PSOC™ Control C3 Class B safety software library

About this document

Scope and purpose

This application note describes the PSOC™ Control C3 MCU IEC 60730 Class B and IEC 61508 safety integrity level (SIL2) safety software library for use with ModusToolbox™. AN includes the safety library source code and example projects for PSOC™ Control C3 device families with self-check routines to help ensure reliable and safe operation. You can integrate the library routines and examples included in the example projects with your application. This application note also describes the API functions that are available in the library.

Intended audience

The intended audiences for this document are design engineers, technicians, and developers of electronic systems.

This document is intended for anyone who uses the PSOC™ Control C3 software libraries for ModusToolbox™ software.

Table of contents

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 executes 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 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 application note 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 the critical functions. Before executing a critical function, both CPUs are required to share that they have completed their corresponding task. 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.

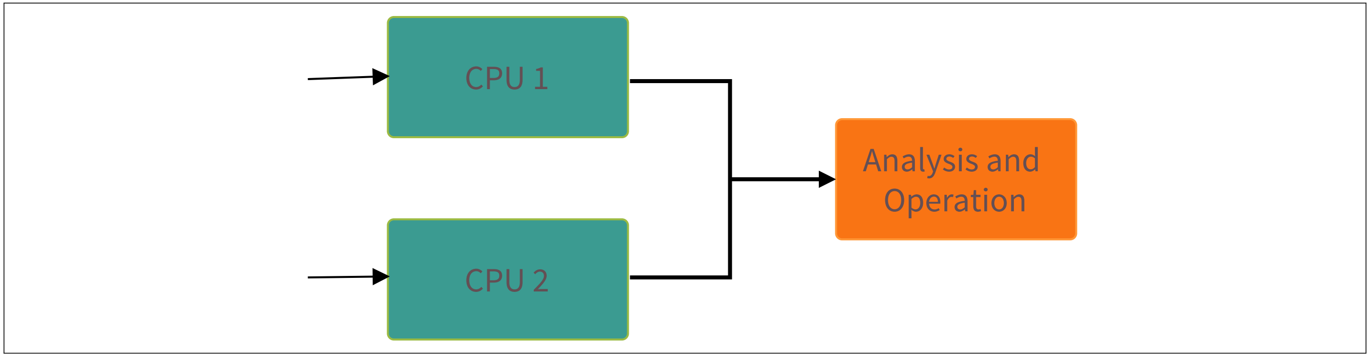


Figure 1 **Dual channel without comparison structure**

The dual-channel structure implementation is costlier because two CPUs (or two MCUs) are required. 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 IEC61508-2 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 few 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 such 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: ≥10-6 to < 10-5 (1 failure in 11 years)
- SIL 2: ≥10-7 to < 10-6 (1 failure in 114 years)
- SIL 3: ≥10-8 to < 10-7 (1 failure in 1,144 years)
- SIL 4: ≥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 IEC 60730 Class B and IEC 61508 requirements

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	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 software library usage

The safety software library described in this application note can be used with PSOC™ Control C3 MCU devices using ModusToolbox™ or the Peripheral Driver Library (PDL). The library includes APIs that are designed to maximize application reliability through fault detection. The mtb-stl library is developed with adherence to Class-B and SIL2 standards. Infineon provides the mtb-stl middleware as well as few code examples in source to customers. The code examples for PSOC™ Control C3 can be found GitHub. The STL and Class B certification of the supported devices component will be a part of enabling the customer to obtain certification for their safety critical system.

This application note describes and implements 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
  - 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

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™ Control C3

Function	Return	Located
cystatus SelfTests_Save_StartUp_ConfigReg(void)	0 - Write in flash is successful 1 - Error detected during flash writing	SelfTest_ConfigRegisters.c SelfTest_ConfigRegisters.h
uint8 SelfTests_StartUp_ConfigReg(void)	0 - No error 1 - Error detected	SelfTest_ConfigRegisters.c SelfTest_ConfigRegisters.h SelfTest_ConfigRegisters.h
uint8 SelfTest_PC (void)	0 - No error 1 - Error detected	SelfTest_CPU.c SelfTest_CPU.h
uint8 SelfTest_CPU_Registers(void)	0 - No error 1 – Error detected	SelfTest_CPU.c SelfTest_CPU.h
uint8_t SelfTest_PROGRAM_FLOW(void)	0 - No error 1 - Error detected	SelfTest_CPU.c SelfTest_CPU.h
void SelfTest_WDT(void)	0 - No error 1 - Error detected	SelfTest_WDT.h SelfTest_WDT.c
uint8 SelfTest_IO()	0 - No error 1 - Error detected (Short to VCC) 2 - Error detected (Short to GND)	SelfTest_IO.c SelfTest_IO.h
uint8_t SelfTest_FPU_Registers(void)	0 - No error 1 - Error detected	SelfTest_FPU_Regs.c SelfTest_FPU_Regs.h
uint8_t SelfTest_DMA_DW(DW_Type * base, uint32_t channel, cy_stc_dma_descriptor_t * descriptor0, cy_stc_dma_descriptor_t * descriptor1,const cy_stc_dma_descriptor_config_t * des0_config, const cy_stc_dma_descriptor_config_t * des1_config,cy_stc_dma_channel_config_t const * channelConfig, en_trig_output_pdma0_tr_t trigLine)	0 - No error 1 - Error detected	SelfTest_DMA_DW.c SelfTest_DMA_DW.h
uint8_t SelfTest_IPC(void)	0 - No error 1 - Error detected	SelfTest_IPC.c SelfTest_IPC.h
uint8 SelfTest_Clock(void)	1 - Error Detected 2 - Test Running 3 - Test Completed 4 - Incorrect Usage	SelfTest_Clock.c SelfTest_Clock.h
uint8 SelfTest_Interrupt(void)	0 - No error 1 - Error detected	SelfTest_Interrupt.c SelfTest_Interrupt.h
uint8_t SelfTest_Counter_Timer(void)	0 - No error 1 - Error detected	SelfTest_Timer_Counter.c SelfTest_Timer_Counter.h
uint8_t SelfTest_PWM(GPIO_PRT_Type *pinbase, uint32_t pinNum)	0 - No error 1 - Error detected	SelfTest_PWM.c SelfTest_PWM.h

Function	**Return**	**Located**
uint8_t SelfTest_PWM_GateKill(TCPWM_Type *base, uint32_t cntNum)	0 - No error 1 - Error detected	SelfTest_PWM_GateKill.c SelfTest_PWM_GateKill.h
uint8 SelfTest_FlashCheckSum(uint32 DoubleWordsToTest)	1 - Error detected 2 - Checksum for one block calculated, but end of Flash was not reached 3 - Pass, Checksum of all flash is calculated and checked	SelfTest_Flash.c SelfTest_Flash.h
uint8_t SelfTests_SRAM_March(void)	1 - Error Detected 2 - Pass, but still testing status 3 - pass and complete status	SelfTest_RAM.c SelfTest_RAM.h
void SelfTests_Init_Stack_Test(uint8_t stack_pattern_blk_size)	None	SelfTest_Stack.c SelfTest_Stack.h
uint8_t SelfTests_Stack_Check(void)	0 - No error 1 - Error detected	SelfTest_Stack.c SelfTest_Stack.h
uint8_t SelfTests_ADC_TrigIn(uint32_t group, uint32_t channel, int16_t expected_res, int16_t accuracy, uint32_t trig_in)	0 - No error 1 - Error detected	SelfTest_Analog.h SelfTest_Analog.c
uint8 SelfTest_UART_SCB(CySCB_Type *base)	1 – Error detected – Data mismatch 2 – Pass test with current values, but not all tests in range from 0x00 to 0xFF have completed 3 – Pass, tested with all values in range from 0x00 to 0xFF 4 – TX Not empty 5 – RX Not empty 6 - Error-UART is not enabled	SelfTest_UART_SCB.c SelfTest_UART_SCB.h
void UartMesMaster_Init(CySCB_Type* uart_base, TCPWM_Type* counter_base, uint32_t cntNum)	None	SelfTest_UART_master_message.h SelfTest_UART_master_message.c
uint8 UartMesMaster_DataProc(uint8_t address, uint8_t *txd, uint8_t tlen, uint8_t * rxd, uint8_t rlen)	0 - No error 1 - Error detected	SelfTest_UART_master_message.h SelfTest_UART_master_message.c

Function	Return	Located
uint8 UartMesMaster_State(void)	0 – The last transaction process finished successfully, the received buffer contains a response. The unit is ready to start a new process 1 – the last transaction process finished with an error and the unit is ready to start a new process 2 – the unit is busy with an active transaction operation.	SelfTest_UART_master_message.h SelfTest_UART_master_message.c
uint8 UartMesMaster_GetDataSize(void)	Received data size in buffer	SelfTest_UART_master_message.h SelfTest_UART_master_message.c
void UartMesSlave_Init(CySCB_Type* uart_base, uint8_t address)	None	SelfTest_UART_slave_message.h SelfTest_UART_slave_message.c
uint8 UartMesSlave_Respond(char * txd, uint8 tlen)	0 - No error 1 - Error detected	SelfTest_UART_slave_message.h SelfTest_UART_slave_message.c
uint8 UartMesSlave_State(void)	0 – the last transaction process is finished 1 – the unit has received a marker and there is received data in the buffer. The master waits for a response. 2 – the unit is busy with sending a response.	SelfTest_UART_slave_message.h SelfTest_UART_slave_message.c
uint8 * UartMesSlave_GetDataPtr(void)	Returns pointer to received data buffer	SelfTest_UART_slave_message.h SelfTest_UART_slave_message.c
uint8 SelfTest_SPI_SCB (CySCB_Type *base)	1 - test failed 2 - still testing 3 - Test completed 4 - TX Not empty 5 - RX Not empty	SelfTest_SPI_SCB.h SelfTest_SPI_SCB.c
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)	3 - test success 1 - test failed 2 - I2C Master Busy	SelfTest_I2C_SCB.h SelfTest_I2C_SCB.c

Function	Return	Located
uint8_t SelfTest_CANFD(CANFD_Type *base, uint32_t chan, const cy_stc_canfd_config_t *config, cy_stc_canfd_context_t *context, stl_canfd_test_mode_t test_mode)	0 - No error 1 - Error detected	SelfTest_CANFD.c SelfTest_CANFD.h
uint8_t SelfTests_DAC_TrigIn(uint32_t adc_channel, uint32_t dac_slice, uint32_t dac_val, int16_t expected_res, int16_t accuracy, uint32_t adc_trig_in, uint32_t dac_trig_in)	0 - No error 1 - Error detected	SelfTest_Analog.h SelfTest_Analog.c
uint8_t SelfTest_Motif_Start(TCPWM_MOTIF_GRP_MOTIF_Type *base , stl_motif_tcpwm_config_t *input_config)	0 - No error 1 - Error detected	SelfTest_Motif.h SelfTest_Motif.c
uint8_t SelfTests_Comparator(LPCOMP_Type const* lpcomp_base, cy_en_lpcomp_channel_t lpcomp_channel, uint8_t expected_res);	0 - No error 1 - Error detected	SelfTest_Analog.h SelfTest_Analog.c
uint8_t SelfTest_Cordic(void)	0 - No error 1 - Error detected	SelfTest_Cordic.h SelfTest_Cordic.c
uint8_t SelfTest_ECC_Flash(uint32_t addr,cy_en_ecc_error_mode_t eccErrorMode)	0 - No error 1 - Error detected	SelfTest_ECC.h SelfTest_ECC.c
uint8_t SelfTest_ECC_Ram(uint32_t addr,cy_en_ecc_error_mode_t eccErrorMode)	0 - No error 1 - Error detected	SelfTest_ECC.h SelfTest_ECC.c

6.1 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 High-speed I/O matrix (HSIOM) registers.

These startup configuration registers are typically static and are in the `cy_device.h` file after the design 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 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**: The following functions are examples and can be applied only to the example project. If you make changes in the configuration, other configuration registers may be generated in the `cy_device.h`. You must change the list of required registers based on the safety needs of your design.*

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
```

This function copies all listed startup configuration registers into the last row(s) of flash as required by the number of registers to save.

***Note**:* *This function should be called once after the initial PSOC™ Control C3 device 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.*

***Note**:* *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 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-32 calculation and verification
- Register comparison with duplicated copy

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

```
#define STARTUP_CFG_REGS_MODE CFG_REGS_CRC_MODE / CFG_REGS_TO_FLASH_MODE
#define CFG_REGS_TO_FLASH_MODE (1u)
```

This mode 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 `stl_ConfigRegisters.h`.

```
#define CFG_REGS_CRC_MODE (0u)
```

In this mode, the function calculates a CRC-32 of registers and stores the CRC in flash. Later, function calls recalculate the CRC-32 and compare it with the saved value. It returns a fail (1) if the values are different.

By default, the CRC mode values are stored at the 9th, 7th, and 8th byte locations from the end of flash. While using alongside the flash test, which uses last 8 bytes of flash, it is recommended to move the CRC mode 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, and CRC_STARTUP_HI in the `stl_ConfigRegisters.h` file.

The following Table 2 provides a list of default registers included in library test. The Table 2 should be used as reference and the list should be updated as per the MCU for which STL is developed.

Table 2 Library test for PSOC™ Control C3

Clock registers	HSIOM registers	IO pin registers
SRSS_WDT_CTL	HSIOM_PRT_PORT_SEL0(HSIOM_PRT0)	GPIO_PRT_CFG(GPIO_PRT0)
SRSS_CLK_PATH_SELECT	HSIOM_PRT_PORT_SEL1(HSIOM_PRT0)	GPIO_PRT_CFG(GPIO_PRT1)
SRSS_CLK_ROOT_SELECT	HSIOM_PRT_PORT_SEL0(HSIOM_PRT1)	GPIO_PRT_CFG(GPIO_PRT2)
SRSS_CLK_SELECT	HSIOM_PRT_PORT_SEL1(HSIOM_PRT1)	GPIO_PRT_CFG(GPIO_PRT3)
SRSS_CLK_TIMER_CTL	HSIOM_PRT_PORT_SEL0(HSIOM_PRT2)	GPIO_PRT_CFG(GPIO_PRT4)
SRSS_CLK_ILO_CONFIG	HSIOM_PRT_PORT_SEL1(HSIOM_PRT2)	GPIO_PRT_CFG(GPIO_PRT5)
SRSS_CLK_OUTPUT_SLOW	HSIOM_PRT_PORT_SEL0(HSIOM_PRT3)	GPIO_PRT_CFG(GPIO_PRT6)
SRSS_CLK_OUTPUT_FAST	HSIOM_PRT_PORT_SEL1(HSIOM_PRT3)	GPIO_PRT_CFG(GPIO_PRT7)
SRSS_CLK_ECO_CONFIG	HSIOM_PRT_PORT_SEL0(HSIOM_PRT4)	GPIO_PRT_CFG(GPIO_PRT8)
SRSS_CLK_PILO_CONFIG	HSIOM_PRT_PORT_SEL1(HSIOM_PRT4)	GPIO_PRT_CFG(GPIO_PRT9)
SRSS_CLK_FLL_CONFIG	HSIOM_PRT_PORT_SEL0(HSIOM_PRT5)	GPIO_PRT_CFG(GPIO_PRT10)
SRSS_CLK_FLL_CONFIG2	HSIOM_PRT_PORT_SEL1(HSIOM_PRT5)	GPIO_PRT_CFG(GPIO_PRT11)
SRSS_CLK_FLL_CONFIG3	HSIOM_PRT_PORT_SEL0(HSIOM_PRT6)	GPIO_PRT_CFG(GPIO_PRT12)
SRSS_CLK_PLL_CONFIG	HSIOM_PRT_PORT_SEL1(HSIOM_PRT6)	GPIO_PRT_CFG(GPIO_PRT13)
	HSIOM_PRT_PORT_SEL0(HSIOM_PRT7)	

(table continues...)

Table 2 (continued) Library test for PSOC™ Control C3

Clock registers	**HSIOM registers**	**IO pin registers**
Analog routing regs	HSIOM_PRT_PORT_SEL1(HSIOM_PRT7)	
PASS_AREF_AREF_CTRL	HSIOM_PRT_PORT_SEL0(HSIOM_PRT8)	
	HSIOM_PRT_PORT_SEL1(HSIOM_PRT8)	
	HSIOM_PRT_PORT_SEL0(HSIOM_PRT9)	
	HSIOM_PRT_PORT_SEL1(HSIOM_PRT9)	
	HSIOM_PRT_PORT_SEL0(HSIOM_PRT10)	
	HSIOM_PRT_PORT_SEL1(HSIOM_PRT10)	
	HSIOM_PRT_PORT_SEL0(HSIOM_PRT11)	
	HSIOM_PRT_PORT_SEL1(HSIOM_PRT11)	
	HSIOM_PRT_PORT_SEL0(HSIOM_PRT12)	
	HSIOM_PRT_PORT_SEL1(HSIOM_PRT12)	
	HSIOM_PRT_PORT_SEL0(HSIOM_PRT13)	
	HSIOM_PRT_PORT_SEL1(HSIOM_PRT13)	

6.2 Program counter test

The PSOC™ Control C3 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 the GCC compiler, these are located in the custom linker file provided as follows:

   ```
   NV_CONFIG2 0x10155554 :
   {
   . = 0x00;
   KEEP (*(PC5555))
   } >flash =0
   NV_CONFIG3 0x100AAAA8 :
   {
   . = 0x00;
   KEEP (*(PCAAAA))
   } >flash =0
   ```

   This GCC linker script example demonstrates the maximum checkerboard pattern while byte-aligned to 4 for a flash size of 0x40000. In the example project, it is already added to the GCC *linker.ld* file.
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 - No error
1 - Error detected
Located in:
SelfTest_CPU.c
SelfTest_CPU.h
```

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

6.3 CPU registers test

PSOC™ Control C3 with the Arm ® Cortex®-M33 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 - No error
1 – Error detected
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™ Control device should not continue to function because its behavior can be unpredictable and therefore potentially unsafe.

6.4 Program flow test

This test demonstrates a specific method is 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.

Function

```
uint8_t SelfTest_PROGRAM_FLOW(void)
Returns:
0 - No error
1 - Error detected
Located in:
SelfTest_CPU.c
SelfTest_CPU.h
```

The function SelfTest_PROGRAM_FLOW is called to do the Program flow test.

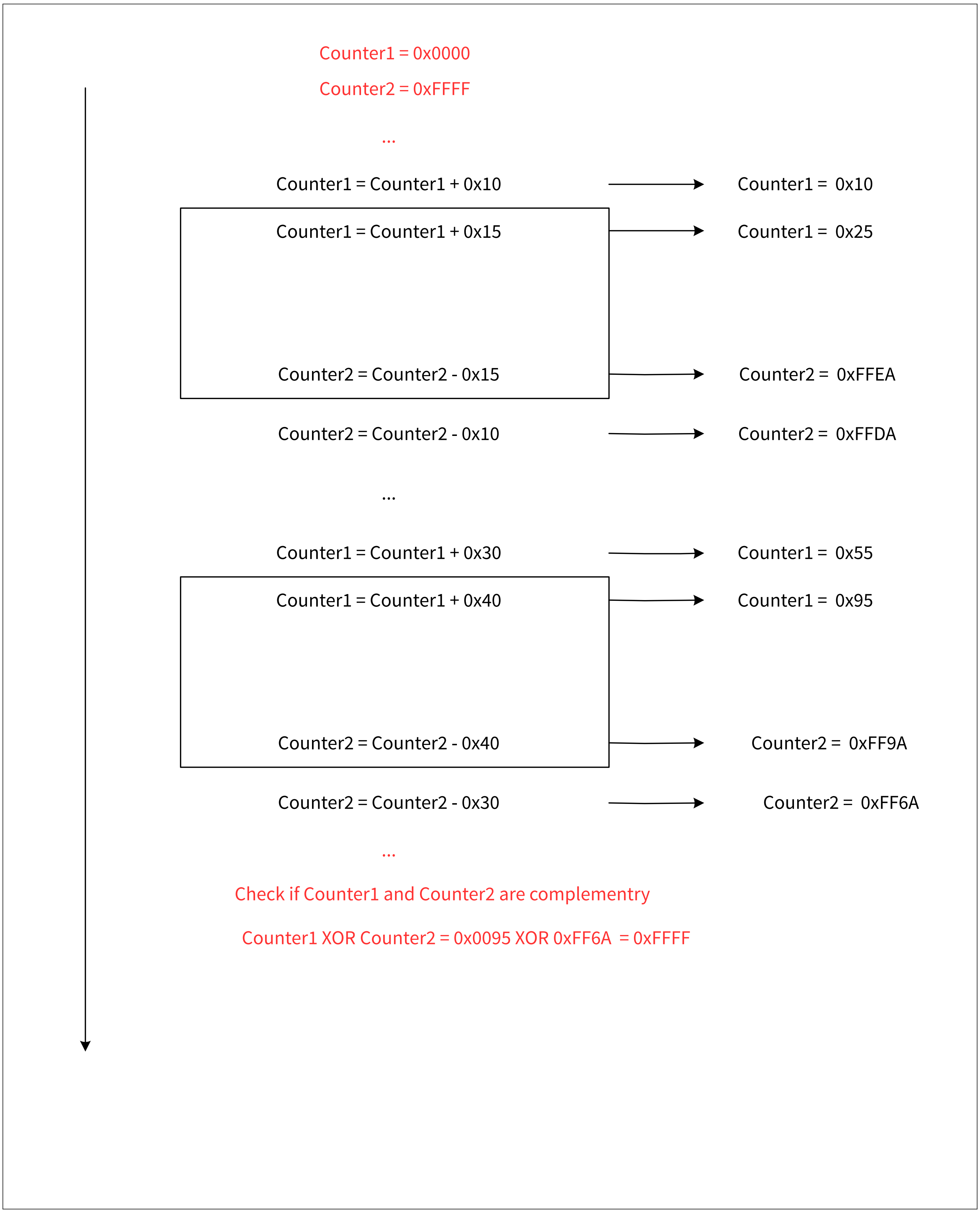

Counter1 = 0x0000
Counter2 = 0xFFFF
...
Counter1 = Counter1 + 0x10
Counter1 = 0x10
Counter1 = Counter1 + 0x15
Counter1 = 0x25
Counter2 = Counter2 - 0x15
Counter2 = 0xFFEA
Counter2 = Counter2 - 0x10
Counter2 = 0xFFDA
...
Counter1 = Counter1 + 0x30
Counter1 = 0x55
Counter1 = Counter1 + 0x40
Counter1 = 0x95
Counter2 = Counter2 - 0x40
Counter2 = 0xFF9A
Counter2 = Counter2 - 0x30
Counter2 = 0xFF6A
...
Check if Counter1 and Counter2 are complementry
Counter1 XOR Counter2 = 0x0095 XOR 0xFF6A = 0xFFFF

Figure 2 **Program flow Test**

6.5 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 - No error
1 - Error detected
Located in:
SelfTest_WDT.h
SelfTest_WDT.c
```

This function provides watchdog reset or stops software execution if watchdog fails.

***Note**:* *Either of WDT or WWDT test should be performed for PSOC™ Control C3 devices.*

Figure 3 shows the test flow chart.

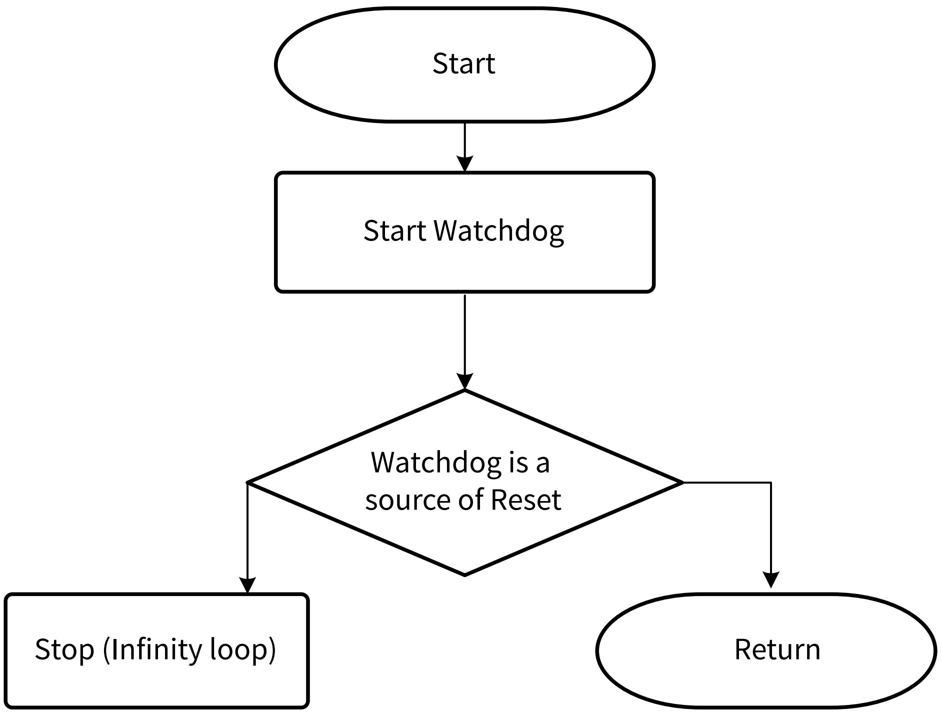


Figure 3 Implementation of WDT test

6.6 ECC test

This function performs the ECC hardware self test to report fault for double bit error.

Function

```
Function:
uint8_t SelfTest_ECC_Flash(uint32_t addr, cy_en_ecc_error_mode_t eccErrorMode)

Parameters:
addr - Flash address
eccErrorMode - Error injection mode

Returns:
0 - No error
1 - Error detected
Function:
uint8_t SelfTest_ECC_Ram(uint32_t addr, cy_en_ecc_error_mode_t eccErrorMode)

Parameters:
addr - Ram address
eccErrorMode - Error injection mode

Returns:
0 - No error
1 - Error detected

Located in:
SelfTest_Ecc.c
SelfTest_Ecc.h
```

6.7 GPIO test

***Note**: Some pins cannot be tied to pull-up (motor control, for example). The test must allow a mask of testable/non-testable pins.*

Any GPIO pin can be 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 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 SelfTest_IO()
Returns:
0 - No error
1 - Error detected (Short to VCC)
2 - Error detected (Short to GND)
Located in:
SelfTest_IO.c
SelfTest_IO.h
```

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

For example:

```
static const uint8 PinToTest[] =
{0xDFu, /* PORT0 mask */
0x3Fu, /* PORT1 mask */
0xFFu, /* PORT2 mask */
0xF3u, /* PORT3 mask */
0x00u, /* 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'.

6.8 FPU register test

The Arm® Cortex®-M33 Floating-Point Unit registers can be represented as 32 single-precision (32-bit) registers or as 16 double-precision (64-bit) registers.

The FPU registers test detects stuck-at faults in the FPU by using the checkerboard test. This test ensures that the bits in the registers are not stuck at value '0' or '1'.

Function

```
uint8_t SelfTest_FPU_Registers(void)
Returns:
0 - No error
1 - Error detected
Located in:
SelfTest_FPU_Regs.c
SelfTest_FPU_Regs.h
```

The function SelfTest_FPU_Registers is called to do the FPU test. If an error is detected, the device should not continue to function because its behavior can be unpredictable and therefore potentially unsafe.

It is a **destructive** test that performs the following major tasks:

1. The registers are tested by performing a write/read/compare test sequence using a checkerboard pattern (0x5555 5555, then 0xAAAA AAAA). These binary sequences are valid floating point values.
2. The test returns an error code if the returned values do not match.

6.9 DMA/DW test

The DataWire and DMA blocks are tested using the following procedure. Each DW or DMA channel is tested using the same procedure.

Function

```
uint8_t SelfTest_DMA_DW(DW_Type * base, uint32_t channel, cy_stc_dma_descriptor_t *
descriptor0, cy_stc_dma_descriptor_t * descriptor1,const cy_stc_dma_descriptor_config_t *
des0_config,const cy_stc_dma_descriptor_config_t * des1_config,cy_stc_dma_channel_config_t
const * channelConfig, en_trig_output_pdma0_tr_t trigLine)
Parameters:
base - The pointer to the hardware DMA block
channel - A channel number
descriptor0 - This is the descriptor to be associated with the channel (transfer 0's to
destination).
descriptor1 - This is the descriptor to be associated with the channel (transfer pattern to
destination).
des0_config - This is a configuration structure that has all initialization information for the
descriptor.
des1_config - This is a configuration structure that has all initialization information for the
descriptor.
channelConfig - The structure that has the initialization information for the channel.
trigLine - The input of the trigger mux.
Returns:
0 - No error
1 - Error detected
Located in:
SelfTest_DMA_DW.c
SelfTest_DMA_DW.h
```

The DMA/DW test performs the following steps:

1. A destination block of size 64 bytes is set to 0 with DW 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 increment of 3 and a length of 22 (64+(3-1))/3.
4. The destination block is verified to contain the correct pattern (shown below with lowest address first):ff0000ff0000ff0000ff0000ff0000ff0000ff0000ff0000…

6.10 IPC test

Each devices has few free IPC channels and IPC interrupts that can be mapped to each channel.

Function

```
uint8_t SelfTest_IPC(void)
Returns:
0 - No error
1 - Error detected
Located in:
SelfTest_IPC.c
SelfTest_IPC.h
```

This function does:

1. Write the message to the channel.
2. Acquire an IPC channel and specify which IPC interrupts must be notified during the lock event.
3. In the ISR of that IPC interrupt read the data.
4. Release the IPC channel from the locked state using same IPC during the release event.
5. Similarly, for the same channel, use all the free interrupt.
6. Similarly, take all IPC channels with all IPC interrupts.

6.11 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 to ± 10 percent. If accuracy greater than 10 percent is required, the ILO may be trimmed to be more accurate using a precision system level signal or production test. If ILO trimming is required, it is trimmed using the CLK_ILO_TRIM register. If the WCO is available, it should be used for this test, since it is much more accurate. If the other oscillators (ECO, EXT_CLK, IMO) are safety critical, customers can use this test as a guideline.

Function

```
uint8 SelfTest_Clock(void)
Returns:
1 - Error Detected
2 - Test Running
3 - Test Completed
4 - Incorrect Usage
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 32.768-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 software delay, 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 flow chart.

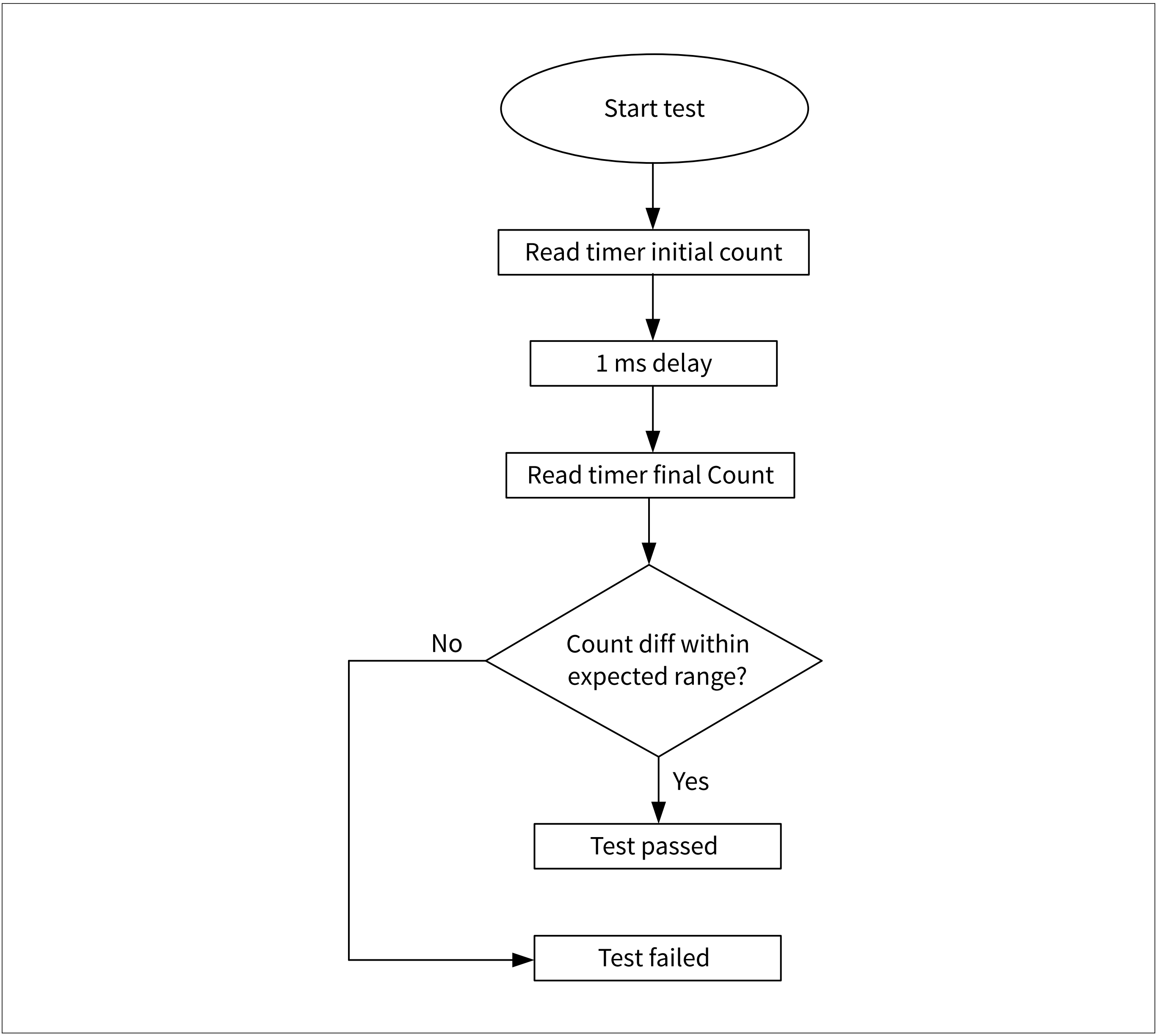


Figure 4 **Clock self-test flow chart**

6.12 Interrupt handling and execution test

For all Arm® processors 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 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(void)
Returns:
0 - No error
1 - Error detected
Located in:
SelfTest_Interrupt.c
SelfTest_Interrupt.h
```

Note: *The component's name should be that shown in Figure 3. Global interrupts must be enabled for this test, but all interrupts except isr_1 must be disabled.*

Timer_1 is configured to generate 13 interrupts per 1 ms. isr_1 counts the number of interrupts that occurred. If the counted value in isr_1 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 5 shows the interrupt self-test flow chart.

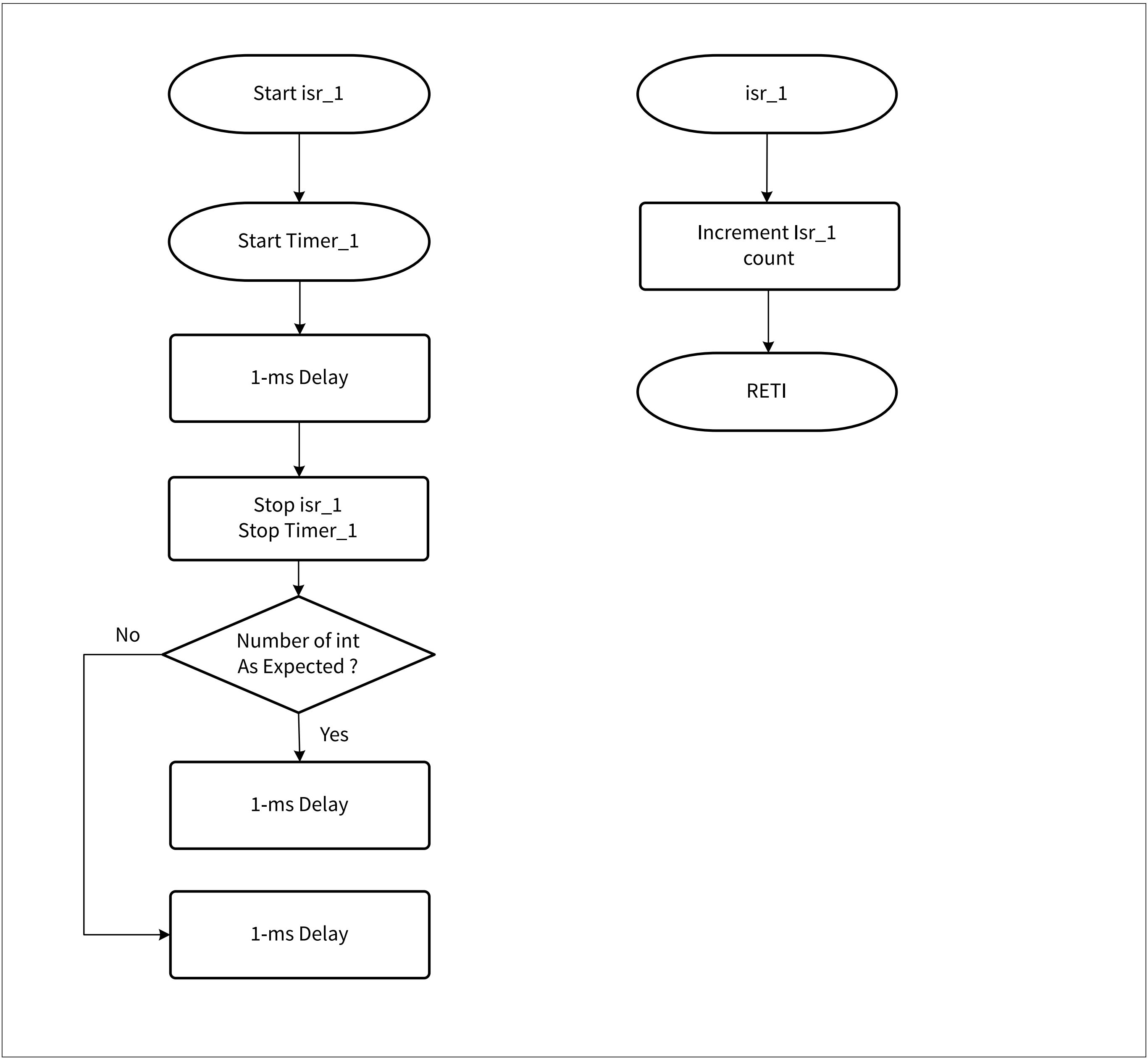


Figure 5 **Interrupt self-test flow chart**

6.13 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™ Control C3 MCUs. It verifies the proper operation and accuracy of these peripherals adhering to the IEC 60730 standards.

6.13.1 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 - No error
1 - Error detected
Located in:
SelfTest_Timer_Counter.c
SelfTest_Timer_Counter.h
```

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

6.13.2 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 15/8 and 17/8. A range is used because the CPU polling loop is asynchronous to the PWM timing

Function

```
uint8_t SelfTest_PWM(GPIO_PRT_Type *pinbase, uint32_t pinNum)
Parameters:
pinbase - The pointer to a GPIO port instance to which the PWM pin is connected to.
pinNum - The GPIO pin number to which the PWM pin is connected to.
Returns:
0 - No error
1 - Error detected
Located in:
SelfTest_PWM.c
SelfTest_PWM.h
```

The test must not drive the PWM output pin.

6.13.3 PWM GateKill test

The "GateKill" function is used in motor controllers and multi-level power converters. When an over-voltage or current state is detected, the GateKill 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)
Parameters:
base - The pointer to a TCPWM instance
cntNum - The Counter instance number in the selected TCPWM
Returns:
0 - No error
1 - Error detected
Located in:
SelfTest_PWM_GateKill.c
SelfTest_PWM_GateKill.h
```

The PWM GateKill is tested using the following procedure:

1. Configure the Kill Mode as Stop on Kill.
2. The low power comparator output or the Voltage range violation of SAR ADC is routed to Kill signal of TCPWM indicating overvoltage/overcurrent condition if voltage on +ve terminal is > -ve.
3. If overvoltage or overcurrent condition it will Kill PWM output.
4. The TCPWM base and CntNum is passed to check whether the counter is stopped or not.
5. If counter is not incrementing/decrementing the PWM output is inactive.

6.14 Flash (invariable memory) test

PSOC™ Control C3 devices include an on chip flash memory is of 256 KB size, there are two 128 KB sectors. Each sector contains 256 rows and each row is of 512 bytes. Word is 128 bits + 9 bit ECC flash memory is organized in row where each row contains 512 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.

6.14.1 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 section. flash_checksum for the GCC 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 Table 2). 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.*

6.14.2 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.

6.14.3 Programming method

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

Function

```
uint8 SelfTest_FlashCheckSum(uint32 DoubleWordsToTest)
Parameters:
DoubleWordsToTest - number of 32-bit Double Words of flash to be calculated per each function
call.
Returns:
1 - Error detected
2 - Checksum for one block calculated, but end of Flash was not reached
3 - Pass, Checksum of all flash is calculated and checked
Located in:
SelfTest_Flash.c
SelfTest_Flash.h
```

The function SelfTest_FlashCheckSum() 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 will automatically increment 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 run time. 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.*

6.15 SRAM (variable memory) test

Note: *PSOC™ Control C3 devices include an on-chip SRAM of up to 64 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™ Control C3 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.

6.15.1 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.

6.15.2 March C algorithm

The March C- Memory Built-In Self Test (MBIST) algorithm covers the majority of SRAM faults, including address decoder faults, Stuck at Fault, State transition fault and various coupling Faults.

The algorithm is as follows:

1. This test first duplicates data in the area to be tested as this test is destructive.
2. Write 0 to the block in ascending order.
3. Read 0 and write 1 in ascending order.
4. Read 1 and write 0 in ascending order.
5. Read 0.
6. Read 0 and write 1 in descending order.
7. Read 1 and write 0 in descending order.
8. Read 0.
9. Copy back the original data Function

Function

```
uint8_t SelfTests_SRAM_March(void)
Returns:
1 - Error Detected
2 - Pass, but still testing status
3 - pass and complete status
Located in:
SelfTest_RAM.c
SelfTest_RAM.h
```

6.15.3 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 that the test is called can be modified. The test takes two seconds 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 one kb stack address space. The block size and frequency that the test is called can be modified.

6.16 Stack overflow test

The stack is a section of 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™ Control C3, 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

```
void SelfTests_Init_Stack_Test(uint8_t stack_pattern_blk_size)
Parameter:
stack_pattern_blk_size - No of bytes to be used to fill the pattern. Must be 2^n where n=1 to
n=8.
Returns:
NONE
Located in:
SelfTest_Stack.c
SelfTest_Stack.h
```

This function is called once to fill the reserved memory block with a predefined pattern.

Function

```
uint8_t SelfTests_Stack_Check(void)
Returns:
0 - No error
1 - Error detected
Located in:
SelfTest_Stack.c
SelfTest_Stack.h
```

This function is called periodically at run time to test for stack overflow. The block size should be an even value and can be modified using the macro located in *SelfTest_Stack.h*:

```
#define STACK_TEST_BLOCK_SIZE 0x08u
```

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

```
#define STACK_TEST_PATTERN 0x55AAu
```

6.17 Analog tests

Analog tests comprise the ADC, DAC, and comparator test. These tests rely on one or two fixed reference voltages. These voltage references can be generated with the help of external hardware. User has to configure the voltage reference and route the voltages to the hardware under test.

6.17.1 Generating the reference voltages

Voltage reference can be provided to the device using external hardware. A possible method could be with the use of a three-resistor voltage divider between VDDA and GND. This will generate two reference voltages: 2/3*VDDA and 1/3*VDDA. These reference voltages can be attached to any GPIO pin and be connected to the AMUXBUS through the GPIO.

6.17.2 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/error control technique with which the inputs/outputs that are designed to be within specified tolerances are compared.

The purpose of the test is to check the SAR ADC analog functions. Each ADC enabled by the test is connected to an external reference voltage. If the measured value is not in the expected range, an error is returned.

Function

```
uint8_t SelfTests_ADC_TrigIn(uint32_t group, uint32_t channel, int16_t expected_res, int16_t
accuracy, uint32_t trig_in)
Parameters:
group - SAR ADC group
channel - Channel to be scanned
expected_res - If count_to_mV = 1 -> pass expected voltage in mV else pass expected counts
accuracy - threshold above and below the expected value that is acceptable.
vbg_channel - Used to convert the count to mV.
trig_in – Trigger signal used for the ADC channel.
Returns:
0 - No error
1 - Error detected
Located in:
SelfTest_Analog.h
SelfTest_Analog.c
```

The ADC accuracy is defined in stl_Analog.h. A 12-bit ADC is used for testing:

```
#define ADC_TEST_ACC 12 // +/- ADC result value
```

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

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

6.17.3 Comparator test

The comparator functional test is performed using two different reference voltages simultaneously. The references can be generated internally or externally, as shown in the 'Generating the Reference Voltages' section. These reference voltages are routed to the inputs of the analog comparator, and then the inputs are switched, forcing the output value of the comparator to change.

Function

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

  Returns:
  0: No error
  1: Error detected

  Parameters
  lpcomp_base: The low-power comparator's registers structure pointer
  lpcomp_channel: The number of the low-power comparator channel
      expected_res: Expected output of the comparator. It has a non-zero value when
      the positive input voltage is expected to be 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. If the test is successful, the function returns '0'; otherwise, it returns '1'.

To test the comparator using two voltage references, follow these steps:

1. Configure reference voltages 1 and 2
2. Configure the comparator for a self-test
3. Route reference voltages 1 and 2 to the comparator's positive and negative terminals, respectively
4. Run the SelfTest_Comparator() function
5. Switch the two reference voltages entering the comparator
6. Run the SelfTest_Comparator() function
7. Restore the comparator and the internal routing

6.17.4 DAC

In this test, the output of the DAC is routed to the input of the ADC. The DAC test is considered successful if the input to the DAC and the output from the ADC (to which the DAC output is routed) are the same.

Function

```
uint8_t SelfTests_DAC_TrigIn(uint32_t adc_channel, uint32_t dac_slice, uint32_t dac_val,
int16_t expected_res, int16_t accuracy, uint32_t adc_trig_in, uint32_t dac_trig_in)
  Returns:
  0: No error
  1: Error detected

  Parameters
  adc_channel: Pointer to the ADC channel
  dac_slice: Pointer to DAC slice
  dac_val: Value to be loaded in DAC register
  expected_res: Expected result in ADC
  accuracy: Error tolerance
  adc_trig_in: Trigger signal used for the ADC channel
  dac_trig_in: Trigger signal used for the DAC channel

  Located in:
  SelfTest_Analog.h
  SelfTest_Analog.c
```

6.18 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 the 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)
Parameters:
base – Pointer to SCB hardware to configure
Returns:
1 – Error detected – Data mismatch
2 – Pass test with current values, but not all tests in range from 0x00 to 0xFF have completed
3 – Pass, tested with all values in range from 0x00 to 0xFF
4 – ERROR_TX_NOT_EMPTY
5 – ERROR_RX_NOT_EMPTY
6 - Error-UART is not enabled.
Located in:
SelfTest_UART_SCB.c
SelfTest_UART_SCB.h
```

The input and output terminals switch between the corresponding pins and loop to each other to provide the internal loopback test by 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 the user to configure Smart I/O before the test and restore Smart I/O after the test.*

6.19 Communications UART data transfer protocol example

For additional system safety when transferring data between system components, you can 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. Figure 6 shows the packet format.

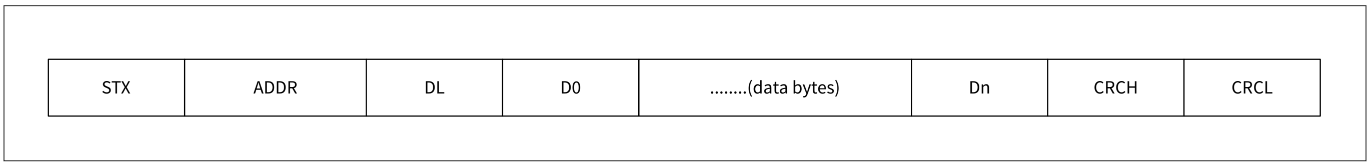


Figure 6 UART packet structure

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 description

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).

(table continues...)

Table 3 (continued) Packet field description

Name	Length	Value	Description
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).

6.19.1 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 the 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.

6.19.2 PSOC™ Control 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

Figure 7 represents the protocol implementation.

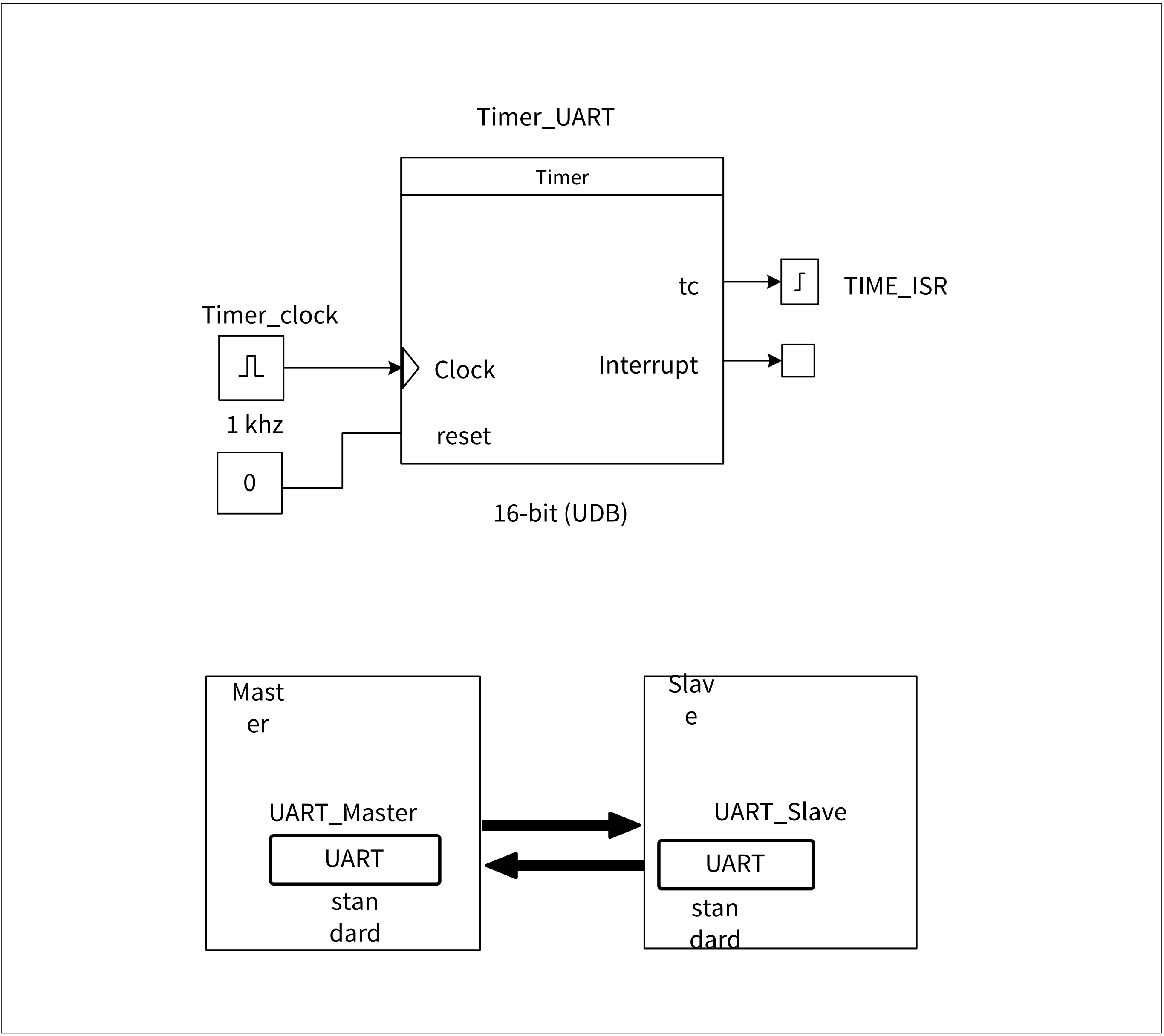


Figure 7 Protocol implementation

This software unit 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)
Parameters:
uart_base - The pointer to the master UART SCB instance.
counter_base - The pointer to the master TCPWM instance.
cntNum - The Counter instance number in the selected TCPWM.
Returns:
None
Located in:
SelfTest_UART_master_message.h
SelfTest_UART_master_message.c
```

This function initializes the UART protocol unit.

Function 2

```
uint8 UartMesMaster_DataProc(uint8_t address, uint8_t *txd, uint8_t tlen, uint8_t * rxd,
uint8_t rlen)
Parameters:
address - slave address for data transfer
txd - pointer to transmitted data
tlen - size of transmitted data in bytes
rxd - pointer to the buffer where the received data will be stored
rlen - size of received data buffer
Returns:
0 - No error
1 - Error detected
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 UartMesMaster_State(void)
Returns:
0 (UM_COMPLETE) – the last transaction process finished successfully, the received buffer
contains a response. The unit is ready to start a new process
1 (UM_ERROR) – the last transaction process finished with an error and the unit is ready to
start a new process
2 (UM_BUSY) – the 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 protocol unit.

Function 4

```
uint8 UartMesMaster_GetDataSize(void)
Returns:
Received data size in buffer
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.

Four functions for working with the protocol unit for the slave

Function 1

```
void UartMesSlave_Init(CySCB_Type* uart_base, uint8_t address)
Parameter:
uart_base - The pointer to the slave UART SCB instance.
address: Slave address
Returns:
None
Located in:
SelfTest_UART_slave_message.h
SelfTest_UART_slave_message.c
```

Function 2

```
uint8 UartMesSlave_Respond(char * txd, uint8 tlen)
Parameters:
txd: Pointer to the transmitted data (request data)
tlen: Length of the request in bytes
Returns:
0 - No error
1 - Error detected
Located in:
SelfTest_UART_slave_message.h
SelfTest_UART_slave_message.c
```

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

Function 3

```
uint8 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
```

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

Function 4

```
uint8 * UartMesSlave_GetDataPtr(void)
Returns:
Returns pointer to received data buffer
Located in:
SelfTest_UART_slave_message.h
SelfTest_UART_slave_message.c
```

This function obtains a pointer to the received data.

6.20 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)
Parameters:
base – Base address of the UART to test.
Returns:
1 - test failed
2 - still testing
3 - Test completed
4 - TX Not empty
5 - RX Not empty
Located in:
SelfTest_SPI_SCB.h
SelfTest_SPI_SCB.c
```

The SPI input and output terminals switch between the corresponding pins and loop to each other to provide the internal loopback test using a 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 the user to configure Smart I/O before the test and restore Smart I/O after the test.*

6.21 Communications I2C test

This test requires connection between I2C Master and Slave externally and uses two SCB blocks.

This project uses a I2C master as the component under test as shown in Figure 8. With no modifications, the project can also be used for testing a I2C slave as both are tested simultaneously (Figure 9).

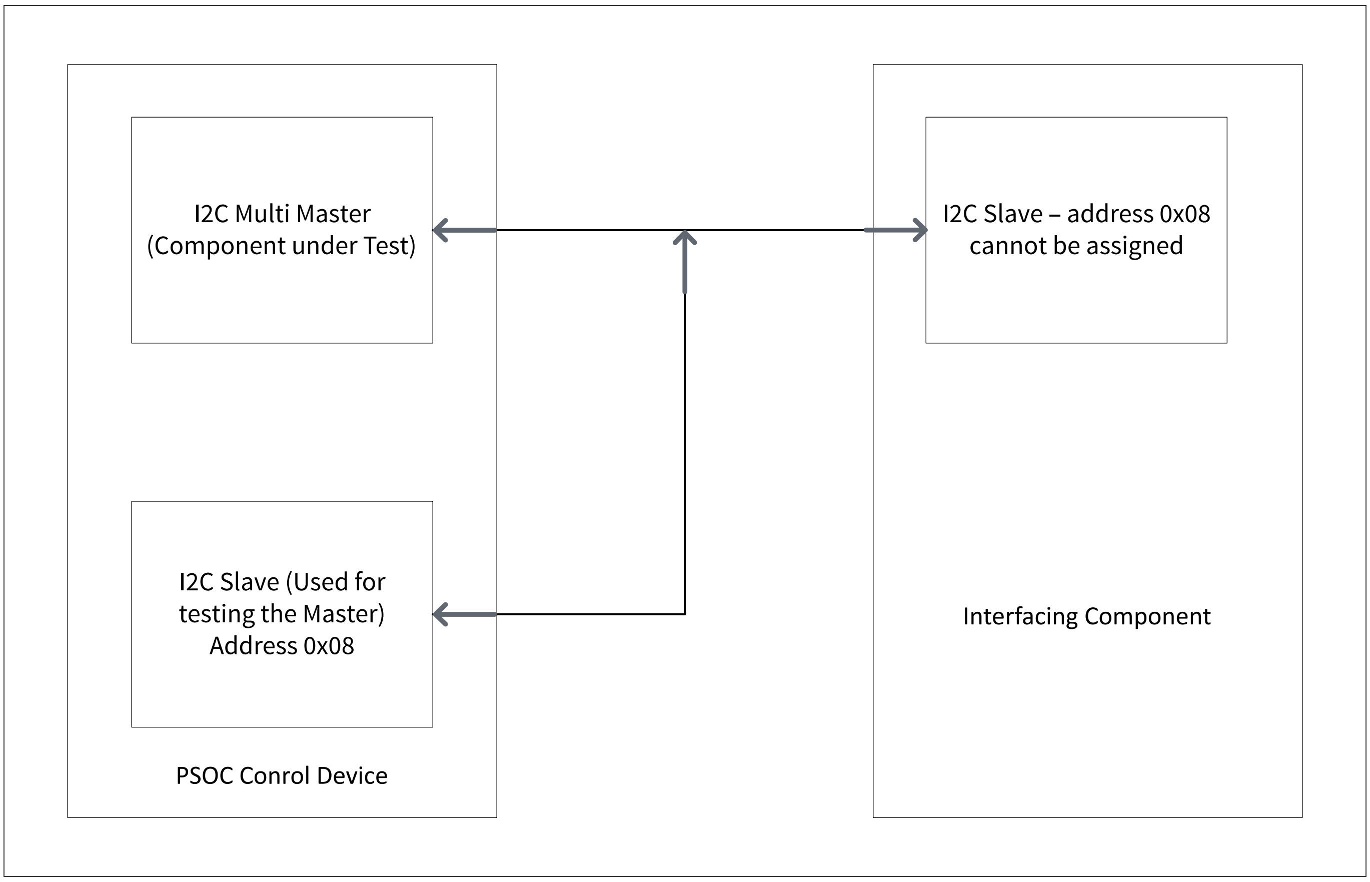

I2C Multi Master (Component under Test)
I2C Slave – address 0x08 cannot be assigned
I2C Slave (Used for testing the Master) Address 0x08
PSOC Conrol Device
Interfacing Component

Figure 8 **I2C master under test**

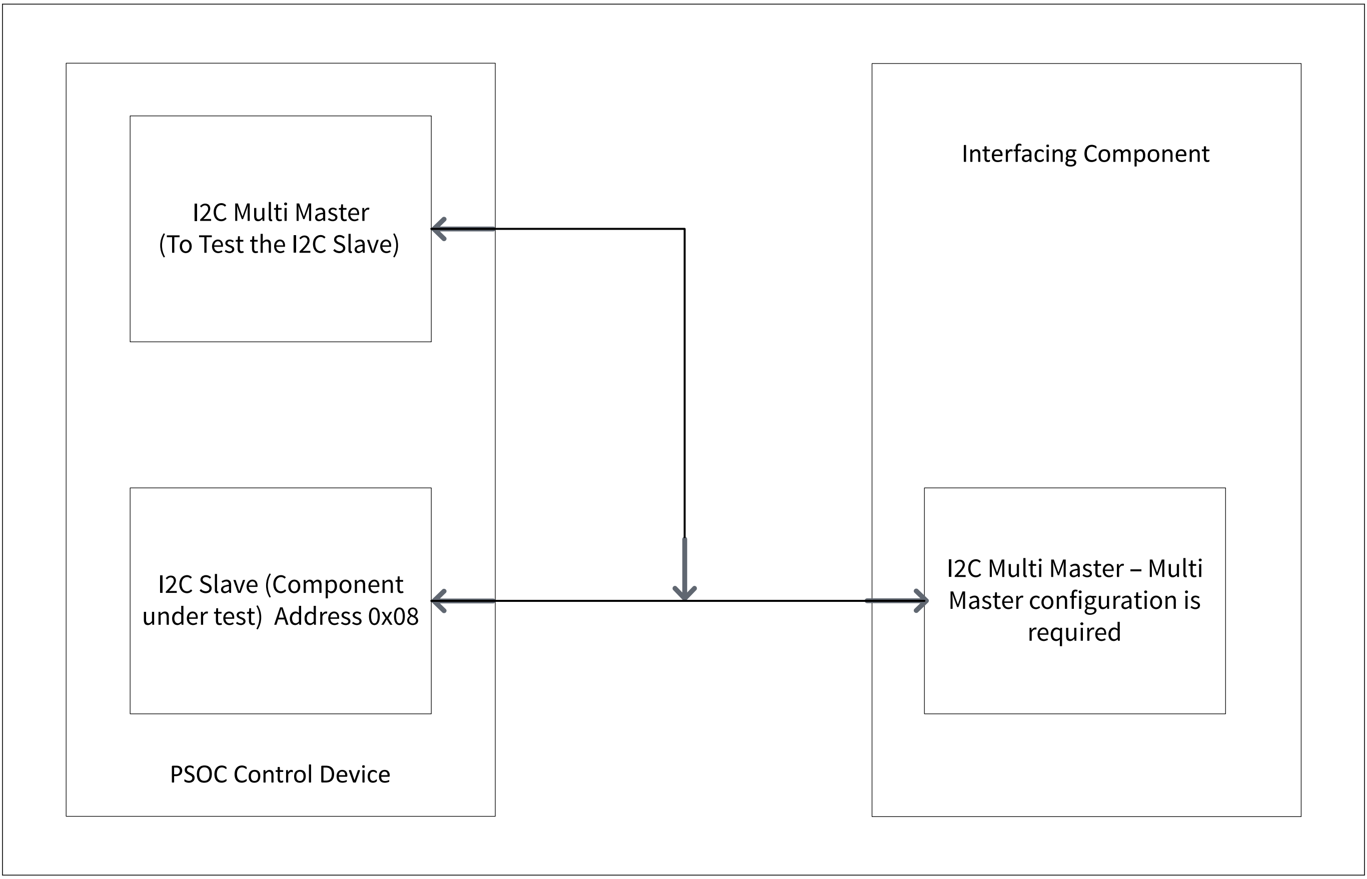

I2C Multi Master
(To Test the I2C Slave)
I2C Slave (Component under test) Address 0x08
PSOC Control Device
Interfacing Component
I2C Multi Master – Multi Master configuration is required

Figure 9 **I2C slave 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 slave and compares it with the complement of data sent in 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)
Parameters:
master_base - The pointer to the master I2C SCB instance.
master_context - The pointer to the master I2C SCB context.
slave_base - The pointer to the slave I2C SCB instance.
slave_context - The pointer to the slave I2C SCB context.
slave_read_buf - The pointer to slave read buffer.
slave_write_buf - The pointer to slave write buffer.
transmitData - data to be transmitted over I2C
Returns:
3 - test success
1 - test failed
2 - I2C Master Busy
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

6.22 Communications CANFD test

This test demonstrates CAN FD loopback mode. The test sequence for verifying CAN FD functionality with loopback mode involves enabling loopback and configuring the FIFOs, setting CAN FD to accept message IDs in the range 0x50 to 0x55, and transmitting messages to validate filtering. A message with ID 0x60 is sent to ensure the RX FIFO remains empty, confirming it is outside the accepted range. Then, a message with ID 0x50 is transmitted and verified to be correctly received in the RX FIFO. If the CAN bus fails, the device should enter a fail-safe state to ensure safety and reliability

The CAN-FD block is tested using the loopback capability. The test steps are:

- Enable loopback externally or internally based on the parameter passed
- Configure CAN-FD to accept message IDs in the range 0x50 to 0x55 in any of the RX FIFO (1 or 2)
- Transmit a message with message ID 0x60
- Verify that the RX FIFO is empty
- Transmit a message with message ID 0x50
- Verify that 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, stl_canfd_test_mode_t test_mode)
Parameters:
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 context structure allocated by the user. The structure is used
during the CAN FD operation for internal configuration and data retention. User must not modify
anything in this structure
test_mode - internal : will not drive the pin , external : will drive the external pins along
with loopback
Returns:
0 - No error
1 - Error detected
Located in:
SelfTest_CANFD.c
SelfTest_CANFD.h
```

If the CAN bus fails to operate, the device should fail-safe. CAN-FD to be tested from IP side with loop back mode.

6.23 Cordic test

The CORDIC block accelerates the calculation of trigonometric functions. These functions include sine, cosine, arctan, sinh, cosh, arctanh, and sqrt. The sine and cosine trigonometric functions are performed to validate the proper working of the CORDIC peripheral.

Function

```
uint8_t SelfTest_Cordic(void)
  Returns:
  0: No error
  1: Error detected

  Located in:
  SelfTest_Cordic.h
  SelfTest_Cordic.c
```

6.24 MOTIF test

MOTIF IP can be used in four different modes:

1. Stand alone Multi channel pattern mode
2. Hall sensor mode
3. Quadrature decoder mode
4. Quadrature decoder alone Multi-channel pattern mode

MOTIF self test implements the quadrature decoder mode and validate the functionality by capturing the resolution of quadrature clock. This function starts the Motif module and Modulation output from motif is started with existing values. It reports the status of the interrupts and clears the triggered interrupts. It compares the defined Multi-Channel Mode Pattern with output modulation value, returns the result OK_STATUS if values match or returns ERROR_STATUS if the values do not match.

Function

```
uint8_t SelfTest_Motif_Start(stl_motif_cfg_handle_t *hPtr)
  Returns:
  0: No error
  1: Error detected
Parameters:
  hPtr: Pointer to the MOTIF configuration structure
  Located in:
  SelfTest_Motif.c
  SelfTest_Motif.h
```

7 List of certified libraries

Table 4 PSOC™ Control C3 certified libraries and their respective versions

Library name	Version	GitHub commit
mtb-stl	3.2.1	612874a7ecedfddd73e117e0a197006315ab20de

8 Summary

This application note described 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 PSOC™ Control C3, designers can comply with regulations while maintaining or reducing electronic systems cost. Use of PSOC™ Control C3 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 will help you program your part for proper flash and ROM diagnostic testing.

1. Build a project in ModusToolbox™ and store the checksum value set to 0x0000 in the source file.

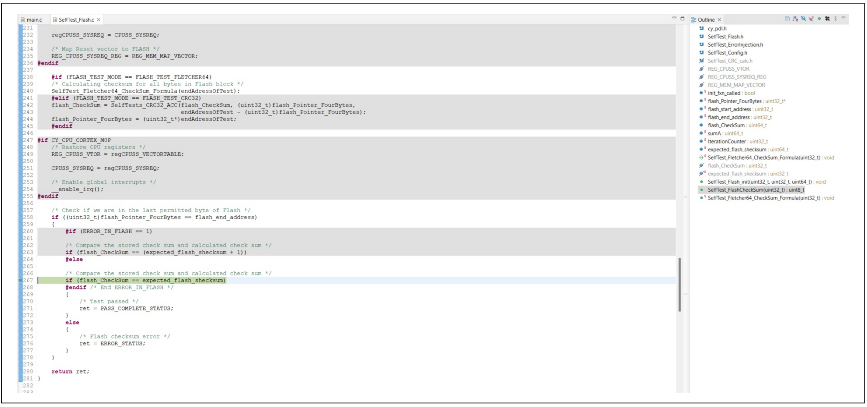
   For the GCC compiler:

```
#if defined(__GNUC__)
/* Allocate last 8 bytes in Flash for flash checksum for PSOC™ Control C3 */
static volatile const uint64_t flash_StoredCheckSum __attribute__((used,

section(".flash_checksum"))) =
    0x0000000000000000ULL;
#endif
```

2. Read the calculated flash checksum. There are two ways:

   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 Figure 10

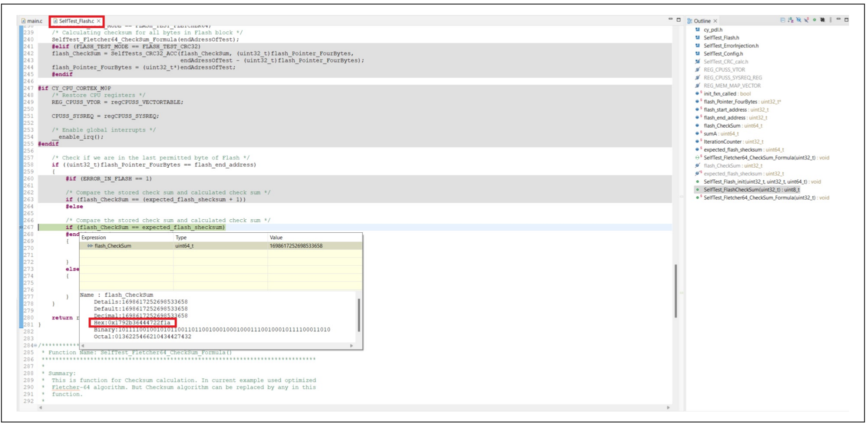


```
    regCPUSS_SYSREQ = CPUSS_SYSREQ;

    /* Map Reset vector to FLASH */
    REG_CPUSS_SYSREQ_REG = REG_MEM_MAP_VECTOR;
#endif

    #if (FLASH_TEST_MODE == FLASH_TEST_FLETCHER64)
    /* Calculating checksum for all bytes in Flash block */
    SelfTest_Fletcher64_CheckSum_Formula(endAdressOfTest);
    #elif (FLASH_TEST_MODE == FLASH_TEST_CRC32)
    flash_CheckSum = SelfTests_CRC32_ACC(flash_CheckSum, (uint32_t)flash_Pointer_FourBytes,
                                         endAdressOfTest - (uint32_t)flash_Pointer_FourBytes);
    flash_Pointer_FourBytes = (uint32_t*)endAdressOfTest;
    #endif

#if CY_CPU_CORTEX_M0P
    /* Restore CPU registers */
    REG_CPUSS_VTOR = regCPUSS_VECTORTABLE;

    CPUSS_SYSREQ = regCPUSS_SYSREQ;

    /* Enable global interrupts */
    __enable_irq();
#endif

    /* Check if we are in the last permitted byte of Flash */
    if ((uint32_t)flash_Pointer_FourBytes == flash_end_address)
    {
        #if (ERROR_IN_FLASH == 1)

        /* Compare the stored check sum and calculated check sum */
        if (flash_CheckSum == (expected_flash_shecksum + 1))
        #else

        /* Compare the stored check sum and calculated check sum */
        if (flash_CheckSum == expected_flash_shecksum)
        #endif /* End ERROR_IN_FLASH */
        {
            /* Test passed */
            ret = PASS_COMPLETE_STATUS;
        }
        else
        {
            /* Flash checksum error */
            ret = ERROR_STATUS;
        }
    }

    return ret;
}
```

```
cy_pdl.h
SelfTest_Flash.h
SelfTest_ErrorInjection.h
SelfTest_Config.h
SelfTest_CRC_calc.h
REG_CPUSS_VTOR
REG_CPUSS_SYSREQ_REG
REG_MEM_MAP_VECTOR
init_fxn_called : bool
flash_Pointer_FourBytes : uint32_t*
flash_start_address : uint32_t
flash_end_address : uint32_t
flash_CheckSum : uint64_t
sumA : uint64_t
IterationCounter : uint32_t
expected_flash_shecksum : uint64_t
SelfTest_Fletcher64_CheckSum_Formula(uint32_t) : void
flash_CheckSum : uint32_t
expected_flash_shecksum : uint32_t
SelfTest_Flash_init(uint32_t, uint32_t, uint64_t) : void
SelfTest_FlashCheckSum(uint32_t) : uint8_t
SelfTest_Fletcher64_CheckSum_Formula(uint32_t) : void
```

Figure 10 Stored Checksum in Debug Mode

      2. Press [F10] to single step past the breakpoint location and hover the mouse over the variable "flash_CheckSum." A value stored in this variable should appear, as shown in Figure 11.



```
    /* Calculating checksum for all bytes in Flash block */
    SelfTest_Fletcher64_CheckSum_Formula(endAdressOfTest);
    #elif (FLASH_TEST_MODE == FLASH_TEST_CRC32)
    flash_CheckSum = SelfTests_CRC32_ACC(flash_CheckSum, (uint32_t)flash_Pointer_FourBytes,
                                         endAdressOfTest - (uint32_t)flash_Pointer_FourBytes);
    flash_Pointer_FourBytes = (uint32_t*)endAdressOfTest;
    #endif

#if CY_CPU_CORTEX_M0P
    /* Restore CPU registers */
    REG_CPUSS_VTOR = regCPUSS_VECTORTABLE;

    CPUSS_SYSREQ = regCPUSS_SYSREQ;

    /* Enable global interrupts */
    __enable_irq();
#endif

    /* Check if we are in the last permitted byte of Flash */
    if ((uint32_t)flash_Pointer_FourBytes == flash_end_address)
    {
        #if (ERROR_IN_FLASH == 1)

        /* Compare the stored check sum and calculated check sum */
        if (flash_CheckSum == (expected_flash_shecksum + 1))
        #else

        /* Compare the stored check sum and calculated check sum */
        if (flash_CheckSum == expected_flash_shecksum)
        #end
        {


        }
        else
        {


        }
    }

    return r
}

/**********
 * Function Name: SelfTest_Fletcher64_CheckSum_Formula()
 ********************************************************************************
 *
 * Summary:
 *  This is function for Checksum calculation. In current example used optimized
 *  Fletcher-64 algorithm. But Checksum algorithm can be replaced by any in this
 *  function.
 *
```

Expression	Type	Value
flash_CheckSum	uint64_t	1698617252698533658

```
Name : flash_CheckSum
    Details:1698617252698533658
    Default:1698617252698533658
    Decimal:1698617252698533658
    Hex:0x1792b36444722f1a
    Binary:1011110010010101100110110010001000100011100100010111100011010
    Octal:0136225466210434427432
```

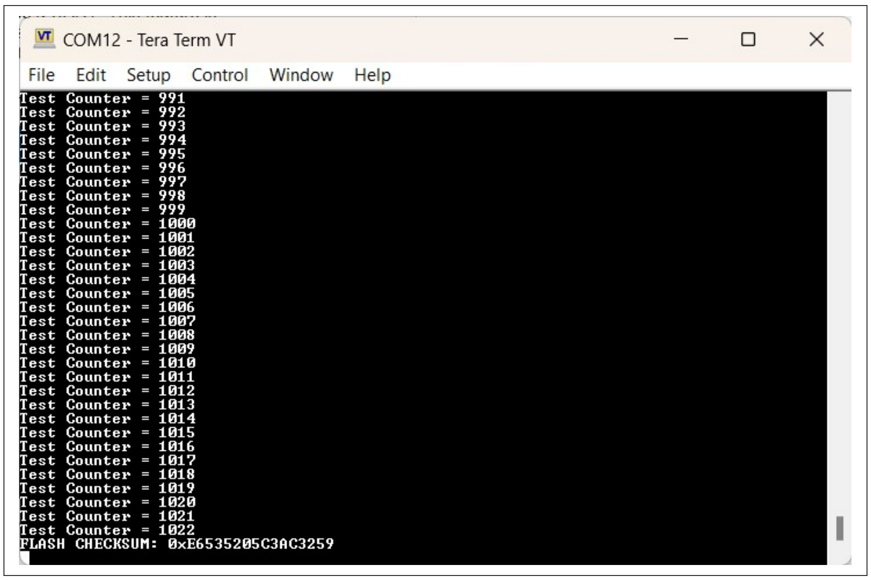
```
cy_pdl.h
SelfTest_Flash.h
SelfTest_ErrorInjection.h
SelfTest_Config.h
SelfTest_CRC_calc.h
REG_CPUSS_VTOR
REG_CPUSS_SYSREQ_REG
REG_MEM_MAP_VECTOR
init_fxn_called : bool
flash_Pointer_FourBytes : uint32_t*
flash_start_address : uint32_t
flash_end_address : uint32_t
flash_CheckSum : uint64_t
sumA : uint64_t
IterationCounter : uint32_t
expected_flash_shecksum : uint64_t
SelfTest_Fletcher64_CheckSum_Formula(uint32_t) : void
flash_CheckSum : uint32_t
expected_flash_shecksum : uint32_t
SelfTest_Flash_init(uint32_t, uint32_t, uint64_t) : void
SelfTest_FlashCheckSum(uint32_t) : uint8_t
SelfTest_Fletcher64_CheckSum_Formula(uint32_t) : void
```

Figure 11 Stored checksum in debug mode

   b. Read the checksum using the communication protocol:

      1. To speed up the process of testing the flash checksum outputs, use a UART. This feature is implemented in Class B firmware. It will print the calculated checksum value when the

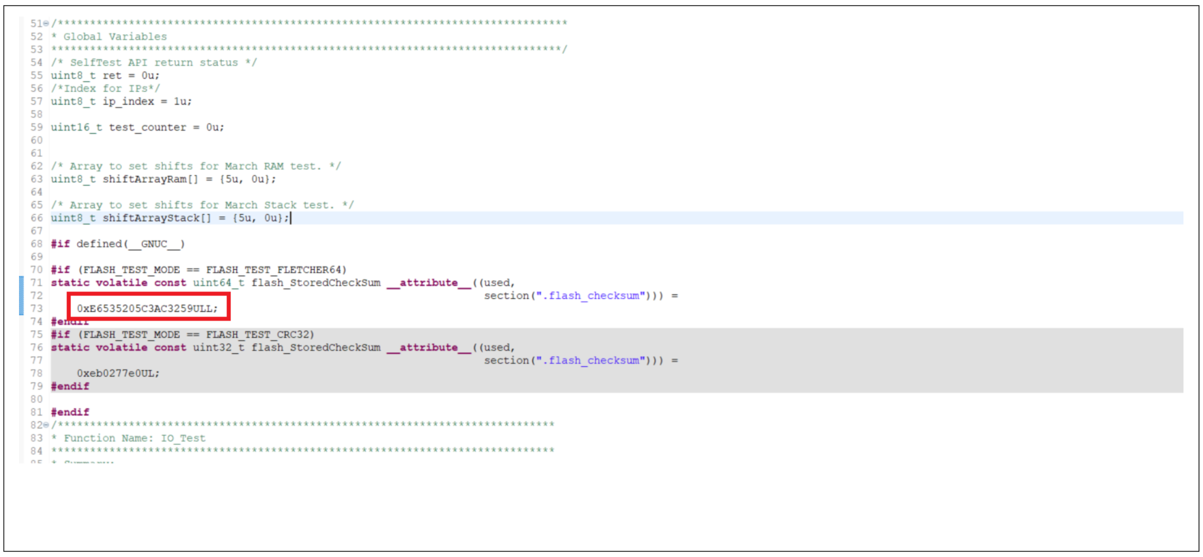
stored flash checksum does not match the calculated flash checksum. To use this project, set the UART parameters shown in Figure 12.



```
Test Counter = 991
Test Counter = 992
Test Counter = 993
Test Counter = 994
Test Counter = 995
Test Counter = 996
Test Counter = 997
Test Counter = 998
Test Counter = 999
Test Counter = 1000
Test Counter = 1001
Test Counter = 1002
Test Counter = 1003
Test Counter = 1004
Test Counter = 1005
Test Counter = 1006
Test Counter = 1007
Test Counter = 1008
Test Counter = 1009
Test Counter = 1010
Test Counter = 1011
Test Counter = 1012
Test Counter = 1013
Test Counter = 1014
Test Counter = 1015
Test Counter = 1016
Test Counter = 1017
Test Counter = 1018
Test Counter = 1019
Test Counter = 1020
Test Counter = 1021
Test Counter = 1022
FLASH CHECKSUM: 0xE6535205C3AC3259
```

Figure 12 Checksum output using UART

3. Copy this checksum value and store it in the checksum location, but remember that PSOC™ Control C3 uses little endian format. The project for the GCC compiler is shown in Figure 13.

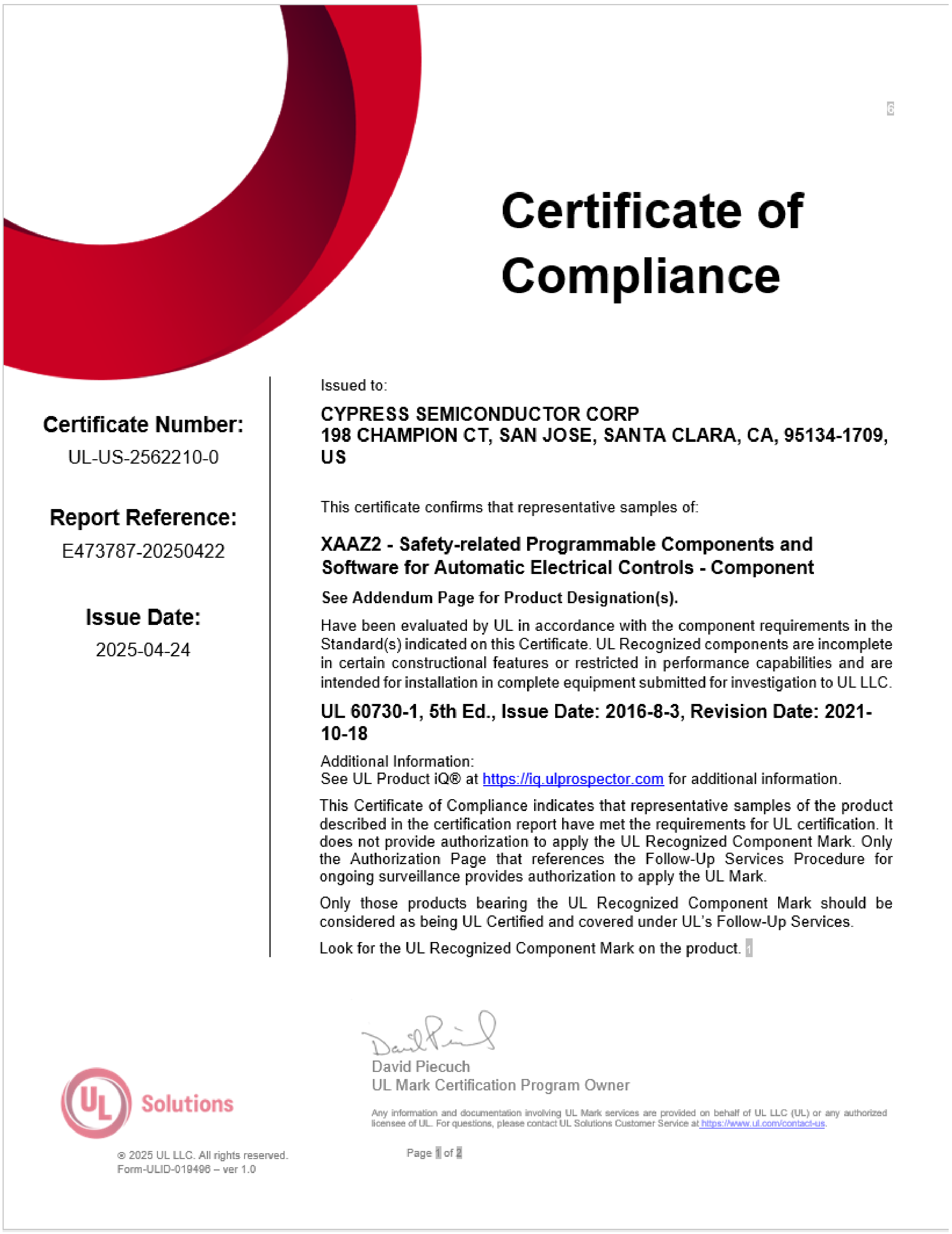


```
/*******************************************************************************
* Global Variables
*******************************************************************************/
/* SelfTest API return status */
uint8_t ret = 0u;
/*Index for IPs*/
uint8_t ip_index = 1u;

uint16_t test_counter = 0u;


/* Array to set shifts for March RAM test. */
uint8_t shiftArrayRam[] = {5u, 0u};

/* Array to set shifts for March Stack test. */
uint8_t shiftArrayStack[] = {5u, 0u};

#if defined(__GNUC__)

#if (FLASH_TEST_MODE == FLASH_TEST_FLETCHER64)
static volatile const uint64_t flash_StoredCheckSum __attribute__((used,
                                                                   section(".flash_checksum"))) =
    0xE6535205C3AC3259ULL;
#endif
#if (FLASH_TEST_MODE == FLASH_TEST_CRC32)
static volatile const uint32_t flash_StoredCheckSum __attribute__((used,
                                                                   section(".flash_checksum"))) =
    0xeb0277e0UL;
#endif

#endif
/*******************************************************************************
* Function Name: IO_Test
*******************************************************************************
```

Figure 13 Reassign Checksum Constant with Actual Checksum

4. Compile the project and program PSOC™ Control C3 device

B Appendix B: IEC 60730-1 certificate of compliance



6

Certificate of Compliance

Certificate Number:
UL-US-2562210-0

Report Reference:
E473787-20250422

Issue Date:
2025-04-24

Issued to:

CYPRESS SEMICONDUCTOR CORP
198 CHAMPION CT, SAN JOSE, SANTA CLARA, CA, 95134-1709, US

This certificate confirms that representative samples of:

XAAZ2 - Safety-related Programmable Components and Software for Automatic Electrical Controls - Component

See Addendum Page for Product Designation(s).

Have been evaluated by UL in accordance with the component requirements in the Standard(s) indicated on this Certificate. UL Recognized components are incomplete in certain constructional features or restricted in performance capabilities and are intended for installation in complete equipment submitted for investigation to UL LLC.

UL 60730-1, 5th Ed., Issue Date: 2016-8-3, Revision Date: 2021-10-18

Additional Information:
See UL Product iQ® at https://iq.ulprospector.com for additional information.

This Certificate of Compliance indicates that representative samples of the product described in the certification report have met the requirements for UL certification. It does not provide authorization to apply the UL Recognized Component Mark. Only the Authorization Page that references the Follow-Up Services Procedure for ongoing surveillance provides authorization to apply the UL Mark.

Only those products bearing the UL Recognized Component Mark should be considered as being UL Certified and covered under UL's Follow-Up Services.

Look for the UL Recognized Component Mark on the product. 1

[illegible]
David Piecuch
UL Mark Certification Program Owner

Any information and documentation involving UL Mark services are provided on behalf of UL LLC (UL) or any authorized licensee of UL. For questions, please contact UL Solutions Customer Service at https://www.ul.com/contact-us.

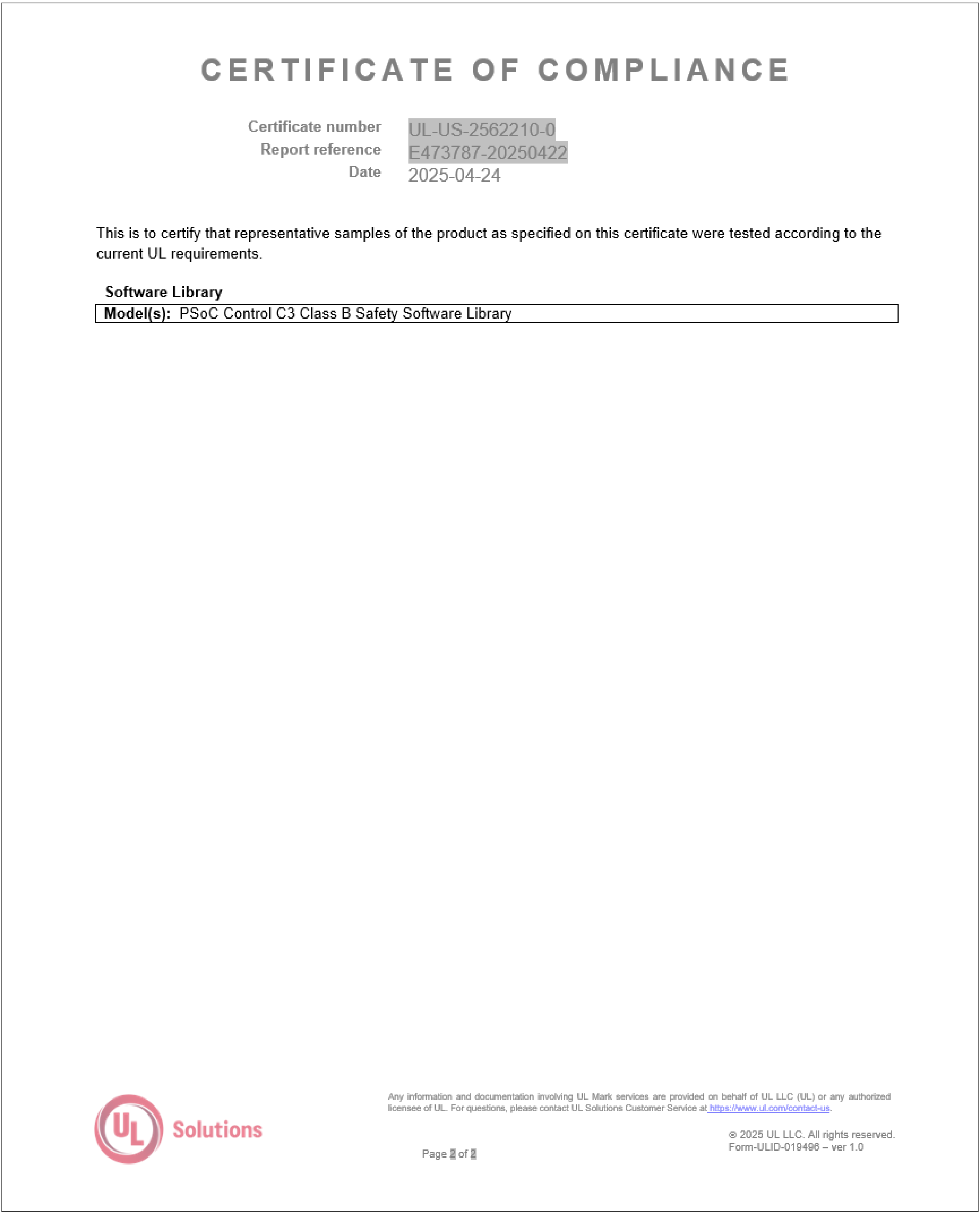
UL Solutions

© 2025 UL LLC. All rights reserved.
Form-ULID-019496 – ver 1.0

Page 1 of 2

Figure 14 **UL compliance certificate**

B Appendix B: IEC 60730-1 certificate of compliance



CERTIFICATE OF COMPLIANCE

Certificate number	UL-US-2562210-0
Report reference	E473787-20250422
Date	2025-04-24

This is to certify that representative samples of the product as specified on this certificate were tested according to the current UL requirements.

Software Library

Model(s):	PSoC Control C3 Class B Safety Software Library

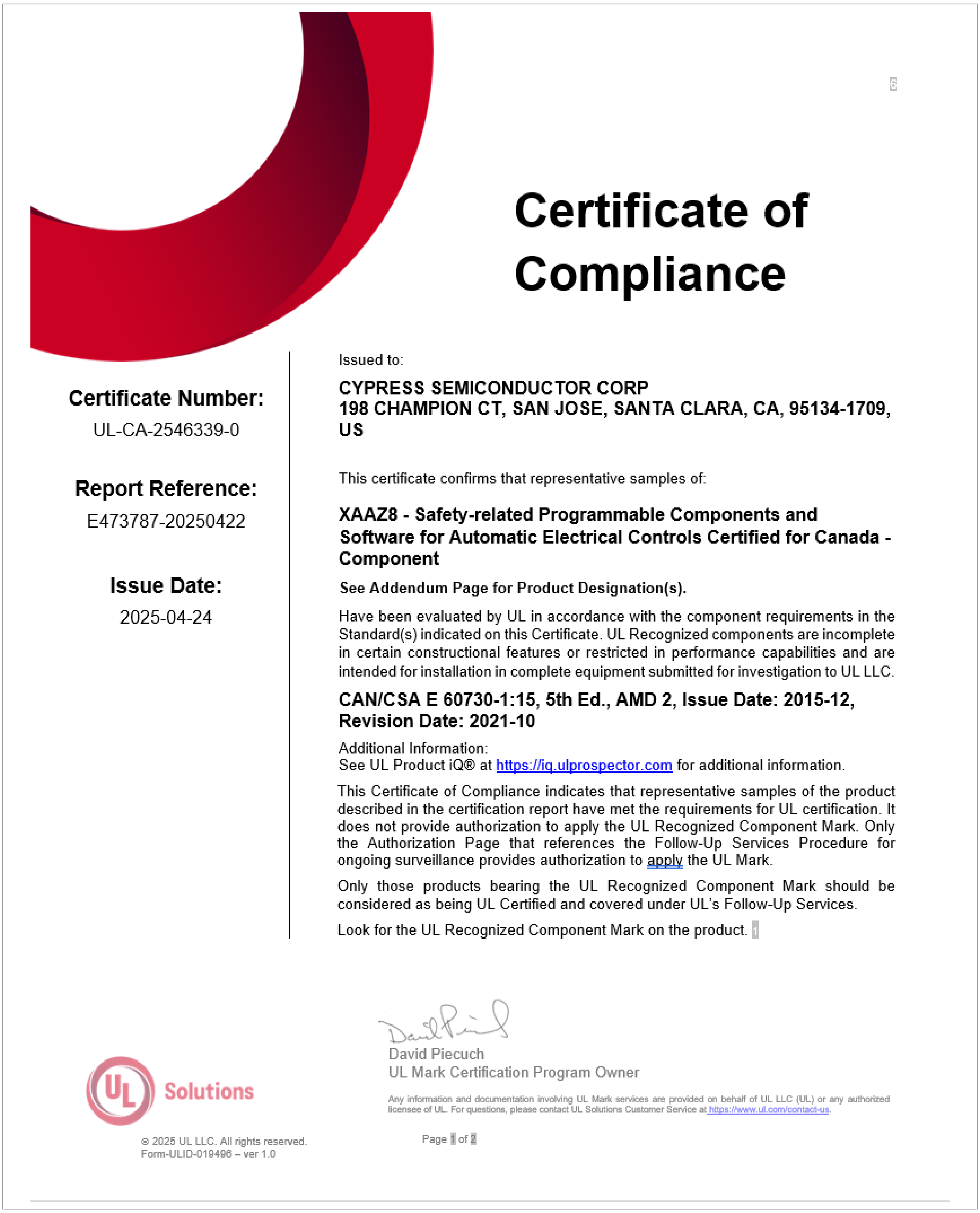
UL Solutions

Any information and documentation involving UL Mark services are provided on behalf of UL LLC (UL) or any authorized licensee of UL. For questions, please contact UL Solutions Customer Service at https://www.ul.com/contact-us.

Page 2 of 2

© 2025 UL LLC. All rights reserved.
Form-ULID-019496 – ver 1.0

Figure 15 **UL compliance certificate**



Certificate of Compliance

Certificate Number:
UL-CA-2546339-0

Report Reference:
E473787-20250422

Issue Date:
2025-04-24

Issued to:

CYPRESS SEMICONDUCTOR CORP
198 CHAMPION CT, SAN JOSE, SANTA CLARA, CA, 95134-1709, US

This certificate confirms that representative samples of:

XAAZ8 - Safety-related Programmable Components and Software for Automatic Electrical Controls Certified for Canada - Component

See Addendum Page for Product Designation(s).

Have been evaluated by UL in accordance with the component requirements in the Standard(s) indicated on this Certificate. UL Recognized components are incomplete in certain constructional features or restricted in performance capabilities and are intended for installation in complete equipment submitted for investigation to UL LLC.

CAN/CSA E 60730-1:15, 5th Ed., AMD 2, Issue Date: 2015-12, Revision Date: 2021-10

Additional Information:
See UL Product iQ® at https://iq.ulprospector.com for additional information.

This Certificate of Compliance indicates that representative samples of the product described in the certification report have met the requirements for UL certification. It does not provide authorization to apply the UL Recognized Component Mark. Only the Authorization Page that references the Follow-Up Services Procedure for ongoing surveillance provides authorization to apply the UL Mark.

Only those products bearing the UL Recognized Component Mark should be considered as being UL Certified and covered under UL's Follow-Up Services.

Look for the UL Recognized Component Mark on the product.

[illegible]
David Piecuch
UL Mark Certification Program Owner

UL Solutions

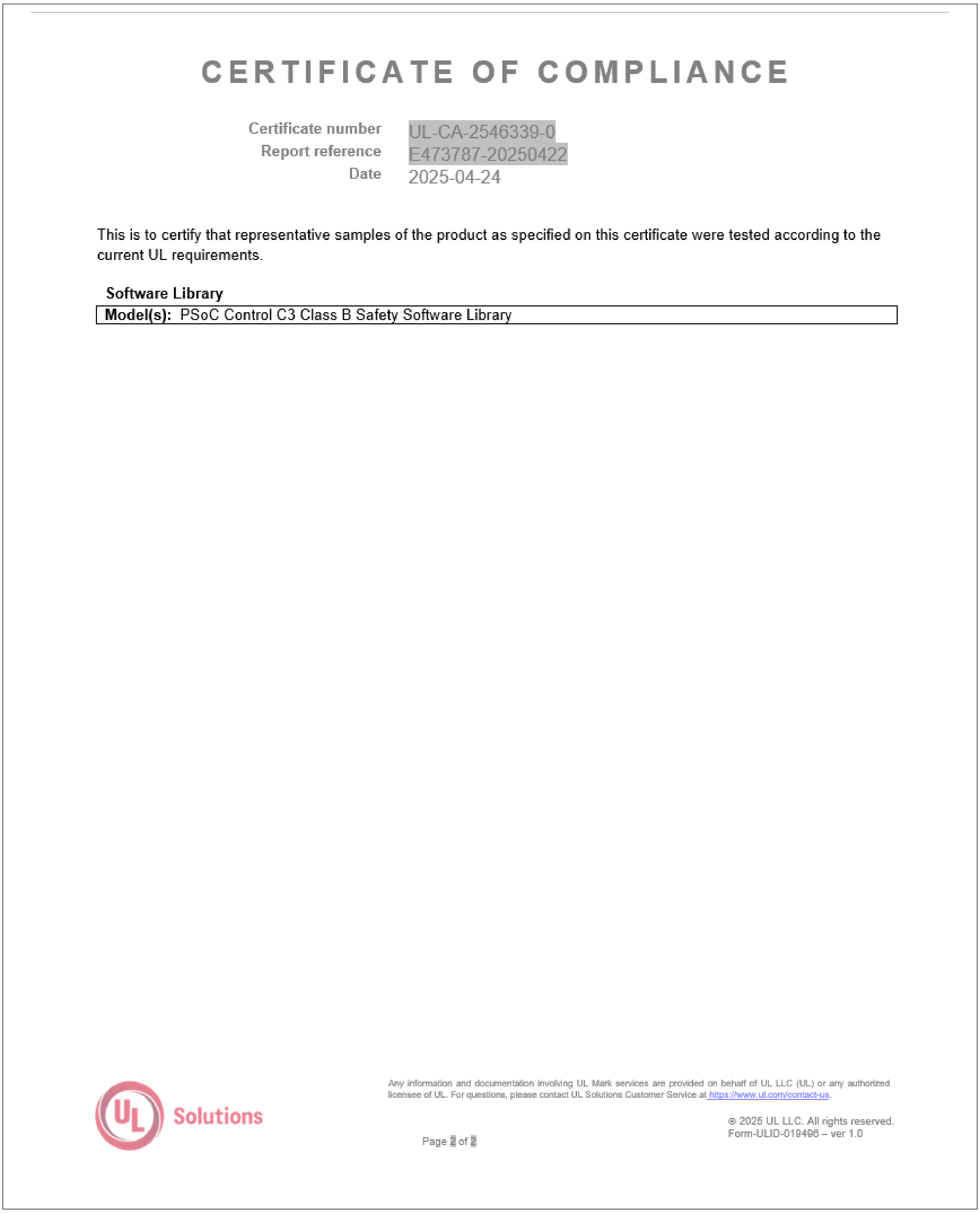
Any information and documentation involving UL Mark services are provided on behalf of UL LLC (UL) or any authorized licensee of UL. For questions, please contact UL Solutions Customer Service at https://www.ul.com/contact-us.

© 2025 UL LLC. All rights reserved.
Form-ULID-019496 – ver 1.0

Page 1 of 2

Figure 16 UL compliance certificate

B Appendix B: IEC 60730-1 certificate of compliance



CERTIFICATE OF COMPLIANCE

Certificate number	UL-CA-2546339-0
Report reference	E473787-20250422
Date	2025-04-24

This is to certify that representative samples of the product as specified on this certificate were tested according to the current UL requirements.

Software Library

Model(s): PSoC Control C3 Class B Safety Software Library

UL Solutions

Any information and documentation involving UL Mark services are provided on behalf of UL LLC (UL) or any authorized licensee of UL. For questions, please contact UL Solutions Customer Service at https://www.ul.com/contact-us.

Page 2 of 2

© 2025 UL LLC. All rights reserved.
Form-ULID-019496 – ver 1.0

Figure 17 UL compliance certificate

C Appendix C. Supported part numbers

Table 5 PSOC™ Control families supported by the certified libraries

Device family	Kit name	Revision
PSC3P2	KIT_PSC3M5_EVK	*A
PSC3M3		
PSC3P5		
PSC3M5		

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	Third edition, Feb 2019
Target device	PSC3P2, PSC3M3, PSC3P5 PSC3M5	Production
Target compiler	GCC	v11.3.1
Generation tool	ModusToolbox™	v3.4
Peripheral Driver Library	PDL	mtb-pdl-cat1 V3.16.0
MISRA checking tool	Coverity Static Analysis Tool	2022.12.0

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.
4.6	A	Typedefs that indicate size and signedness should be used in place of the basic numerical types.	Variables declared using basic numerical types are used as the expression to iterate the for loop. This doesnot have any major impact on the code functionality.

(table continues...)

Table 7 (continued) Deviated rules

MISRA-C:2012 rule	Rule class (R/A)	Rule description	Description of deviation(s)
4.8	A	If a pointer to a structure or union is never dereferenced within a Translation Unit, then the implementation of the object should be hidden.	The implementation of an object need only be hidden if all pointers to that type in a translation unit are never dereferenced and there are no other reasons for the internal details of the structure/union to be known.
4.9	A	A function should be used in preference to a function-like macro where yet are interchangeable.	Using function like macros unavoidable because use of platform PDL.
5.5	R	Identifiers shall be distinct from macro names	The diagnostic rule requires that macro identifiers should be distinct within the limits recommended by standard. C90: 31 first characters C99: 63 first characters
5.8	R	Identifiers that define objects or functions with external linkage shall be unique	An identifier with external keyword should be unique in a program. The name should not be used by other identifiers .
5.9	A	Identifiers that define objects or functions with internal linkage should be unique	An identifier with external keyword should be unique in a program. The name should not be used by other identifiers .
8.3	R	All declarations of an object or function shall use the same names and type qualifiers	It demands that the declaration and definition are identical, and the declaration must be in prototype format
8.4	R	A compatible declaration shall be visible when an object or function with external linkage is defined	Use extern keyword for identifier wherever is possible
8.5	R	An external object or function shall be declared once in one and only one file	Objects or functions with external linkage should be declared once.
8.6	R	An identifier with external linkage shall have exactly one external definition.	Declared but never defined in C. Defined in assembly files.
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.

(table continues...)

Table 7 (continued) Deviated rules

MISRA-C:2012 rule	Rule class (R/A)	Rule description	Description of deviation(s)
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 a evaluation. This is ok since the only time the volatile variable has more than 1 operation per evaluation is in the ISR.
14.3	R	Controlling expressions shall not be invariant	Boolean expression always evaluates true unavoidable due to use of PDL
15.5	A	A function should have a single point of exit at the end.	A function should have only one 'return' statement, which must come after all the other statements in the function's body. Multiple 'return' statements could obscure the code and make it harder to maintain.
18.4	A	The +, -, += and -= operators should not be applied to an expression of pointer type	Pointers should not be used in expressions with the operators '+', '-', '+=', and '-=' but can be used with the subscript '[]' and increment/decrement ('++'/'--') operators.

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/electronics/programmable electronic safety-related systems, IEC 61508-2 Edition 2.0, 2010-04*: Available online

Revision history

Document revision	Date	Description of changes
**	2025-04-04	Initial release
*A	2025-05-02	Web release
*B	2025-05-28	Added UL compliance certificate and details of certified libraries

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™.

Edition 2025-05-28

Published by

Infineon Technologies AG

81726 Munich, Germany

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

Email: erratum@infineon.com

Document reference
IFX-oxy1738732986361