timer0 is a continuous up counting 16-bit timer with overflow. The test starts by reading the current count of the timer, then waits 1 ms using a TCPWM Timer delay clocked by the IMO, and finally reads the timer count a second time. The two count values are then subtracted and optionally corrected for the special case of a timer overflow mid test. The measured period (nominally 33 counts) is then tested. If it is within the predefined range, the test is passed. Figure 4 shows the clock self-test flowchart.

The self-test is designed to be called several times. The initial call to `Selftest_Clock()` will configure the TCPWM timer and retrieve the initial WDT count. Subsequent calls will check if the ISR for the TCPWM timer has triggered. When it is detected that the ISR has been called once, it will compare the WDT value from the initial self-test function call and the WDT value from the ISR. This method of testing allows the self-test to run in the background if needed.

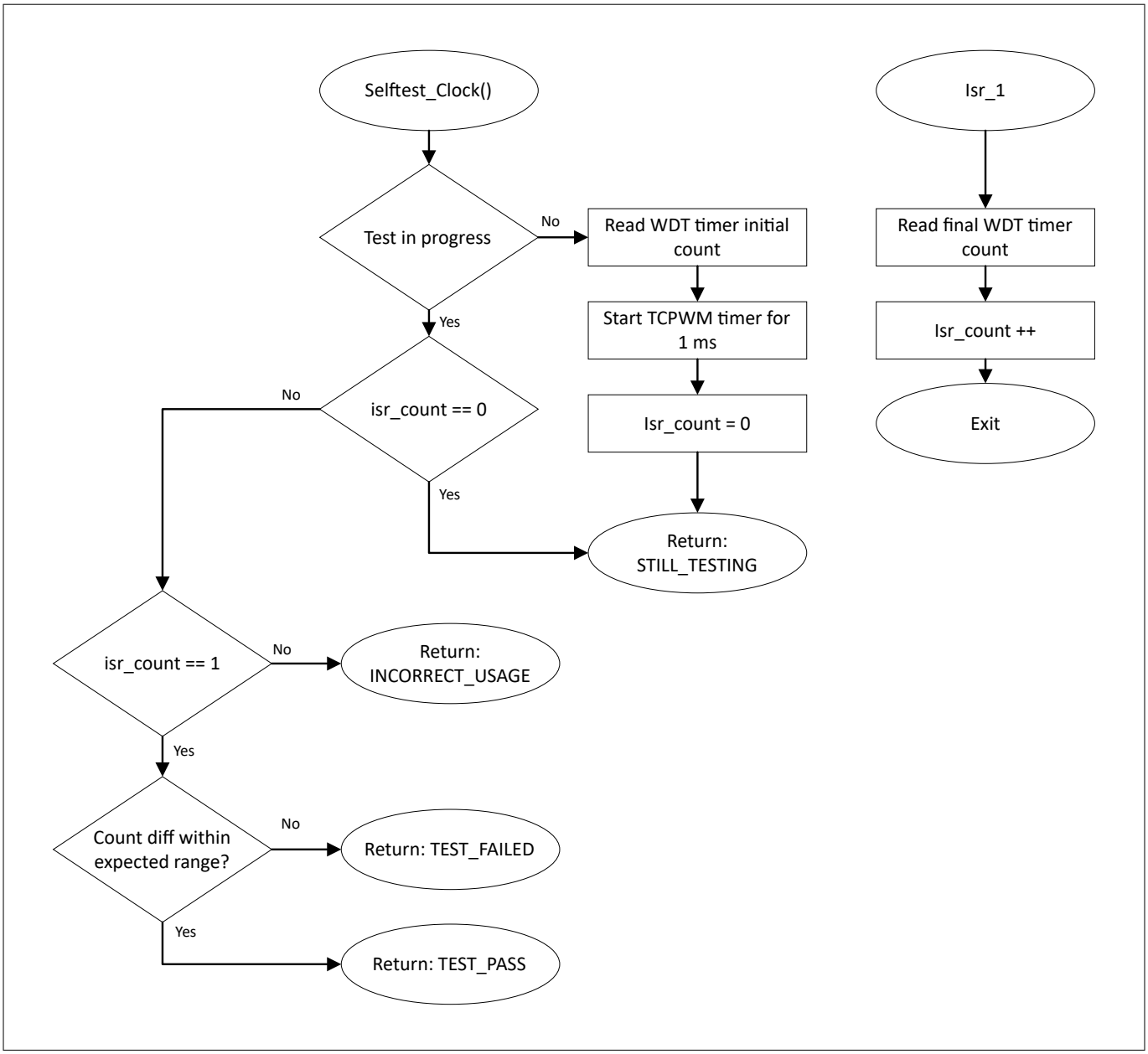


Figure 4 Clock self-test flowchart

6.9 Flash (invariable memory) test

PSOC™ 4 devices include an on-chip flash memory of up to 384 KB. The flash memory is organized in rows, where each row contains 128 data bytes.

To complete a full diagnostic of the flash memory, a checksum of all used flash needs to be calculated.

The current library uses a Fletcher's 64-bit checksum or 32-bit CRC32. The Fletcher's 64-bit method was chosen as the default because it maintains sufficient bit error detection and is significantly faster and smaller than CRC-32. Fletcher's 64-bit algorithm reads a full 32-bit word and usually only performs 2 additions per read while requiring no ROM lookup table as compared to CRC32.

You can change the checksum modifying the FLASH_TEST_MODE macro in `SelfTest_Flash.h`.

FLASH_END_ADDR in the `SelfTest_Flash.h` file defines the end of flash address that needs to be monitored.

6.9.1 Fletcher checksum method

The checksum flash test reads each ROM or flash location and accumulates the values in a 64-bit variable to calculate a running checksum of the entire flash memory. The actual 64-bit checksum of flash is stored in the last 8 bytes of flash itself. The provided custom linker file includes a “flash_checksum” section for the GCC and Arm® compilers. The address is defined in line for the IAR compiler. When the test reaches the end of flash minus 8 bytes (0x05FFF8 on 384 KB devices), it stops. Custom linker files are used to place the checksum in the desired location (see the note below). The calculated checksum value is then compared with the actual value stored in the last 8 bytes of flash. A mismatch indicates flash failure, and code execution is frozen to avoid trying to execute invalid code.

***Note**: The checksum can also be stored in SFLASH, EEPROM, or any other external flash.*

6.9.2 CRC-32 method

The CRC-32 method uses a lookup table to speed execution but is slower than the Fletchers 64-bit method due to the need to read memory at byte boundaries and perform multiple bit operations per read. The CRC32 polynomial chosen is the most common version defined in ISO/IEC/IEEE 802-3 with polynomial 0x04C11DB7 (Reversed = 0xEDB88320). To maintain compatibility with the Fletcher 64-bit storage size the CRC32 value is stored in the last 4 bytes of the 8 byte reserved Flash space.

6.9.3 Programming steps

Before starting the test, you need to set the correct precalculated checksum, as described in Appendix A.

Function

```
uint8 SelfTest_FlashCheckSum(uint32_t DoubleWordsToTest)
```

Returns:

- 1 – Error detected
- 2 – Checksum for one block calculated, but end of the flash was not reached
- 3 – Pass; checksum of all flash is calculated and checked

Parameters:

- `DoubleWordsToTest` – Number of 32-bit double words of flash to be calculated per each function call

Located in:

- SelfTest_Flash.c
- SelfTest_Flash.h

The `SelfTest_FlashCheckSum()` function is called to perform the flash memory corruption test using the checksum method. During the call, this function calculates the checksum for one block of flash. The size of the block can be set using parameters in the `SelfTest_Flash.h` file:

```
/*Set size of one block in Flash test*/
#define FLASH_DOUBLE_WORDS_TO_TEST (512u)
```

The function must be called multiple times until the entire flash area is tested. Each call to the function is auto-incremented to the next test block. If the checksum for the block is calculated and the end address of the tested flash is reached, the test returns 0x03. If the checksum for the block is calculated but the end address of flash is not reached, the test returns 0x02. If an error is detected, the test returns 0x01.

***Note**: The check does not work if there is a change in flash during runtime. The checksum needs to be updated before calling the test. Other tests that may change the contents of flash must be called prior to the flash test.*

6.10 SRAM (variable memory) test

***Note**: PSOC™ 4 devices include an on-chip SRAM of up to 32 KB. Part of this SRAM includes the stack located at the end of memory.*

The variable memory test implements the periodic static memory test defined in section H.2.19.6 of the IEC 60730 standard. It detects single-bit faults in the variable memory. Variable memory tests can be destructive or non-destructive. Destructive tests destroy the contents of memory during testing, whereas nondestructive tests preserve the memory contents. While the test algorithm used in this library is destructive, it is encapsulated in code that first saves the memory contents before a test and then restores the contents after completion.

The variable memory contains data, which varies during program execution. The RAM memory test is used to determine if any bit of the RAM memory is stuck at '1' or '0'. The March memory test and checkerboard test are among the most widely used static memory algorithms to check for DC faults.

The March tests comprise a family of similar tests with slightly different algorithms. The March test variations are denoted by a capital letter and allow tailoring of the test to a specific architecture's test requirements. The March C test is implemented for the PSOC™ 4 Safety Software Library because it provides better test coverage than the checkerboard algorithm and is the optimal March method for this device. Separate functions are implemented for the "variable SRAM" and "stack SRAM" areas to ensure no data is corrupted during testing.

6.10.1 March C test

March tests perform a finite set of operations on every memory cell in the memory array. The March C test is used to detect the following types of faults in the variable memory:

- Stuck-at fault
- Addressing fault
- Transition fault
- Coupling fault

The test complexity is 11n, where "n" indicates the number of bits in memory, because 11 operations are required to test each location. While this test is normally destructive, Infineon provides the March C test without data corruption by testing only a small block of memory in each test, allowing the block's contents to be saved and restored.

6.10.2 GALPAT test

GALPAT test is similar to March tests, which perform a finite set of operations on every memory cell in the memory array. The GALPAT tests differ only in the algorithm being used to perform the test.

6.10.3 March C algorithm

The March C-memory built-In self-test (MBIST) algorithm covers the majority of SRAM faults, including addressing faults, stuck-at fault, state transition fault, and various coupling faults.

The algorithm:

1. Duplicates the data in the area to be tested as this test is destructive
2. Writes '0' to the block in ascending order
3. Reads '0' and writes '1' in ascending order

4. Reads '1' and writes '0' in ascending order
5. Reads '0'
6. Reads '0' and writes '1' in descending order
7. Reads '1' and writes '0' in descending order
8. Reads '0'
9. Copies back the original data function

Function

```
uint8_t SelfTest_SRAM(stl_sram_test_mode_t type, uint8_t* startAddr, uint32_t size, uint8_t*
buffAddr, uint32_t buffSize)
```

Returns:

- 0 – Test pass
- 1 – Test failed

Parameters:

- `type` – The type of RAM test to be run; MARCH or GALPAT
- `startAddr` - The pointer to start of the RAM block to be tested
- `Size` - The size of RAM block to be tested
- `buffAddr` - The pointer to start of the buffer to be used to store/restore data of RAM block to be tested. If NULL, test would be performed without store/restore being performed
- `buffSize` - The `size` of buffer used to store/restore. If `buffSize` is smaller than `size`, the test internally loops through the RAM block operating on memory size equal to `buffSize` in each iteration

Located in:

- `SelfTest_RAM.c`
- `SelfTest_RAM.h`

Function

```
uint8_t SelfTest_SRAM_Stack(uint8_t* stackBase, uint32_t stackSize, uint8_t* altStackBase)
```

Returns:

- 0 – Test pass
- 1 – Test failed

Parameters:

- `stackBase` – The pointer to the stack base
- `stackSize` - The size stack
- `altStackBase` - The pointer to start of the RAM area to be used as alternate stack base

Located in:

- `SelfTest_RAM.c`
- `SelfTest_RAM.h`

6.10.4 GALPAT algorithm

The algorithm:

1. Duplicates data in the area to be tested as this test is destructive
2. Initializes the chosen range of memory uniformly (i.e., all 0s or all 1s)

3. Tests the first memory cell and then inverts; all remaining cells are inspected to ensure their contents are correct
4. Checks after every read access to one of the remaining cells and the inverted cell
5. Repeats this procedure for each cell in the chosen memory range
6. Carries a second run with the opposite initialization
7. Copies back the original data function

Any difference produces a failure message. It can detect stuck-at faults and direct coupling faults. At the end of this test, the original data is restored to its correct location in memory. You can run the GALPAT test by executing the same `SelfTest_SRAM()` API with the first parameter type set to '1'.

6.10.5 Test time

The default SRAM test configuration tests 256 bytes each time it is called until it tests the specified SRAM size address space. The block size and frequency the test is called is modifiable. The test takes 1.8 ms per 1 MB in its default configuration.

The default stack memory test configuration tests 21 bytes each time it is called until it tests the 1 KB stack address space. The block size and frequency the test is called is modifiable. The test takes 1.6 ms for each 1 KB in its default configuration.

6.10.6 Stack-free SRAM test (assembly implementation)

Assembly-based March tests with stack-free execution, safe to test both variable RAM and stack memory. Uses March C- (10n, full coverage) or March X (6n, fast runtime) algorithms.

Test modes

```
typedef enum {     STL_SRAM_MARCH_C_MINUS
                = 0, /**< March C- (10n complexity, full coverage) */
STL_SRAM_MARCH_X       = 1 /**< March X (6n complexity, fast runtime) */ }
stl_sram_march_mode_t;
```

Function

uint8_t SelfTest_SRAM_March_Full(uint32_t startAddress, uint32_t endAddress, uint32_t blockSize, uint32_t backupAddress, stl_sram_march_mode_t mode)

Performs complete SRAM March test on entire memory range in single call. Backup buffer is tested first, then each block is backed up, tested, and restored.

Parameters:

startAddress - Start of test region (4-byte aligned)

endAddress - End of test region (4-byte aligned)

blockSize - Test block size (4-byte aligned)

backupAddress - Backup buffer (outside test region, ≥ blockSize)

mode - STL_SRAM_MARCH_C_MINUS or STL_SRAM_MARCH_X from stl_sram_march_mode_t enum

Returns: 0 = passed, 1 = failed

Function

uint8_t SelfTest_SRAM_March_Runtime(uint32_t startAddress, uint32_t endAddress, uint32_t* currentAddress, uint32_t blockSize, uint32_t backupAddress, stl_sram_march_mode_t mode)

Incremental SRAM test - one block per call. Initialize *currentAddress to startAddress before first call. Function advances state automatically and wraps after last block.

Parameters:
startAddress - Start of test region (4-byte aligned)
endAddress - End of test region (4-byte aligned)
currentAddress - State pointer (initialize to startAddress)
blockSize - Test block size (4-byte aligned)
backupAddress - Backup buffer (outside test region, ≥ blockSize)
mode - STL_SRAM_MARCH_C_MINUS or STL_SRAM_MARCH_X from stl_sram_march_mode_t enum
Returns: 0 = passed, 1 = failed

Usage example

```
/* Memory configuration */ #define
                RAM_START       0x20000000 #define RAM_END
                0x20007F00  /* 32 KB - BLOCK_SIZE */ #define
                BLOCK_SIZE      256 /* 256 bytes per block */ #define
                BACKUP_BUFFER   0x20007F00  /* Must be outside test region */ /* Full test at
startup */ result
                = SelfTest_SRAM_March_Full(RAM_START, RAM_END, BLOCK_SIZE, BACKUP_BUFFER,
STL_SRAM_MARCH_C_MINUS); /* Fast periodic runtime test (call repeatedly during application code
execution)
                */ static uint32_t pos = 0x20000000; result =
SelfTest_SRAM_March_Runtime(RAM_START, RAM_END, &pos, BLOCK_SIZE, BACKUP_BUFFER,
STL_SRAM_MARCH_X);
```

Note: Disable interrupts during test execution.

Stack usage

Both functions use the stack only at entry and exit to save/restore CPU context (9 registers: R4-R11 + LR = 36 bytes). During the actual March test execution phase, **no stack operations occur** - all data is held in CPU registers (R4-R12). The internal `March_Segment_Internal` function is fully stack-free.

- SelfTest_SRAM_March_Full: PUSH at entry (36 bytes), POP at exit. No stack during test loop
- SelfTest_SRAM_March_Runtime: PUSH at entry (36 bytes), POP at exit. No stack during block test

To test stack memory: place the backup buffer in a non-stack RAM region, then include the stack address range in startAddress/endAddress. The stack pointer remains valid throughout because the function's own stack frame is established before the test begins.

Memory usage

Item	Size	Notes
Code Size	520 bytes	Both functions with internal routines
Backup Buffer	>= block size	User-allocated, outside test region
Stack Usage	36 bytes	Only at entry/exit (9 registers saved)
Runtime test state	4 bytes	Runtime test only (currentAddress variable)

Performance (8 KB at 48 MHz)

Test	Execution time
Full March C-	3.43 ms
Full March X	2.44 ms
Runtime March C- (256 bytes/call)	0.084 ms
Runtime March X (256 bytes/call)	0.054 ms

Test sources are located in stl/TOOLCHAIN_xxx/SelfTest_RAM_MARCH_xxx.S files, where xxx is GCC_ARM, ARM and IAR.

6.11 Stack overflow test

The stack is a section of the RAM used by the CPU to store information temporarily. This information can be data or an address. The CPU needs this storage area since there are only a limited number of registers.

In PSOC™ 4, the stack is located at the end of RAM and grows downward. The stack pointer is 32 bits wide and is decremented with every PUSH instruction and incremented with POP.

The purpose of the stack overflow test is to ensure that the stack does not overlap with the program data memory during program execution. This can occur, for example, if recursive functions are used.

To perform this test, a reserved fixed-memory block at the end of the stack is filled with a predefined pattern, and the test function is periodically called to verify it. If stack overflow occurs, the reserved block will be overwritten with corrupted data bytes, which should be treated as an overflow error.

Function

```
uint8_t SelfTests_Stack_Check_Range(uint16_t* stack_address, uint16_t stack_length)
```

Returns:

- 0 – Test pass
- 1 – Test failed

Parameters:

- `stack_address` – The pointer to the stack
- `stack_length` - The length of the stack

Function

```
void SelfTests_Init_Stack_Range(uint16_t* stack_address, uint16_t stack_length, uint8_t
stack_pattern_blk_size)
```

Returns:

- None

Parameters:

- `stack_address` – The pointer to the stack
- `stack_length` – The length of the stack
- `stack_pattern_blk_size` - No of bytes to be used to fill the pattern. Must be 2^n where n=1 to n=8

Located in:

- `SelfTest_Stack.c`
- `SelfTest_Stack.h`

This function is called periodically at run time to test for stack overflow.

The pattern can be modified using the macro located in `SelfTest_Stack.h`:

```
#define STACK_TEST_PATTERN 0x55AAu
```

6.12 Digital I/O test

PSOC™ 4 provides up to 98 programmable GPIO pins. Any GPIO pin can be CAPSENSE™, LCD, analog, or digital. Drive modes, strengths, and slew rates are programmable.

Digital I/Os are arranged into ports, with up to eight pins per port. Some of the I/O pins are multiplexed with special functions (USB, debug port, crystal oscillator). Special functions are enabled using control registers associated with the specific functions.

The test goal is to ensure that I/O pins are not shorted to GND or Vcc.

In normal operating conditions, the pin-to-ground and pin-to-VCC resistances are very high. To detect any shorts, resistance values are compared with the PSOC™ 4 internal pull-up resistors.

To detect a pin-to-ground short, the pin is configured in the resistive pull-up drive mode. Under normal conditions, the CPU reads a logical one because of the pull-up resistor. If the pin is connected to ground through a small resistance, the input level is recognized as a logical zero.

To detect a sensor-to-VCC short, the sensor pin is configured in the resistive pull-down drive mode. The input level is zero under normal conditions.

***Note**: This test is application-dependent and may require customization. The default test values may cause the pins to be momentarily configured into an incorrect state for the end application.*

Function

```
uint8_t SelfTest_IO(void)
```

Returns:

- 0 – No error
- 1 – Error detected (short to VCC)
- 2 – Error detected (short to GND)

Function

```
uint8_t SelfTest_IO_GetPortError(void)
```

Returns:

- PORT number that cause an error

Function

```
uint8_t SelfTest_IO_GetPinError(void)
```

Returns:

- PIN number that cause an error

Function

```
void SelfTest_IO_SetPinMask(const uint8_t* pinMaskArr)
```

Returns:

- None

Parameters:

- `pinMaskArr` – The custom pin mask array

Located in:

- `SelfTest_IO.c`
- `SelfTest_IO.h`

The function `SelfTests_IO()` is called to check shorts of the I/O pins to GND or Vcc. The `PinToTest` array in the `SelfTests_IO()` function is used to set the pins that must be tested.

For example:

```
          static const uint8 PinToTest[] =
{
        0xCCu,        /* PORT0 mask */
        0x1Fu,        /* PORT1 mask */
        0xC2u,        /* PORT2 mask */
        0xF0u,        /* PORT3 mask */
        0x0Cu,        /* PORT4 mask */
};
```

Each pin is represented by the corresponding bit in the `PinToTest` table port mask. Pin 0 is represented by the LSB, and pin 7 by the MSB. If a pin should be tested, a corresponding bit should be set to '1'.

6.13 Analog tests

Analog tests comprise the ADC, comparator, and Op-Amp test. These tests rely on two fixed reference voltages. Depending on the specific PSOC™ device, these voltage references can be generated internally or externally with the help of external hardware. It is up to the user to configure the voltage reference and route the voltages to the hardware under test. This provides flexibility for the user and control over internal routing.

The AMUX buses are segmented allowing for greater flexibility in normal use. In the mtb-example-psoc4-analog-test code example, these segments are joined, thereby allowing the voltage references to route to the ADC and other analog resources under test.

6.13.1 Generating reference voltages

You can provide the voltage reference to the device using external hardware or using an internal reference voltage. A possible method is to use a voltage divider between VDDA and GND to generate either 2/3*VDDA or 1/3*VDDA as the reference voltage. These reference voltages can be connected to the pin configured as the analog input pin.

6.13.2 ADC test

The ADC test implements an independent input comparison as defined in Section H.2.18.8 of the IEC 60730 standard. It provides a fault or error control technique using which the inputs or outputs designed to be within specified tolerances are compared.

This test aims to check the SAR ADC analog functions. Each ADC enabled by the test is connected to an internal or external reference voltage using the programmable analog block. If the measured value is more than the expected value, an error is returned.

Function

```
uint8_t SelfTests_ADC(SAR_Type * base, uint32_t channel, int16_t expected_res, int16_t
accuracy, uint32_t vbg_channel, bool count_to_mV)
```

Returns:

- 0 – Test pass
- 1 – Test failed

Parameters:

- `base` – Pointer to structure describing SAR instance registers
- `channel` – Number of the channel reading the voltage
- `expected_res` – Expected result in mV
- `accuracy` – Threshold above and below the expected value that is acceptable
- `vbg_channel` – only used in CAT1C device; used to convert the count to mV
- `count_to_mV` – convert the count to mV

Located in:

- `SelfTest_Analog.h`
- `SelfTest_Analog.c`

To perform this test, the ADC is configured to scan the passed channel. A predefined (reference) voltage is generated externally and is sampled by the ADC.

The test is a success if the digitalized input voltage value is equal to the required reference voltage value within the defined accuracy. When the test is a success, the function returns ‘0’; otherwise, it returns ‘1’.

6.13.3 Comparator test

If the inputs to the comparators can be connected to the internal reference voltage, one side of each comparator is connected to the internal reference voltage and the other side is connected to ground. The results are checked. The inputs are switched and the result is checked again. If the inputs cannot be modified via the programmable analog block, the test uses the pull-up and pull-down resistors in the GPIO to conduct a similar test.

Function

```
uint8_t SelfTest_Comparator(LPCOMP_Type const* lpcomp_base,
              cy_en_lpcomp_channel_t lpcomp_channel,uint8_t expected_res)
```

Returns:

- 0 – Test pass
- 1 – Test failed

Parameters:

- `lpcomp_base` – Low-power comparator registers structure-pointer
- `lpcomp_channel` – Low-power comparator channel number
- `expected_res` – Expected output of the comparator. Non-zero value when it is expected that the positive input is greater than the negative input voltage

Located in:

- `SelfTest_Analog.h`
- `SelfTest_Analog.c`

The output state is read and compared with the expected value. When the test is a success, the function returns '0'; otherwise, it returns '1'.

6.13.4 Op-Amp test

While the operational amplifier (Op-Amp) test is not a requirement in the IEC 60730 standard, it is useful while using the operational amplifiers in safety-critical blocks.

The OpAmp test adheres to the principle of "independent output comparison" testing defined in Section H.2.18.8 of the IEC 60730 standard. It provides a fault or error control technique by which inputs or outputs – designed to be within specified tolerances – are compared. The Op-Amp is tested by generating the same output as the input (unity gain buffer) and the result is read by the SAR ADC.

Function

```
uint8 SelfTests_Opamp (SAR_Type* sar_base, int16_t expected_res, int16_t accuracy, uint32_t
opamp_in_channel, bool count_to_mV)
```

Returns:

- 0 – Test pass
- 1 – Test failed

Parameter:

- `sar_base` – Pointer to structure describing SAR instance registers
- `expected_res` – Expected voltage in mV
- `accuracy` – Threshold above and below the expected value that is acceptable
- `opamp_in_channel` – Channel to be scanned (where the output of Op-Amp is connected)
- `count_to_mV` – convert the count to mV

Located in:

- `SelfTest_Analog.h`
- `SelfTest_Analog.c`

The test is a success if the input Op-Amp voltage value is equal to the expected value within the defined accuracy. When the test is successful, the function returns '0'; otherwise it returns '1'.

6.14 Communications UART test

This test implements the UART data loopback test. This loopback can be done internally with smart I/O or wired physically externally. The test is a success if the transmitted byte is equal to the received byte and returns 2. Each function call increments the test byte. After 256 function calls, when the test finishes testing all 256 values and they are all a success, the function returns 3. Smart I/O can be used for loopback for the SCBs that are connected to smart I/O.

It should be noted that not all devices can route RX and TX signals through smart I/O. Please refer to device datasheet for SCB to smart I/O connectivity. However, this test can still be used with all SCBs if the UART RX and TX pins are tied together externally.

Function

```
uint8       SelfTest_UART_SCB(CySCB_Type* base)
```

Returns:

- 1 – test failed
- 2 - Still testing
- 3 - Test completed, OK
- 4 – Error, TX not empty
- 5 – Error, RX not empty
- 6 - Error, UART not enabled

Parameters:

- `base` – Pointer to SCB hardware to configure

Located in:

- `SelfTest_UART_SCB.h`
- `SelfTest_UART_SCB.c`

The input and output terminals switch between the corresponding pins and loop to each other to provide the internal loopback test using the UART multiplexer and demultiplexer. If the receiving or transmitting buffers are not empty before the test, the test is not executed and returns an `ERROR_RX_NOT_EMPTY` or `ERROR_TX_NOT_EMPTY` status.

During the call, the function transmits 1 byte. The transmitted value increments after each function call. The range of test values is from 0x00 to 0xFF.

***Note**: When using smart I/O for the internal loop back, it is up to you to configure smart I/O before the test and restore smart I/O after the test.*

6.15 Communications SPI test

This test implements the SPI data loopback test. This loopback can be done internally with smart I/O or wired physically externally. The test is a success if the transmitted byte is equal to the received byte and returns 2. Each function call increments the test byte. After 256 function calls, when the test finishes testing all 256 values and they are all a success, the function returns '3'.

It should be noted that not all devices can route the MOSI and MISO signals through smart I/O. Please refer to device datasheet for SCB to smart I/O connectivity. However, this test can still be used with all SCBs if the SPI MOSI and MISO pins are tied together externally.

Function

```
uint8 SelfTest_SPI_SCB(CySCB_Type* base)
```

Returns:

- 1 - Test failed
- 2 - Test in progress

- 3 - Test completed
- 4 - TX not empty
- 5 - RX not empty

Parameters:

- `base` – Pointer to SCB hardware to configure

Located in:

- `SelfTest_SPI_SCB.h`
- `SelfTest_SPI_SCB.c`

During the call, the function transmits 1 byte. The transmitted value increments after each function call. The range of test values is from 0x00 to 0xFF.

***Note**: When using smart I/O for the internal loop back, it is up to you to configure smart I/O before the test and restore smart I/O after the test.*

6.16 Communications I2C test

This test requires connection between I2C master and slave externally and uses two SCB blocks.

This project uses an I2C master as the component under test as shown in Figure 5. With no modifications, the project can also be used for testing an I2C slave as both are tested simultaneously.

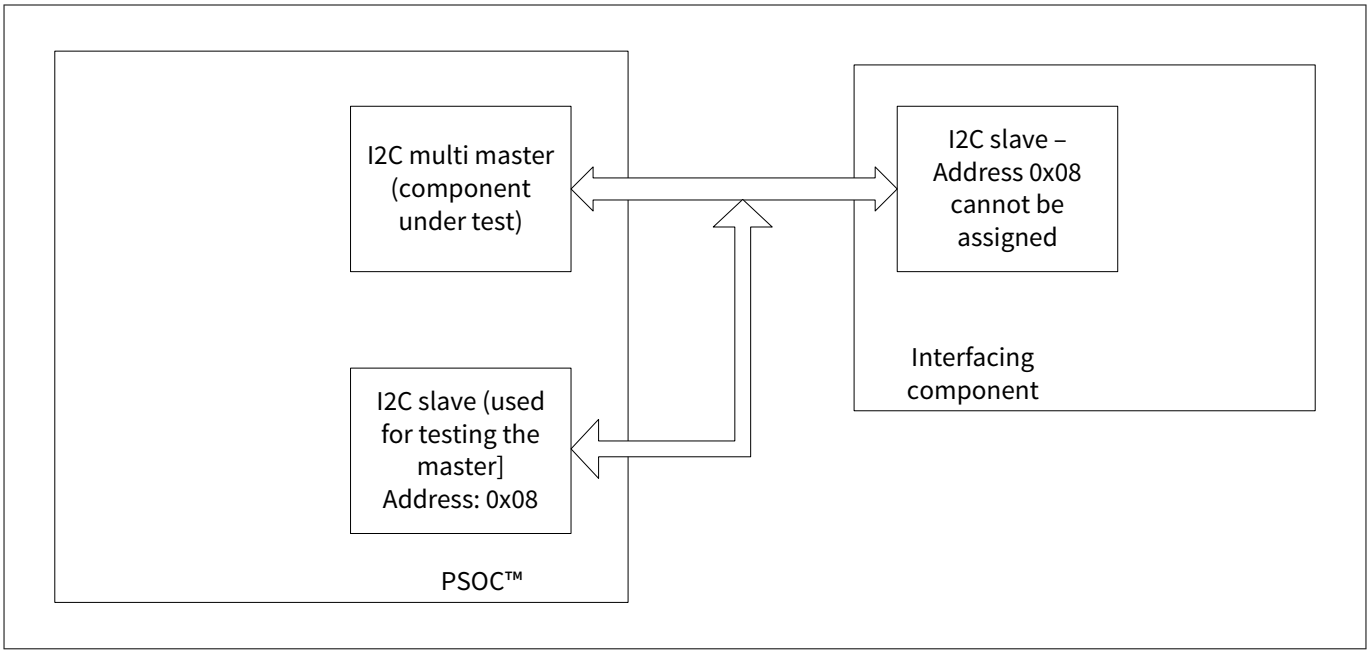


Figure 5 I2C master under test

1. Master periodically sends an I2C data addressing slave [0x08]
2. Slave reads the data, complements, and writes to slave read buffer
3. Master reads the data from the slave and compares it to the complement of the data sent in the last interaction

Function

```
uint8_t SelfTest_I2C_SCB (CySCB_Type* master_base, cy_stc_scb_i2c_context_t* master_context,
CySCB_Type* slave_base, cy_stc_scb_i2c_context_t* slave_context, uint8_t* slave_read_buf,
uint8_t* slave_write_buf)
```

Returns:

- 1 – Test failed
- 2 – Still testing
- 3 - Test completed OK

Parameters:

- `master_base` – Pointer to the master I2C SCB instance
- `master_context` – Pointer to the master I2C SCB context
- `slave_base` – Pointer to the slave I2C SCB instance
- `slave_context` – Pointer to the slave I2C SCB context
- `slave_read_buf` – Pointer to slave read buffer
- `slave_write_buf` – Pointer to slave write buffer

Located in:

- `SelfTest_I2C_SCB.h`
- `SelfTest_I2C_SCB.c`

I2C write API – operation

- Initiates I2C master transaction to write data to the slave
- Waits for write transfer completion
- Reads the slave write buffer
- Performs 1's complement on the read data
- Writes the complemented data to slave read buffer

I2C read API – operation

- Initiates I2C master transaction to read data from the slave
- Waits for read transfer completion
- Checks whether the data read is equal to the complement of data written

6.17 Startup configuration registers test

This test describes and shows an example of how to check the startup configuration registers:

1. Test digital clock configuration registers
2. Test analog configuration registers (set to default values after startup)
3. Test the GPIO configuration and HSIOM registers

These startup configuration registers are typically static and are configured in files prefixed with "cycfg" (for example, `cycfg_clocks.c`). These configuration files can be found in the `GeneratedSource` folder after the project is built. In rare use cases, some of these registers may be dynamically updated. Dynamically updated registers must be excluded from this test. Dynamic registers are instead tested in the application with the knowledge of the current correct value.

Two test modes are implemented in the functions:

- Store duplicates of startup configuration registers in flash memory after device startup. Periodically, the configuration registers are compared with stored duplicates. Corrupted registers can be restored from flash after checking.
- Compare the calculated CRC with the CRC previously stored in flash if the CRC status semaphore is set. If the status semaphore is not set, the CRC must be calculated and stored in flash, and the status semaphore must be set.

***Note**: If you make changes in the device configurator, other configuration registers may be generated in the cycfg files. You must then change the list of required registers based on the registers your configuration modifies.*

Function

```
cystatus SelfTests_Save_StartUp_ConfigReg(void)
```

Returns:

- 0– Write in flash is successful
- 1 – Error detected during flash writing

Located in:

- `SelfTest_ConfigRegisters.c`
- `SelfTest_ConfigRegisters.h`

***Notes**:*

1. *This function should be called once after the initial PSOC™ 4 power up and initialization before entering the main program. It writes the startup configuration register values to flash. After this initial write, typically during manufacturing, the register values are already stored, and this function does not need to be called again*
2. *Make sure the Startup configuration registers or CRC values are not stored in a flash location which is already in use. For example, flash test uses the last 8 bytes to store the flash checksum by default*

Function

```
uint8_t SelfTests_StartUp_ConfigReg(void)
```

Returns:

- 0 – No error
- 1 – Error detected

Located in:

- `SelfTest_ConfigRegisters.c`
- `SelfTest_ConfigRegisters.h`

This function checks the listed startup configuration registers. There are two modes of checking:

- CRC-16 calculation and verification
- Register comparison with duplicated copy

You can define the mode using the parameters in the `SelfTest_ConfigRegisters.h` file:

```
#define STARTUP_CFG_REGS_MODE CFG_REGS_CRC_MODE or CFG_REGS_TO_FLASH_MODE
```

```
#define STARTUP_CFG_REGS_MODE (CFG_REGS_TO_FLASH_MODE)
```

In flash mode, the test stores duplicates of registers in flash and compares registers with the duplicates. It returns a fail (1) if the values are different. Registers can be restored in this mode. The `SelfTests_Save_StartUp_ConfigReg` function is used to store duplicates in flash. `CONF_REG_FIRST_ROW` defines the location of the startup configuration registers in flash memory and is automatically calculated in `SelfTest_ConfigRegisters.h`.

```
#define STARTUP_CFG_REGS_MODE      (CFG_REGS_CRC_MODE)
```

In CRC mode, the function calculates a CRC32 of registers and stores the CRC in flash. Later, function calls recalculate the CRC32 and compare it with the saved value. It returns a fail (1) if the values are different.

By default, the CRC values are stored in the 9th, 7th, and 8th byte locations from the end of flash. While using alongside the flash test, which uses the last 8 bytes of flash, it is recommended to move the CRC values to the 16th, 14th, and 15th byte locations from the end of flash.

You can define custom location for CRC-32 values by updating defines `CRC_STARTUP_SEMAPHORE_SHIFT`, `CRC_STARTUP_LO` in the `SelfTest_ConfigRegisters.h` file.

6.18 Watchdog test

This function implements the watchdog functional test. The function starts the WDT and runs an infinite loop. If the WDT works, it generates a reset. After the reset, the function analyzes the reset source. If the watchdog is the source of the reset, the function returns; otherwise, the infinite loop executes.

Function

```
uint8_t SelfTest_WDT(void)
```

Returns:

- 0 – Test passed
- 1 – Test failed

Located in:

- `SelfTest_WDT.h`
- `SelfTest_WDT.c`

Figure 6 shows the test flowchart.

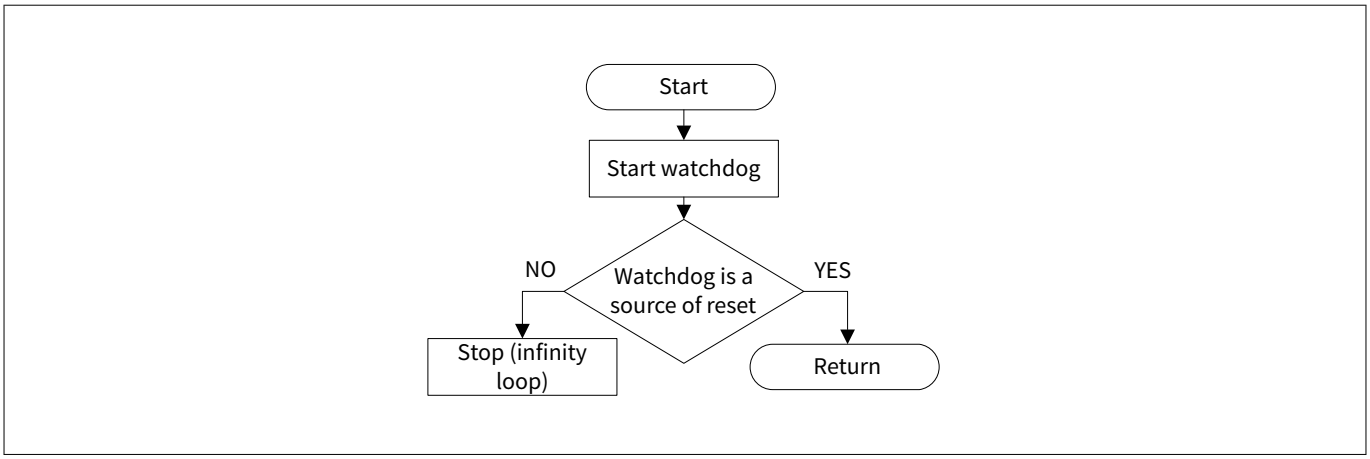


Figure 6 PSOC™ implementation of WDT test

6.19 TCPWM test

This test demonstrates the use of the Class B Safety Test Library to test the TCPWM resource configured as a timer/counter, PWM, and PWM output Gate Kill in PSOC™ 4 MCUs. It verifies the proper operation and accuracy of these peripherals adhering to the IEC 60730 standards.

6.19.1 Timer/counter test

Timer/counter is used to count clocks (timer) or external events (counter). The test uses the timer function. The test should be run on every safety-critical timer/counter.

Function

```
uint8_t SelfTest_Counter_Timer(void)
```

Returns:

- 0 – Test passed
- 1 – Test failed

Function

```
void SelfTest_Timer_Counter_init(TCPWM_Type* base, uint32_t cntNum,
cy_stc_tcpwm_counter_config_t const* config, IRQn_Type intsrc)
```

Returns:

- None

Parameters:

- `base` – Pointer to a TCPWM instance
- `cntNum` – Counter instance number in the selected TCPWM
- `config` – Pointer to the configuration structure
- `Intsrc` – Interrupt source

Located in:

- SelfTest_Timer_Counter.h
- SelfTest_Timer_Counter.c

The timer/counter is tested using the following procedure:

1. Configure the block to timer/counter personality
2. Configure the input clock to the CPU clock
3. Check that the counter is incrementing

6.19.2 PWM test

The PWM is tested using the following procedure:

1. Configure a 16-bit PWM to run at 1/3 duty on, 2/3 off duty cycle with a 1 millisecond period. Start the PWM
2. Run the CPU in a loop for 5 milliseconds. Poll the output continuously in the loop. Count the instances of 0 and 1 output
3. The test is successful if the off/on ratio is between 1 7/8 and 2 1/8. A range is used as the CPU polling loop is asynchronous to the PWM timing

Function 1

```
uint8_t SelfTest_PWM(GPIO_PRT_Type *pinbase, uint32_t pinNum)
```

Returns:

- 0 – Test passed
- 1 – Test failed

Parameters:

`pinbase` – Pointer to a GPIO port instance to which the PWM pin is connected

`pinNum` – GPIO pin number to which the PWM pin is connected

Function 2

```
uint8_t SelfTest_PWM_init(TCPWM_Type* base, uint32_t cntNum, cy_stc_tcpwm_pwm_config_t const*
config, IRQn_Type intr_src)
```

Returns:

- 0 – Initialization successful
- 255 – Initialization failed

Parameters:

`base` – The pointer to a TCPWM instance

`cntNum` – The Counter instance number in the selected TCPWM

`config` – The pointer to the configuration structure

`Intr_src` – Interrupt source

Located in:

- SelfTest_PWM.h
- SelfTest_PWM.c

6.19.3 PWM gate kill test

The "gate kill" function is used in motor controllers and multi-level power converters. When an overvoltage or current state is detected, the gate kill shuts down the output drivers less than 50 nanoseconds after the overvoltage or overcurrent occurs.

Function

```
uint8_t SelfTest_PWM_GateKill(TCPWM_Type *base, uint32_t cntNum)
```

Returns:

- 0 – Test passed
- 1 – Test failed

Parameters:

`base` – Pointer to a TCPWM instance

`cntNum` – Counter instance number in the selected TCPWM

Located in:

- `SelfTest_PWM_GateKill.h`
- `SelfTest_PWM_GateKill.h`

PWM gate kill is tested using the following procedure:

1. Configure kill mode as stop on kill
2. The low-power comparator output or the voltage range violation of the SAR ADC is routed to the kill signal of the TCPWM, indicating an overvoltage/overcurrent condition if the voltage on the +ve terminal is > -ve
3. If an overvoltage or overcurrent condition occurs, the PWM output is killed
4. The TCPWM `base` and `CntNum` are passed to check whether the counter is stopped
5. If the counter is not incrementing/decrementing, the PWM output is inactive

6.20 CAPSENSE™ test

The CAPSENSE™ safety library includes several tests to verify the CAPSENSE™ components and proper functionality of related peripherals. This library is pre-certified by Infineon (per IEC 60730-1) and are available as included as part of the standard CAPSENSE™ component. This library includes tests for verifying:

- Sensor shorts
- External capacitor
- Integrity of CAPSENSE™ configuration data
- Integrity of sensor signal (raw count) and sensor baseline
- VDDA

***Note**: For more details, code examples, and certificate (distributed under NDA), contact Infineon technical support.*

6.21 Communications UART data transfer protocol example

For additional system safety when transferring data between system components, use communication protocols with CRCs and packet handling. An example of safety communication follows.

Data is placed into the packets with a defined structure to be transferred. All packets have a CRC16 calculated with the packet data to ensure the packet's safe transfer. The figure below shows the packet format.

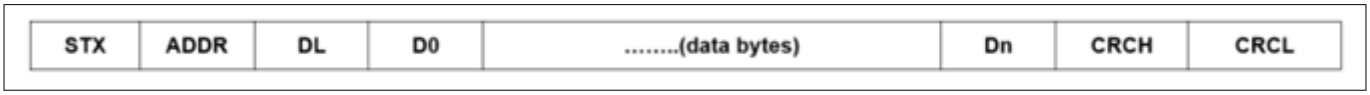


Figure 7 Packet structure

To allow the reserved start of packet marker (STX) value to be transmitted, use a common escape method. When any byte in a packet is equal to STX or ESC, it changes to a 2-byte sequence. If packet byte = ESC, replace it with 2 bytes (ESC, ESC + 1). If any packet byte = STX, then replace it with 2 bytes (ESC, STX + 1). This procedure provides a unique packet start symbol. The ESC byte is always equal to 0x1B. It is not a part of the packet and is always sent before the (packet byte + 1) or (ESC, STX + 1). Table 3 shows the packet field descriptions.

Table 3 Packet field descriptions

Name	Length	Value	Description
STX	1 byte	0x02	Unique start of packet marker = 0x02
ADDR	1 or 2 bytes	0x00...0xFF except 0x02	Slave address. If this byte is equal to STX, it changes to a 2-byte sequence: (ESC) + (STX + 1). If this byte is equal to ESC, it changes to a 2-byte sequence: (ESC) + (ESC +1)
DL	1 or 2 bytes	0x00...0xFF except 0x02	Data length of packet (without protocol bytes). If this byte is equal to STX, it changes to a 2-byte sequence: (ESC) + (STX + 1). If this byte is equal to ESC, it changes to a 2-byte sequence: (ESC) + (ESC + 1)
D0...Dn (data)	1...510 bytes	0x00...0xFF except 0x02	Packet's data. If any byte in the data is equal to STX, it changes to a 2-byte sequence: (ESC) + (STX + 1). If any byte in the data is equal to ESC, it changes to a 2-byte sequence: (ESC) + (ESC + 1)
CRCH	1 or 2 bytes	0x00...0xFF except 0x02	MSB of packet CRC. CRC-16 is used. CRC is calculated for all packet bytes from ADDR to the last data byte. CRC is calculated after the ESC changing procedure. If this byte is equal to STX, it changes to a 2-byte sequence: (ESC) + (STX + 1). If this byte is equal to ESC, it changes to a 2-byte sequence: (ESC) + (ESC + 1)
CRCL	1 or 2 bytes	0x00...0xFF except 0x02	LSB of packet CRC. CRC-16 is used. CRC is calculated for all packet bytes from ADDR to the last data byte. CRC is calculated after the ESC changing procedure. If this byte is equal to STX, it changes to a 2-byte sequence: (ESC) + (STX + 1). If this byte is equal to ESC, it changes to a 2-byte sequence: (ESC) + (ESC + 1)

6.21.1 Data delivery control

The communication procedure can be divided into three parts:

- Send request (opposite side receives request)
- Wait for response (opposite side analyzes request)
- Receive response (opposite side sends response)

"Send request" consists of sending STX, sending the data length and data using the byte changing procedure, calculating the CRC, and sending the CRC.

"Receive response" consists of finding the STX and starting the CRC calculation. If the received address is invalid, the search for the STX byte is repeated. If the address is valid, the data length and data bytes are received. The CRC counter then stops and two CRC bytes are received. These bytes are compared with the calculated CRC value.

After sending a request, the guard timer is started to detect if a response is not received within the timeout period.

6.21.2 PSOC™ implementation

The UART SCB Components are used to physically generate the signals. The software CRC-16 calculation is applied to every sent/received byte (except STX and the CRC itself). To detect an unsuccessful packet transaction, the timer is used.

Three interrupts implemented in this project provide a fully interrupt-driven background process:

- The transmit interrupt in the UART is configured for a FIFO not full event to take the new data from the RAM and place it into the TX buffer, and for a transmit complete event to start or stop the CRC calculation
- The receive interrupt in the UART is configured for a FIFO not empty event to analyze the received data, calculate the CRC, and store the received data into RAM
- The timer interrupt is used to detect the end of an unsuccessful transmission

This software is implemented as an interrupt-driven driver. That is, the user only starts the process and checks the state of the unit. All operation is done in the background.

Four functions for working with the protocol unit for the master

Function 1

```
void UartMesMaster_Init(CySCB_Type* uart_base, TCPWM_Type* counter_base,  uint32_t cntNum)
```

Returns: None

Parameters:

- `uart_base` – Pointer to the master UART SCB instance
- `counter_base` – Pointer to the master TCPWM instance
- `cntNum` – Counter instance number in the selected TCPWM

Located in:

- `SelfTest_UART_master_message.h`
- `SelfTest_UART_master_message.c`

This function initializes the UART message unit.

Function 2

```
uint8_t UartMesMaster_DataProc(uint8 address, char * txd, uint8 tlen, char * rxd, uint8 rlen)
```

Returns:

- 0 – If the unit started a message process
- 1 – If not (because the unit is busy or data input is invalid)

Parameter:

- **address** – Slave address for data transfer
- **txd** – Pointer to the transmitted data
- **tlen** – Size of the transmitted data in bytes

- **rxd** – Pointer to the buffer where the received data will be stored
- **rlen** – Size of the received data buffer

Located in:

- `SelfTest_UART_master_message.h`
- `SelfTest_UART_master_message.c`

This function starts the process of transmitting and receiving messages and returns the result of the process start: '0' = success and '1' = error. An error can occur because the unit is already busy sending a message or a null transmitting length was detected.

Function 3

```
uint8_t UartMesMaster_State(void)
```

Returns:

- 0 (UM_ERROR) – Last transaction process finished with an error, and the unit is ready to start a new process
- 1 (UM_COMPLETE) – Last transaction process finished successfully; the received buffer contains a response. The unit is ready to start a new process
- 2 (UM_BUSY) – Unit is busy with an active transaction operation

Located in:

- `SelfTest_UART_master_message.h`
- `SelfTest_UART_master_message.c`

This function returns the current state of the UART message unit.

Function 4

```
uint8_t UartMesMaster_GetDataSize(void)
```

Returns:

- Data size

Located in:

- `SelfTest_UART_master_message.h`
- `SelfTest_UART_master_message.c`

This function returns the received data size that is stored in the receive buffer. If the unit is busy or the last process generated an error, it returns '0'.

Five functions for working with the protocol unit for the slave

Function 1

```
void UartMesSlave_Init(CySCB_Type* uart_base, uint8_t address)
```

Returns:

- None

Parameter:

- `uart_base` – Pointer to the slave UART SCB instance
- `address` – Slave bus address

Located in:

- `SelfTest_UART_slave_message.h`
- `SelfTest_UART_slave_message.c`

This function initializes the UART message unit.

Function 2

```
uint8_t UartMesSlave_Respond(uint8_t * txd, uint8 tlen)
```

Returns:

- 0 – If the unit starts a response process
- 1 – If not (because of IDLE or RESPOND state)

Parameter:

- `txd`: Pointer to the transmitted data (request data)
- `tlen`: Length of the request in bytes

Located in:

- `SelfTest_UART_slave_message.h`
- `SelfTest_UART_slave_message.c`

This function starts the response process. It returns the result of the process start. Success is '0', and error is '1' (the unit has not received a marker).

Function 3

```
uint8_t UartMesSlave_State(void)
```

Returns:

- 0 (UM_IDLE) – The last transaction process is finished
- 1 (UM_PACKREADY) – The unit has received a marker and there is received data in the buffer. The master waits for a response
- 2 (UM_RESPOND) – The unit is busy with sending a response

Located in:

- `SelfTest_UART_slave_message.h`
- `SelfTest_UART_slave_message.c`

Function 4

```
uint8_t UartMesSlave_GetDataSize(void)
```

Returns:

- Data size

Located in:

- `SelfTest_UART_slave_message.h`
- `SelfTest_UART_slave_message.c`

This function obtains the received data size that was stored in the receive buffer. If the unit state is not UM_PACKREADY, it returns 0.

Function 5

```
uint8_t * UartMesSlave_GetDataPtr(void)
```

Returns:

- Pointer to data

Located in:

- `SelfTest_UART_slave_message.h`
- `SelfTest_UART_slave_message.c`

This function returns a pointer to the received data.

6.22 Communications CAN FD/ CAN test

The CAN FD or CAN block is tested using the loopback capability and the test is as follows:

1. Enable loopback externally or internally based on the parameter passed
2. Configure CAN FD or CAN to accept message IDs in the range of 0x50 to 0x55 in any of the Rx FIFO (1 or 2)
3. Transmit a message with message ID 0x60
4. Verify the Rx FIFO is empty
5. Transmit a message with message ID 0x52
6. Verify the message is received correctly in the Rx FIFO

Function

```
uint8_t SelfTest_CANFD(CANFD_Type* base, uint32_t chan, const
        cy_stc_canfd_config_t* config, cy_stc_canfd_context_t* context, cy_stc_canfd_test_mode_t
        test_mode)
```

Returns:

- 0 – Test passed
- 1 – Test failed

Parameter:

- `base`: The pointer to a CAN FD instance
- `chan`: The CAN FD channel number
- `config`: The pointer to the CAN FD configuration structure
- `context`: The pointer to the user-allocated context structure
- `test_mode` - Internal: Does not drive the pin; External: Drives the external pins along with loopback

Located in:

- `SelfTest_CANFD.h`
- `SelfTest_CANFD.c`

Function

```
uint8_t SelfTest_CAN(CAN_Type* base, const cy_stc_can_config_t* config,
        cy_stc_can_context_t* context, cy_stc_can_test_mode_t testMode)
```

Returns:

- 0 – Test passed
- 1 – Test failed

Parameter:

- `base`: The pointer to a CAN instance
- `config`: The pointer to the CAN configuration structure
- `context`: The pointer to the user-allocated context structure
- `testMode` - The test mode to be used for the self-test

Located in:

- `SelfTest_CAN.h`
- `SelfTest_CAN.c`

6.23 DMA test

The direct memory access (DMA) blocks are tested using the following procedure:

1. A destination block of size 64 bytes is set to '0' with DMA transfers using 16 x 32-bit transfers from a fixed address
2. The destination block is verified to be all '0'
3. The same destination block is filled with 00 00 ff by using an 8-bit DMA from a fixed address with an
4. increment of 1 and a length of 22 (64+(3-1))/3
5. The destination block is verified to contain the correct pattern (shown below with lowest address first):

   `ff0000ff0000ff0000ff0000ff0000ff0000ff0000ff0000…`

Function

```
uint8_t SelfTest_DMAC(DMAC_Type* base, uint32_t channel, uint32_t
        trigLine)
```

Returns:

- 0 – Test passed
- 1 – Test failed

Parameter:

- `base`: The pointer to the hardware DMAC block
- `channel`: The DMAC channel number
- `trigLine`: The input trigger to start data transfer for the selected DMAC channel

Located in:

- `SelfTest_DMAC.h`
- `SelfTest_DMAC.c`

7 Test files

Table 4 Test files and versions

File name	Version	MD5 Hash HEX
ModusToolbox™ Safety Test Library	3.7.0.7205	10f973ab4b3f443aab09b994c37cb16387d996c2
CAPSENSE™ Middleware Library	11.0.0.9453	36580294e1587b2df8193f254f7e9b9b4aa6dd13

8 Summary

This application note describes how to implement diagnostic tests defined by the IEC 60730 and IEC 61508 standards. Incorporation of these standards into the design of white goods and other appliances will add a new level of safety for consumers.

By taking advantage of the unique hardware configurability offered by Infineon's PSOC™ 4, designers can comply with regulations while maintaining or reducing electronic systems cost. Using PSOC™ and the safety software library enables the creation of a strong system-level development platform to achieve superior performance, fast time-to-market, and energy efficiency.

A Appendix A: Set checksum in flash (invariable memory) test

The following instructions help you program your part for proper flash and ROM diagnostic testing:

1. Build a project in ModusToolbox™ with the stored checksum value set to 0x0000 in the `SelfTest_Flash.c` file.

   For the GCC compiler:

   ```
   static volatile const uint64_t flash_StoredCheckSum __attribute__((used,
   section(".flash_checksum"))) = 0x00000000UL;
   ```

2. Read the calculated flash checksum

   a. Read the checksum in debug mode

      1. Open the project file `SelfTest_Flash.c` and set the breakpoint in debug mode to the line shown in the figure below
      2. Press [F10] to single step past the breakpoint location and hover the mouse over the `flash_CheckSum` variable. A value stored in this variable, as shown in Figure 8, should appear

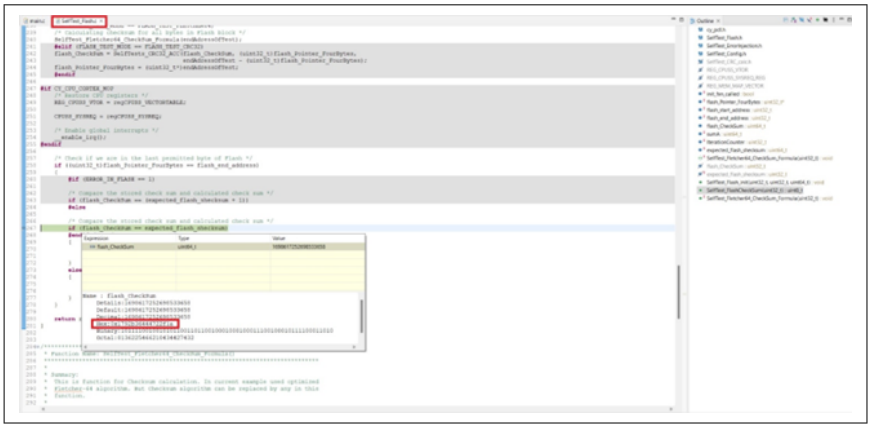


Figure 8 Stored checksum in debug mode

   b. Read the checksum using the communication protocol:

   - To speed up the process of testing the flash checksum outputs, use a UART. This feature is implemented in Class B firmware. It prints the calculated checksum value when the stored flash checksum does not match the calculated flash checksum. To use this project, set the UART parameters shown in Figure 8

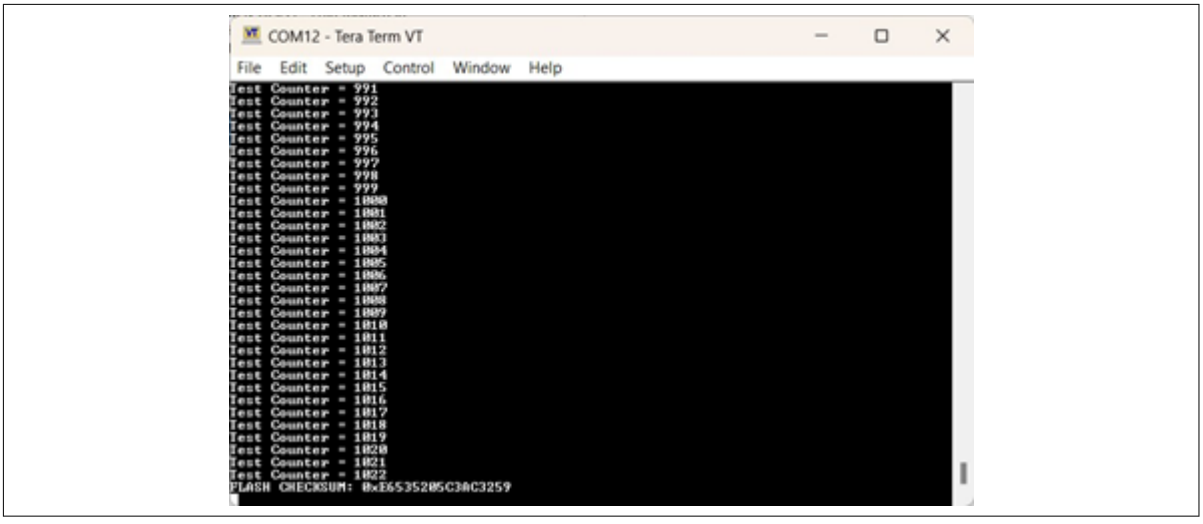


Figure 9 Checksum output using UART

3. Copy this checksum value and store it in the checksum location in the source file

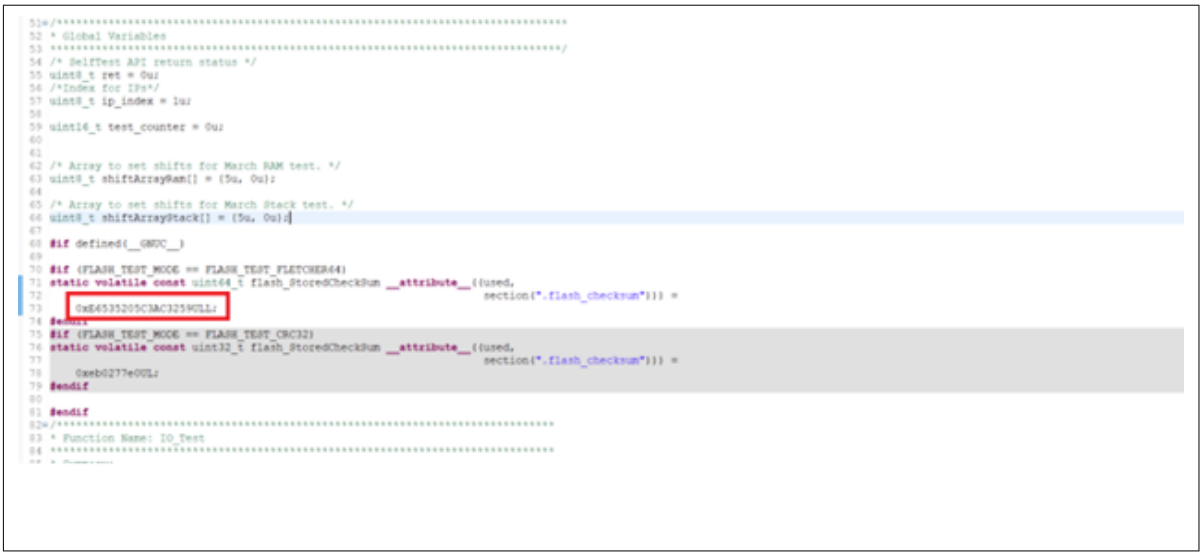


Figure 10 Reassign checksum constant with actual checksum

4. Compile the project and program the PSOC™ 4 device

B Appendix B: IEC 60730-1 certificates of compliance

***Note**: The public facing UL website (UL Product iQ) is updated to reflect the latest PSOC™ 4 Class B certificate.*



IEC IECEE

Ref. Certif. No.

US-42301-UL

IEC SYSTEM FOR MUTUAL RECOGNITION OF TEST CERTIFICATES FOR ELECTRICAL EQUIPMENT (IECEE) CB SCHEME

CB TEST CERTIFICATE

Product	Software Libraries
Name and address of the applicant	CYPRESS SEMICONDUCTOR CORP 198 CHAMPION CT SAN JOSE, CA 95134-1709 United States
Name and address of the manufacturer	CYPRESS SEMICONDUCTOR CORP 198 CHAMPION CT SAN JOSE, CA 95134-1709 United States
Name and address of the factory *Note: When more than one factory, please report on page 2*	CYPRESS SEMICONDUCTOR CORP 198 CHAMPION CT SAN JOSE, CA 95134-1709 United States ☐ Additional Information on page 2
Ratings and principal characteristics	Not Applicable
Trademark / Brand (if any)	Not Applicable
Customer's Testing Facility (CTF) Stage used	
Model / Type Ref.	Integrated, Protective Control with Type 2 action (Self-Test Software Library – Safety Control), PSoC 4 IEC 60730 Class B Safety Software Library (AN36847)
Additional information (if necessary may also be reported on page 2)	Only the requirements of IEC 60730-1, as documented in the accompanying test report, have been addressed. ☐ Additional Information on page 2
A sample of the product was tested and found to be in conformity with	IEC 60730-1:2013, IEC 60730-1:2013/AMD1:2015, IEC 60730-1:2013/AMD2:2020
As shown in the Test Report Ref. No. which forms part of this Certificate	4790880026-20230619 issued on 2023-06-19

This CB Test Certificate is issued by the National Certification Body

UL Solutions

UL Solutions (US), 333 Pfingsten Rd IL 60062, Northbrook, USA
UL Solutions (Demko), Borupvang 5A DK-2750 Ballerup, DENMARK
UL Solutions (JP), Marunouchi Trust Tower Main Building 6F, 1-8-3 Marunouchi, Chiyoda-ku, Tokyo 100-0005, JAPAN
UL Solutions (CA), 7 Underwriters Road, Toronto, M1R 3B4 Ontario, CANADA
For full legal entity names see www.ul.com/ncbnames

Date: 2023-06-21

Signature: Jolanta M. Wroblewska

Figure 11 UL compliance certificate



10 YEARS OF PRODUCT SAFETY

VDE

INSTITUTE

VDE Prüf- und Zertifizierungsinstitut GmbH • Merianstr. 28 • 63069 Offenbach

Cypress Semiconductor Corporation
198 Champion Court
[redacted]
SAN JOSE CA 95134
USA

Offenbach, 2024-09-02

Your ref.	Your letter	Our ref. - please indicate	Contact
[redacted]	2023-10-20	5030935-4970-0001/313289 TL2/swa	[redacted]

Micro controller
Translation: In any case the German version shall prevail

PRÜFBERICHT
zur Information des Auftraggebers

Test Report for the Information of the applicant

I Produkt / *Product*:

Typ / *Type*: **PSoC™ 4 MCU familie/family**
60730-1 Selbstdiagnose Bibliothek Klasse B und CPASENSE™ Test.
60730-1 Self diagnostic library class B and CAPSENSE™ test.

Dear Sirs,

dieser Prüfbericht enthält das Ergebnis einer einmaligen Untersuchung an dem zur Prüfung vorgelegten Erzeugnis. Ein Muster dieses Erzeugnisses wurde geprüft, um die Übereinstimmung mit den nachfolgend aufgeführten Normen bzw. Abschnitten von Normen festzustellen.
Die Prüfung wurde durchgeführt vom 2024-05-05 to 2024-07-28.

This test report contains the result of a singular investigation carried out on the product submitted. A sample of this product was tested to found the accordance with the thereafter listed standards or clauses of standards resp. The testing was carried out from 2024-05-05 to 2024-07-28.

VDE Prüf- und Zertifizierungsinstitut GmbH
Merianstr. 28
63069 Offenbach, Germany
Phone +49 69 8306-0
Fax +49 69 8305-555
www.vde.com/institute
vde-institut@vde.com
Commerzbank AG Frankfurt
IBAN DE91 5008 0000 0198 0270 00
BIC DRES DE FF XXX
VAT-ID DE 261 922 990
Tax number 04425092556
Managing Directors:
Ansgar Hinz
Dr. jur. Beate Mand
Location: Offenbach
Registered at the local court Offenbach am Main,
No. HRB 43618

1/9

Figure 12 VDE compliance report (Page 1)

Der Prüfbericht berechtigt Sie nicht zur Benutzung eines Zertifizierungszeichens des VDE und berücksichtigt ausschließlich die Anforderungen der unten genannten Regelwerke.

The test report does not entitle for the use of a VDE Certification Mark and considers solely the requirements of the specifications mentioned below.

Wenn gegenüber Dritten auf diesen Prüfbericht Bezug genommen wird, muss dieser Prüfbericht in voller Länge an gleicher Stelle verfügbar gemacht werden.

Whenever reference is made to this test report towards third party, this test report shall be made available on the very spot in full length.

Beschreibung / Description

Bibliothek / Library

Die Selbstdiagnose Methoden decken die Anforderungen der Tabelle H2 der 60730-1 Abschnitt 7.2 ab. Siehe Kapitel „Methoden".

The self-diagnose methods are covering the requirements of Table H2 of 60730-1 Clause 7.2. see chapter "methods"

Die CAPSENSE™ Bibliothek der PSoC™4 Familie ist für die Erzeugung kapazitiver Sensorfelder vorgesehen.

The CAPSENSE library is *foreseen to be used to create capacitive sensor fields in conjunction with the micro controller family* PSoC™4

Das Paket bietet Methoden zur Erkennung und Behandlung typischer Fehler in kapazitiven Sensortastensteuerungen.

The package provides methods to detect and handle typical failures in capacitive sensor button controls.

Methoden des Diagnosepakets

- CPU Registertest
- Startup Configuration Register Test
- Program counter Test
- Programm Ablaufüberwachung
- Interrupt Überwachung
- Clock Test
- Flash Test
- SRAM Test
- March C Test
- Stack Test
- Digital I/O Test
- ADC Test
- Comparator Test
- Opamp Test
- UART Test

Figure 13 VDE compliance report (Page 2)

- SPI Test
- Capsense™ tests*

Methods of the diagnostic package

- CPU register test
- Startup configuration register test
- Program counter Test
- Program flow test
- Interrupt test
- Clock test
- Flash Test
- SRAM Test
- March C Test
- Stack Test
- Digital I/O Test
- ADC Test
- Comparator Test
- Opamp Test
- UART Test
- SPI Test
- Capsense™ tests*

English only

*

7.18 CAPSENSE™ test

The CAPSENSE™ safety library includes several tests to verify the CAPSENSE™ components and proper functionality of related peripherals. This library is pre-certified by Infineon (per IEC 60730-1) and are available as included as part of the standard CAPSENSE™ component. This library includes tests for verifying:

- Sensor shorts
- External capacitor
- Integrity of CAPSENSE™ configuration data
- Integrity of sensor signal (raw count) and sensor baseline
- VDDA

Figure 14 VDE compliance report (Page 3)

TABLE H.2 – MEASURES TO ADDRESS FAULT/ERRORS (Software Class B)

Component	Fault/error	Declared measures	Verdict
1. CPU	-	-	-
1.1 Registers	Stuck at	CPU register test	P
1.3 Program counter	Stuck at	Program counter test Program flow test	P
2. Interrupt handling and execution	No interrupt	Interrupt test	P
	Too frequent interrupt	Interrupt test	P
3. Clock	Wrong frequency (for quartz synchronized clock: harmonics/ sub-harmonics only)	Clock test	P
4. Memory	-	-	-
4.1 Invariable memory	All single bit faults	FLASH Test	P
4.2 Variable memory	DC fault	March C test	P
4.3. Addressing (relevant to variable and invariable memory)	Stuck at	With 4.1 and 4.2	P
5. Internal data path	-	-	-
5.1 Data	Stuck at	With 4.1 and 4.2	P
5.2 Addressing	Wrong address	With 4.1 and 4.2	P
6. External communication	-	-	-
6.1 Data	Hamming distance 3	Loopback tests for UART and SPI, further tests may be necessary in the final application.	P
6.2 Addressing	Wrong address	N/A	N/A
6.3 Timing	Wrong point in time	N/A	N/A
	Wrong sequence	N/A	N/A
7. Input/output periphery	-	-	-

VDE Prüf- und Zertifizierungsinstitut GmbH
Merianstr. 28
63069 Offenbach, Germany
Phone +49 69 8306-0
Fax +49 69 8306-555
www.vde.com/institute
vde-institut@vde.com
Commerzbank AG Frankfurt
IBAN DE91 5008 0000 0198 0270 00
BIC DRES DE FF XXX
VAT-ID DE 261 922 990
Tax number 04425092555
Managing Directors
Ansgar Hinz
Dr. jur. Beate Mand
Location Offenbach
Registered at the local court Offenbach am Main,
No. HRB 43618

4/9

Figure 15 VDE compliance report (Page 4)

7.1 Digital I/O	Fault conditions specified in Cl.H.13	Errors to Vcc and Gnd Further failures may need to be considered in the final application	P
7.2 Analog I/O	-	-	-
7.2.1 A/D and D/A-convertor	Fault conditions specified in Cl. H.13	ADC test	P
7.2.2 Analog multiplexer			N/A
9. Custom chips e.g. ASIC, GAL, gate array	Any output outside the static and dynamic functional specification		N/A

Figure 16 VDE compliance report (Page 5)

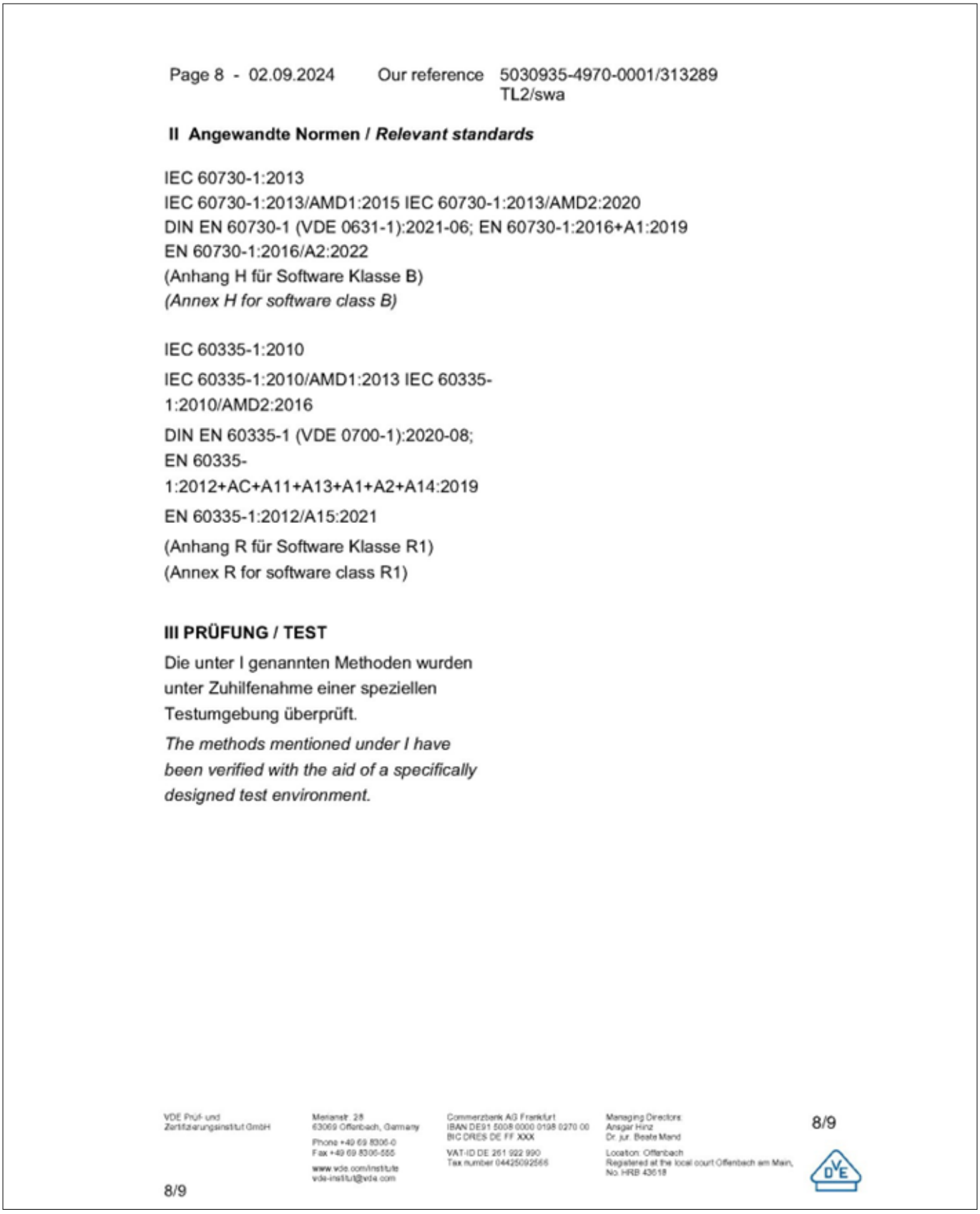
Test File List

File Name	Version	MD5 Hash HEX
SelfTest_Analog.c	2.0	4DE4F05A48A83AB4EEA8F65AB6C52837
SelfTest_Analog.h	2.0	0CE91504AA03F1A1574951968F57A8EC
SelfTest_Clock.c	2.0	19BD8BA39F135841DE0FDAF4371C665E
SelfTest_Clock.h	2.0	635C7258BC068DA5737A9987B750CE76
SelfTest_ConfigRegisters.c	2.0	132610656813C5AF71C661DD869B2643
SelfTest_ConfigRegisters.h	2.0	102E012419ED441C6CB712DEC1795CDA
SelfTest_CPU.c	2.0	AA91A7068A395C88F03F333A2FEACFFF
SelfTest_CPU.h	2.0	1DA7BDE5CB1C36C24201FD43131FCD24
SelfTest_CPU_Regs_GCC.s	2.0	3ACC8B37F9A9337E2DCE381EE8821305
SelfTest_CPU_Regs_IAR.s	2.0	09D5CB4B6F931FB1B32B0443D9860FD9
SelfTest_CPU_Regs.h	2.0	67E394266C180F03A9EB417C68C2D85B
SelfTest_CRC_calc.c	2.0	F1D39B0681B65B2C858194006C9CA323
SelfTest_CRC_calc.h	2.0	76CF6CB094006FD85CA0304EE3415A70
SelfTest_Flash.c	2.0	B0EA5A9F14CD938341F6EAE17F389DF6
SelfTest_Flash.h	2.0	D97934D4B6BD2F951ED9C7A71A524EC4
SelfTest_Interrupt.c	2.0	76A401EBAFB9B78C49F6160C1CA9B3FB
SelfTest_Interrupt.h	2.0	E6AF3F2ECAF346007BA01E69D2C61108
SelfTest_IO.c	2.0	FC20FDB410949D724D254E94AD4DF501
SelfTest_IO.h	2.0	8407D43F8C1A7C8DDA2133AF7740131E
SelfTest_RAM.c	2.0	0BC17EEA7D603EE04E7824535AB03B67
SelfTest_RAM.h	2.0	6D38C82A5E59DCF51B3FC5DDBDC595A8
SelfTest_SRAM_March_GCC.s	2.0	7D0BE45C70C18B30E42E5E93F2D24197
SelfTest_SRAM_March_IAR.s	2.0	F14E5CA9C2CD5BEBFB79601FB65BE8EA
SelfTest_SRAM_Config_GCC.inc	2.0	5ED1FDCBBA4145ED73EF4D80302E1791
SelfTest_SRAM_Config_IAR.inc	2.0	665FC0E6A58D63C70E52DCBE2971016E
SelfTest_Stack.c	2.0	E3036FFCAE03340DAB268CF979AFDE38
SelfTest_Stack.h	2.0	4C1871E98228FF4C93701C90339A1ACC
SelfTest_UART_SCB.c	2.0	8835286EB479506326DE55F564856C78
SelfTest_UART_SCB.h	2.0	A3DDE512B924F7EB852C52AE1660C57E

Figure 17 VDE compliance report (Page 6)

SelfTest_I2C_SCB.c	2.0	1513172CBA33CC63772F39B3DCF7852D
SelfTest_I2C_SCB.h	2.0	0D81FBBE1DEDB1927767266E340F1EDA
SelfTest_SPI_SCB.c	2.0	C219A9F3AB9E644578AABC335D5AF015
SelfTest_SPI_SCB.h	2.0	135C0D97BF5E6102E7202D7F6CC198B3
SelfTest_WDT.c	2.0	5702BD9DF5789F935E5C949045BD1052
SelfTest_WDT.h	2.0	2B71FB9B5011955CB8F9BF09377C8AE6
UART_slave_message.c	2.0	3CCA9666FF5A72D4B5CBC697E5D3295B
UART_master_message.c	2.0	82EB11D06640F61148A8971BEAAE4FCF
UART_Debug.c	2.0	EB121120BB39A48B5FC9E568556BD0C9
UART_slave_message.h	2.0	5F0C214005CE0F5BC305D62C7750F071
UART_master_message.h	2.0	4961BC5D5E5CBAF7AC331A8BFAEEAAF2
UART_Debug.h	2.0	FE20C7C33C6191270B23212F857A07CA
TestConfiguration.h	2.0	80B0D9192726F41D7A163242DDBC180D
Cy_capsense_selftest_v2.c	4.0	6D1B9AF86501F9AA57FC54F685E955E4
Cy_capsense_selftest_v2.h	4.0	BB9FD9216117AFC3E7471725CE6725FB
Cy_capsense_selftest_v3.c	4.0	7740A68DBDCA2E6172371790558F7452
Cy_capsense_selftest.c	4.0	D75F821B05BD0F0F2A872BCE759B7C05
Cy_capsense_selftest.h	4.0	BBC6CAA165B01DA4BD494176F9315998

Figure 18 VDE compliance report (Page 7)



Page 8 - 02.09.2024 Our reference 5030935-4970-0001/313289
TL2/swa

II Angewandte Normen / *Relevant standards*

IEC 60730-1:2013
IEC 60730-1:2013/AMD1:2015 IEC 60730-1:2013/AMD2:2020
DIN EN 60730-1 (VDE 0631-1):2021-06; EN 60730-1:2016+A1:2019
EN 60730-1:2016/A2:2022
(Anhang H für Software Klasse B)
(Annex H for software class B)

IEC 60335-1:2010
IEC 60335-1:2010/AMD1:2013 IEC 60335-1:2010/AMD2:2016
DIN EN 60335-1 (VDE 0700-1):2020-08;
EN 60335-1:2012+AC+A11+A13+A1+A2+A14:2019
EN 60335-1:2012/A15:2021
(Anhang R für Software Klasse R1)
(Annex R for software class R1)

III PRÜFUNG / TEST

Die unter I genannten Methoden wurden unter Zuhilfenahme einer speziellen Testumgebung überprüft.
The methods mentioned under I have been verified with the aid of a specifically designed test environment.

VDE Prüf- und Zertifizierungsinstitut GmbH
Merianstr. 28
63069 Offenbach, Germany
Phone +49 69 8306-0
Fax +49 69 8306-555
www.vde.com/institute
vde-institut@vde.com
Commerzbank AG Frankfurt
IBAN DE91 5008 0000 0198 0270 00
BIC DRES DE FF XXX
VAT-ID DE 261 922 990
Tax number 04425092555
Managing Directors
Ansgar Hinz
Dr. jur. Beate Mand
Location: Offenbach
Registered at the local court Offenbach am Main, No. HRB 43618
8/9
8/9

Figure 19 VDE compliance report (Page 8)

IV ERGEBNIS / *RESULT*

Die unter I benannten Selbst-Diagnose-Routinen erfüllen die Anforderungen der unter II benannten Normen. Die unter I benannten Selbst-Diagnose-Routinen können zum Aufbau einer Selbst-Test Bibliothek gemäß der unter II benannten Normen verwendet werden.

The self-diagnostic routines mentioned under I fulfill the requirements of the standards mentioned under II. The self-diagnostic routines mentioned under I are suitable to be used to create a self-test library according the standards mentioned under II.

Mit freundlichen Grüßen





Figure 20 VDE compliance report (Page 9)

B Appendix B: IEC 60730-1 certificates of compliance

***Note**: VDE certification was completed using an earlier version of the mtb-stl library and does not apply to the latest version referenced in Section 7.*

C Appendix C: Supported part numbers

Table 5 PSOC™ 4 families supported by the certified libraries

Device family	Datasheet	Revision
PSOC™ 4100S	PSOC™ 4 MCU: PSOC™ 4100S	*Q
PSOC™ 4100S Max	PSOC™ 4 MCU: PSOC™ 4100S Max	*F
PSOC™ 4100S Plus 256 KB	PSOC™ 4 MCU: PSOC™ 4100S Plus 256 KB	*H
PSOC™ 4500S	PSOC™ 4 MCU: PSOC™ 4500S	*E
PSOC™ 4000S	PSOC™ 4 MCU: PSOC™ 4000S	*L
PSOC™ 4000T	PSOC™ 4 MCU: PSOC™ 4000T	*M
PSOC™ 4100S Plus	PSOC™ 4 MCU: PSOC™ 4100S Plus	*L
PSOC™ 4100T Plus	PSOC™ 4 MCU: PSOC™ 4100T Plus	*H
PSOC™ 4700S	PSOC™ 4 MCU: PSOC™ 4700S	*E

For details on the supported part numbers, see the “Ordering Information” section of the corresponding device datasheet.

D Appendix D: MISRA compliance

Table 6 and Table 7 in this appendix provide details on MISRA-C:2004 compliance and deviations for the test projects.

The motor industry software reliability association (MISRA) specification covers a set of 122 mandatory rules and 20 advisory rules that apply to firmware design. The automotive industry compiled it to enhance the quality and robustness of the firmware code embedded in automotive devices.

Table 6 Verification environment

Component	Name	Version
Test specification	MISRAC:2012 guidelines for the use of the C language in critical systems	3rd edition, Feb 2019
Target device	PSOC™ 4100S, PSOC™ 4100S Max, PSOC™ 4100S Plus 256 KB, PSOC™ 4500S, PSOC™ 4000S, PSOC™ 4000T, PSOC™ 4100S Plus, PSOC™ 4100T Plus, PSOC™ 4700S	Production
Target compiler	GCC	v14.2.1
	ARM	v6.22
	IAR	v9.70.4
Generation tool	ModusToolbox™	v3.9
Peripheral Driver Library	PDL	mtb-pdl-cat2 V2.1.0
MISRA checking tool	Coverity Static Analysis Tool	2021.9.0

Table 7 Deviated rules

MISRA-C:2012 rule	Rule class (R/A)	Rule description	Description of deviation(s)
1.2	A	Language extensions should not be used.	Using low-level compiler-specific commands are required. Confirmed correct for each supported compiler
2.5	A	A project should not contain unused macro declarations	Not all features of library are able to be enabled at the same time, so macros required for disabled tests appear unused
3.1	R	The character sequences /* and // shall not be used within a comment	Hyper Link used with “/” and “//”. Only included in comments for use in documenting the library. Not used in code
4.9	A	A function should be used in preference to a function-like macro where yet are interchangeable	Using function-like macros is unavoidable because of platform PDL usage
8.6	R	An identifier with external linkage shall have exactly one external definition	Declared but never defined in C. Defined in assembly files

(table continues...)

Table 7 (continued) Deviated rules

MISRA-C:2012 rule	Rule class (R/A)	Rule description	Description of deviation(s)
8.7	A	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit	Remove global variables as possible based on safety library integration with user project
8.9	A	An object should be defined at block scope if its identifier only appears in a single function	Scope is dependent on library integration in user project
8.13	A	A pointer should point to a const-qualified type whenever possible	False positives
11.4	A	A conversion should not be performed between a pointer to object and an integer type	Casting pointer to integer and vice versa unavoidable due to PDL use requirements
13.2	R	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders	Using a volatile variable more than once in an evaluation. This is ok since the only time the volatile variable has more than 1 operation per evaluation is in the ISR
13.5	R	The right hand operand of a logical && or || operator shall not contain persistent side effects	Using volatile variable in comparison is unavoidable due to testing in the ISR
14.2	R	A for loop shall be well-formed	False positive
14.3	R	Controlling expressions shall not be invariant	Boolean expression always evaluates true unavoidable due to use of PDL

Related resources

- Webpage: PSOC™ 4 Arm® Cortex®-M0/M0+
- Infineon Developer Community: PSOC™ 4 MCU forum

References

[1] IEC 60730 Standard: *Automatic electrical controls, IEC 60730-1 Edition 6.0, 2022-09*; Available online

[2] IEC 61508 Standard: *Functional safety of electrical/electronic/programmable electronic safety-related systems, IEC 61508-2 Edition 2.0, 2010-04*; Available online

Revision history

Document revision	Date	Description of changes
**	2023-07-28	Initial release
*A	2023-10-12	Updated ‘Safety software library’ section
*B	2024-02-06	Fixed broken links
*C	2024-09-20	Added VDE compliance report, see Figure 12
*D	2025-02-05	Updated for mtb-stl v3.x
*E	2025-10-14	Updated to add support for new BSPs
*F	2026-11-06	Updated Table 2, Table 4, Table 5, Table 6 Added APSR test, PRIMASK test, CONTROL test, Stack-free SRAM test (assembly implementation) sections

Trademarks

PSOC™, formerly known as PSoC™, is a trademark of Infineon Technologies. Any references to PSoC™ in this document or others shall be deemed to refer to PSOC™.

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2026-11-06

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2026 Infineon Technologies AG

All Rights Reserved.

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-hcu1773314540090

Important notice

Products which may also include samples and may be comprised of hardware or software or both ("Product(s)") are sold or provided and delivered by Infineon Technologies AG and its affiliates ("Infineon") subject to the terms and conditions of the frame supply contract or other written agreement(s) executed by a customer and Infineon or, in the absence of the foregoing, the applicable Sales Conditions of Infineon. General terms and conditions of a customer or deviations from applicable Sales Conditions of Infineon shall only be binding for Infineon if and to the extent Infineon has given its express written consent.

For the avoidance of doubt, Infineon disclaims all warranties of non-infringement of third-party rights and implied warranties such as warranties of fitness for a specific use/purpose or merchantability. Infineon shall not be responsible for any information with respect to samples, the application or customer's specific use of any Product or for any examples or typical values given in this document.

If this document is marked as 'Preliminary', 'Target', 'Draft' or 'Proposal', Infineon reserves the right to change all information given in this document at any time without notice. Subject to the development and release of a Product for series supply by Infineon, the technical specifications of the Product are set forth in the relevant datasheet provided by Infineon without such marking.

The data contained in this document is exclusively intended for technically qualified and skilled customer representatives. It is the responsibility of the customer to evaluate the suitability of the Product for the intended application and the customer's specific use and to verify all relevant technical data contained in this document in the intended application and the customer's specific use. The customer is responsible for properly designing, programming, and testing the functionality and safety of the intended application, as well as complying with any legal requirements related to its use.

Unless otherwise explicitly approved by Infineon, Products may not be used in any application where a failure of the Products or any consequences of the use thereof can reasonably be expected to result in personal injury.

However, the foregoing shall not prevent the customer from using any Product in such fields of use that Infineon has explicitly designed and sold it for, provided that the overall responsibility for the application lies with the customer. Infineon expressly reserves the right to use its content for commercial text and data mining (TDM) according to applicable laws, e.g. Section 44b of the German Copyright Act (UrhG). If the Product includes security features:

Because no computing device can be absolutely secure, and despite security measures implemented in the Product, Infineon does not guarantee that the Product will be free from intrusion, data theft or loss, or other breaches ("Security Breaches"), and Infineon shall have no liability arising out of any Security Breaches.

If this document includes or references software:

The software is owned by Infineon under the intellectual property laws and treaties of the United States, Germany, and other countries worldwide. All rights reserved. Therefore, you may use the software only as provided in the software license agreement accompanying the software.

If no software license agreement applies, Infineon hereby grants you a personal, non-exclusive, non-transferable license (without the right to sublicense) under its intellectual property rights in the software (a) for software provided in source code form, to modify and reproduce the software solely for use with Infineon hardware products, only internally within your organization, and (b) to distribute the software in binary code form externally to end users, solely for use on Infineon hardware products. Any other use, reproduction, modification, translation, or compilation of the software is prohibited.

For further information on the Product, technology, delivery terms and conditions, and prices, please contact your nearest Infineon office or visit www.infineon.com.