PSoC® 4 – IEC 60730 Class B and IEC 61508 SIL Safety Software Library

Authors: Vasyl Parovinchak, Taras Kuzo

Associated Project: Yes
Associated Part Family: CY8C40xx, CY8C41xx, CY8C42xx, CY8C42xx-M
Software Version: : PSoC Creator™ 3.1
Related Application Notes: AN78175

AN89056 describes the PSoC® 4 IEC 60730 Class B and IEC 61508 safety integrity level (SIL) Safety Software Library and includes example projects 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.

Contents

1 Introduction

Today, the majority of automatic electronic controls for home appliance 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 end product to enter an unknown or unsafe state.

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

This 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 test 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:

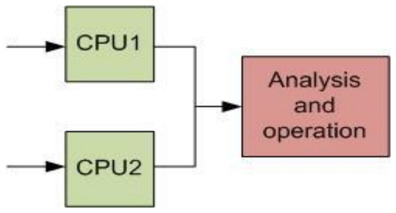
- Single channel with functional test
- Single channel with periodic self-test
- Dual channel without comparison (see Figure 1)

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 more costly 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 IEC 61508-2 Annex A

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

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

The process starts by evaluating the risk for each hazardous event of the controlled equipment. Typically, diagnostic coverage and safe failure fraction are then determined based on the likelihood of each failure occurring, combined with the consequence of the failure. This weights 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: $\geq 10^{-6}$ to $< 10^{-5}$ (1 failure in 11 years)
- SIL 2: $\geq 10^{-7}$ to $< 10^{-6}$ (1 failure in 114 years)
- SIL 3: $\geq 10^{-8}$ to $< 10^{-7}$ (1 failure in 1,144 years)
- SIL 4: $\geq 10^{-9}$ to $< 10^{-8}$ (1 failure in 11,446 years)

4 IEC 60730 Class B and IEC 61508 Requirements

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

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

Table 1. Components Required to Be Tested for Single-Channel Structures

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

The user application must determine whether interrupts need to be enabled or disabled during 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

The Safety Software Library described in this application note can be used with PSoC 4 devices. The library includes APIs that are designed to maximize application reliability through fault detection.

Some self-tests can be applied by only adding an appropriate API function to the **.c* and **.h* files from the Class B Safety Software Library. Others can be applied by adding an appropriate API function to the **.c* and **.h* files and modifying the project schematic.

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

- Self-test functions to help meet the IEC 60730-1 Class B and IEC 61508-2 standards.
  - CPU registers: Test for stuck bits
  - Program counter: Test for jumps to the correct address
  - Program flow: Test for checking correct firmware program flow
  - Interrupt handling and execution: Test for proper interrupt calling and periodicity
  - Clock: Test for wrong frequency
  - Flash (invariable memory): Test for memory corruption
  - SRAM (variable memory): Test for stuck bits and proper memory addressing
  - Stack overflow: Test for checking stack overflow with the program data memory during program execution
  - Digital I/O: Test for pins short
  - ADC and DAC: Test for proper functionality
  - Comparator: Test for proper functionality
  - Communications (UART, SPI): Test for correct data reception
- Additional self-test functions that PSoC 4 can support due to programmable interconnect. Often, the end application also needs these self-tests, even though they are not provided in Appendix B of IEC 60730-1 or IEC 61508-2.
  - Opamp test
  - Universal Digital Block (UDB) configuration registers test
  - Startup configuration registers test
  - Watchdog test: Test for chip reset
  - Additional windowed watchdog timer (WDT) to monitor firmware execution

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

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

6 API Functions for PSoC 4

6.1 CPU Registers Test

PSoC 4 with the Cortex-M0 CPU has 16-bit and 32-bit registers:

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

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

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

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

Function

```
uint8  SelfTest_CPU_Registers(void)

Returns:  0  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 device should not continue to function because its behavior can be unpredictable and therefore potentially unsafe.

6.2 Program Counter Test

The PSoC 4 CPU program counter R15 register is part of the CPU register set. To test these registers, a checkerboard test is commonly used; the addresses 0x5555 and 0xAAAA must be allocated for this test. 0x5555 and 0xAAAA represent the checkerboard bit patterns.

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 located in flash memory at different addresses are called. For PSoC 4, this can be done by using the linker script in an **.ld* file.

   ```
   NV_CONFIG1 0x5555 :
   {
       . = 0x00;
       *(PC5555);
   } >rom =0

   NV_CONFIG2 0xAAAA :
   {
       . = 0x00;
       *(PCAAAA);
   } >rom =0
   ```

   In the example project, it is already added to the *custom_cm0gcc.ld* file. You can add the linker file by choosing **Project** > **Build Setting** > **Linker** > **Custom Linker Script**.
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
```

Note: For PSoC 4, the *custom_cm0gcc.ld* file must be added to the linker.

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

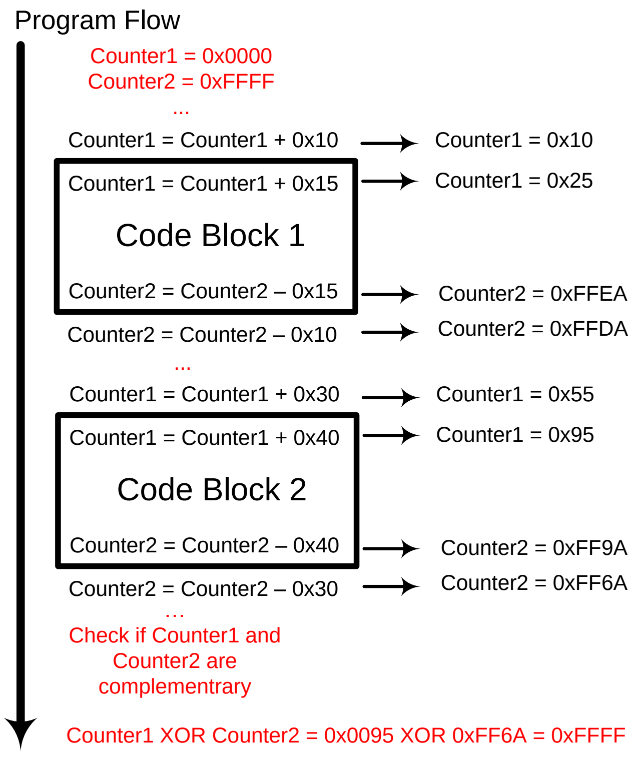
6.3 Program Flow Test

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.

Figure 2. Program Flow Test


Program Flow
Counter1 = 0x0000
Counter2 = 0xFFFF
...
Counter1 = Counter1 + 0x10
Counter1 = 0x10
Counter1 = Counter1 + 0x15
Counter1 = 0x25
Code Block 1
Counter2 = Counter2 – 0x15
Counter2 = 0xFFEA
Counter2 = Counter2 – 0x10
Counter2 = 0xFFDA
...
Counter1 = Counter1 + 0x30
Counter1 = 0x55
Counter1 = Counter1 + 0x40
Counter1 = 0x95
Code Block 2
Counter2 = Counter2 – 0x40
Counter2 = 0xFF9A
Counter2 = Counter2 – 0x30
Counter2 = 0xFF6A
...
Check if Counter1 and Counter2 are complementrary
Counter1 XOR Counter2 = 0x0095 XOR 0xFF6A = 0xFFFF

6.4 Interrupt Handling and Execution Test

The PSoC 4 interrupt controllers provide the mechanism for hardware resources to change the program address to a new location independent of the current execution in main code. They also handle 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.

The SelfTest_Interrupt() function is called to check the interrupt controller operation. Calling the function starts the timers.

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 4 shows the interrupt self-test flow chart.

Figure 3. PSoC Creator Schematic for Interrupt Self-Test

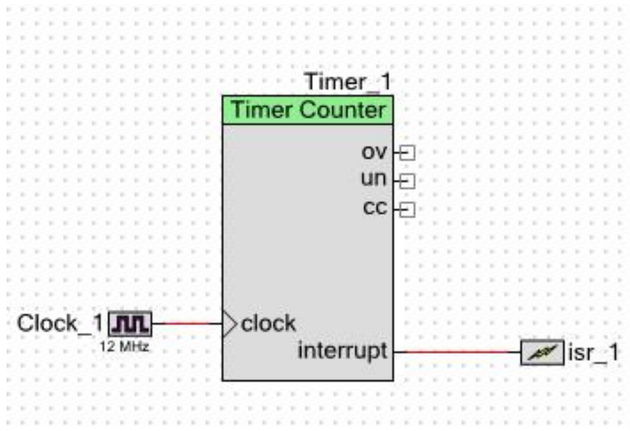
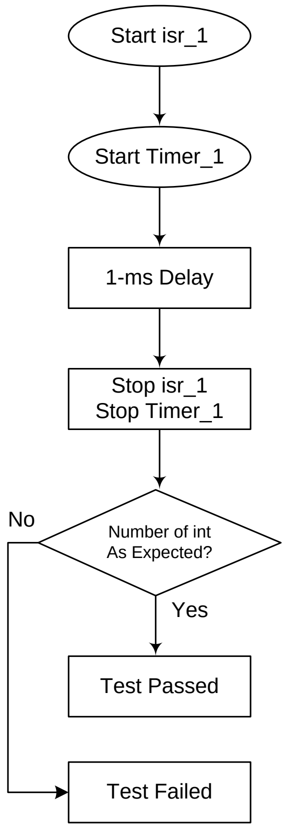


Figure 4. Interrupt Self-Test Flow Chart



6.5 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 ± 60 percent. If accuracy greater than 60 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.

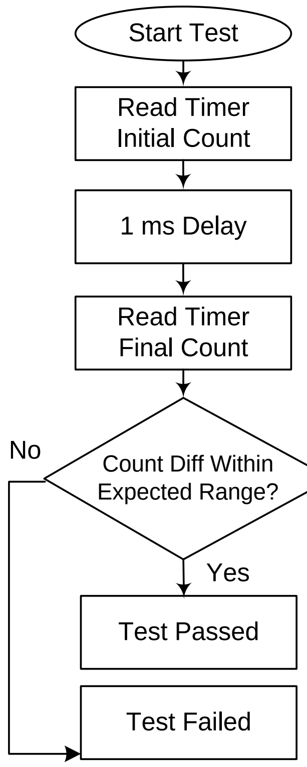
Function

```
uint8  SelfTest_Clock(void)

Returns:  0   No error
    Not 0  Error detected

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 tests 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 5 shows the clock self-test flow chart.

Figure 5. Clock Self-Test Flow Chart



6.6 Flash (Invariable Memory) Test

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

6.6.1 Checksum Method

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. The Fletcher’s 64-bit method was chosen because it is sufficiently reliable.

You can change the Fletcher’s checksum method to any checksum method in function SelfTest_CheckSum_Formula() in the *SelfTest_Flash.c* file.

PSOC_FLASH_SIZE in the *SelfTest_Flash.h* file defines the flash size that needs to be monitored.

The proposed 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. When the test reaches the end of flash minus 8 bytes (0x8FF8 on 32-KB devices), it stops. 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.

6.6.2 Programming Steps

Before starting the test, you need to set the correct precalculated checksum, as described in Set Checksum in Flash (Invariable Memory) Test.

Function

```
uint8 SelfTest_FlashCheckSum()

        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.

6.7 SRAM (Variable Memory) Test

Note: PSoC 4 devices include an on-chip SRAM of up to 16 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 nondestructive. Destructive tests destroy the contents of memory during testing, whereas nondestructive tests preserve the memory contents. While the test algorithm used in this library is destructive, it is encapsulated in code that first saves the memory contents before a test and then restores the contents after completion.

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

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

6.7.1 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, Cypress 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.

6.7.2 March C Algorithm

March test notations:

>	Arrange address sequence in ascending order
<	Arrange address sequence in descending order
<>	Arrange address sequence in either ascending or descending order
r0	Indicate read operation (reads '0' from a memory cell)
r1	Indicate read operation (reads '1' from a memory cell)
w0	Indicate write operation (writes '0' from a memory cell)
w1	Indicate write operation (writes '1' from a memory cell)

```
MarchC
{
  <> (w0)
  > (r0 then w1)
  > (r1 then w0)
  <> (r0)
  < (r0 then w1)
  < (r1 then w0)
  > (r0)
}
```

Function

```
uint8  SelfTests_SRAM_March(void)
```

```
Returns:  1   Error detected
    2   Still testing
    3   Pass, all RAM is tested
```

```
Located in:   SelfTest_RAM.c
              SelfTest_RAM.h
```

This function is called to do the March SRAM test in run time without data corruption.

Using the start address and end address pointers, the function performs the run-time March C test by backing up the area of SRAM under test to another reserved part of SRAM and then restoring the data.

The reserved part of SRAM is also tested using the March test before data is copied. This area of memory corrupts; therefore, storage or placement of variables in it is prohibited. Control over this memory is assigned to the user. It is recommended that you allocate it to an array that is not used and set a compiler directive prohibiting optimization for this array. This is demonstrated in the *AN_89056_Cpu* project.

The reserved area of SRAM is located at the end of SRAM just before the stack area and is set using the following parameters in the *SelfTest_SRAM_March.s* file:

```
MARCH_BUFF_ADDR_START:     .word (CYDEV_SRAM_BASE + CYDEV_SRAM_SIZE - CYDEV_STACK_SIZE
- RESERVE_BLOCK_SIZE)

MARCH_BUFF_ADDR_END:       .word (CYDEV_SRAM_BASE + CYDEV_SRAM_SIZE -
CYDEV_STACK_SIZE)

.if (TEST_BLOCK_SRAM_SIZE>TEST_BLOCK_STACK_SIZE)
    .equ RESERVE_BLOCK_SIZE,     TEST_BLOCK_SRAM_SIZE
.else
    .equ RESERVE_BLOCK_SIZE,     TEST_BLOCK_STACK_SIZE
.endif
```

The location of different sections inside SRAM is next. For example:

```
CYDEV_SRAM_BASE = 0x20000000;
CYDEV_SRAM_SIZE = 0x00001000;
```

`[0x20000000; (0x20001000-CYDEV_STACK_SIZE- RESERVE_BLOCK_SIZE)]`	Variable SRAM
`[(0x20001000-CYDEV_STACK_SIZE- RESERVE_BLOCK_SIZE); (0x20001000-CYDEV_STACK_SIZE)]`	Buffer for March C test (reserved part of SRAM)
`[(0x20001000-CYDEV_STACK_SIZE); 0x20001000]`	Stack SRAM

During the call, this function tests one block of SRAM. The size of the block is the same as that of the reserved buffer area in SRAM. It can be set using the following parameters in the *SelfTest_SRAM_March.s* file:

```
.equ TEST_BLOCK_SRAM_SIZE, 0x00000400
```

This test covers the variable SRAM area only and not the stack area. If the block is tested successfully and the start address of the reserved area is reached, the test returns 0x03. If the block is tested successfully, but the start address of the reserved area is not reached, the test returns 0x02. If an error is detected, the test returns 0x01.

Function

```
void SelfTests_Init_March_SRAM_Test(uint8 shift)
```

```
Located in:  SelfTest_RAM.c
SelfTest_RAM.h
```

This function initializes the SRAM base address and should be called in two cases:

- Before the first call of SelfTests_SRAM_March(). This case initializes the start test address the first time.
- When all SRAM is tested and SelfTests_SRAM_March() returns the status “Pass” (0x02u). This case reinitializes the start test address.

The parameter “shift” sets the shift from the start test address. It is used to set different start addresses and allows testing to cover all block boundaries of previous test passes. Without use of the “shift” parameter, the first and last bytes of each block will not be tested for interaction with the adjacent test blocks. Typically, the “shift” parameter will alternate between 0 and half the test block size of each full test sequence.

For example:

Case 1:

```
TEST_BLOCK_SRAM_SIZE = 10;
CYDEV_SRAM_SIZE      = 100;
SelfTests_Init_March_SRAM_Test(0x00u);
```

During each SelfTests_SRAM_March() call, the test range will be:

[0-9]; [10-19]; [20-29] …..[80-99]; [90-99]

Case 2:

```
TEST_BLOCK_SRAM_SIZE = 10;
CYDEV_SRAM_SIZE      = 100;
SelfTests_Init_March_SRAM_Test(0x05u);
```

During each SelfTests_SRAM_March() call, the test range will be:

[5-14]; [15-24]; [25-34] …..[85-94]; [95-99]

You can change the “shift” parameter after each full SRAM test.

Function

```
uint8  SelfTests_Stack_March (void)

Returns:  1   Error detected
    2   Still testing
    3   Pass, all RAM is tested

Located in:   SelfTest_RAM.c
              SelfTest_RAM.h
```

This function is called to do the March stack tests in run time without data corruption.

Using the start address and end address pointers, the function performs a run-time March C test by backing up the area of stack under test to a reserved part of SRAM and then restoring the data.

The reserved part of SRAM is also tested using the March test before data copy. This function uses the same reserved area of SRAM as that used for the SRAM March test.

The reserved part of SRAM corrupts; therefore, storage or placement of variables is prohibited in this area of memory. Control over this memory is assigned to the user. It is recommended that you allocate it to an array that is not used and set a compiler directive prohibiting optimization for this array.

During the call, this function tests one block of SRAM. The size of the block is the same as that of the reserved buffer area in SRAM. It can be set using the following parameters in the *SelfTest_SRAM_March.s* file:

```
.equ TEST_BLOCK_STACK_SIZE, 0x00000040
```

If the block is tested successfully and the end address of SRAM is reached (meaning all stack RAM is tested), the test returns 0x03. If the block is tested successfully, but the end address of SRAM is not reached, the test returns 0x02. If an error is detected, the test returns 0x01.

Function

```
void SelfTests_Init_March_Stack_Test (uint8 shift)

Located in:   SelfTest_RAM.c
             SelfTest_RAM.h
```

This function initializes the stack SRAM base address and should be called in two cases:

- Before the first call of SelfTests_Stack_March(). This case initializes the start test address the first time.
- When all stack SRAM is tested and SelfTests_Stack_March() returns the status “Pass” (0x02u). This case reinitializes the start test address.

The parameter “shift” sets the shift from the start test address. It is used to set different start addresses and to cover all variants of addressing.

An example of this function’s use is the same as that described for the SelfTests_Init_March_SRAM_Test(uint8 shift) function.

6.8 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 4, the stack is located at the end of RAM and grows downward. The stack pointer is 32 bits wide and is decremented with every PUSH instruction and incremented with POP**.**

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

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

Function

```
void SelfTests_Init_Stack_Test(void)

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 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
```

6.9 Digital I/O Test

PSoC 4 provides up to 36 programmable GPIO pins. Any GPIO pin can be CapSense®, LCD, analog, or digital. Drive modes, strengths, and slew rates are programmable.

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

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

In normal operating conditions, the pin-to-ground and pin-to-VCC resistances are very high. To detect any shorts, resistance values are compared with the PSoC 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.

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

For example:

```
static const uint8 PinToTest[] =
{
        0b11011111,   /* PORT0 mask */
        0b00111111,   /* PORT1 mask */
        0b11111111,   /* PORT2 mask */
        0b11110011,   /* PORT3 mask */
        0b00000000,   /* 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’.

6.10 ADC and DAC Test

Note: Applies to CY8C41xx and CY8C42xx only.

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 ADC and IDAC analog functions. Because the PSoC 4 has no voltage reference, an IDAC with an external resistor is used.

This test is easily implemented by using the reconfigurable hardware of the PSoC device and an IDAC with an external resistor to form the voltage reference, for example, a 4.99-kΩ 1 percent external resistor. This configuration can deliver up to 3.05 V in 12-mV steps (8-bit resolution). Figure 6 shows a schematic implementation of the ADC and DAC test based on PSoC 4. Additional analog system elements are present for use in the optional comparator and opamp tests. Pin_Opamp1, Pin_Opamp2, Pin_ADC1, and Pin_ADC2 are the user pins. The ADC is configured to scan three channels. The first two channels are for test purposes, and the third is for user purposes. Other user channels can be added. ADC channel “0” is connected to the DAC via an opamp and is used for the opamp test. Channel “1” is connected to the DAC directly and is used for the ADC and DAC test.

Function

```
uint8  SelfTest_ADC(void)
```

```
Returns:  0   No error
    1   Error detected
```

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

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

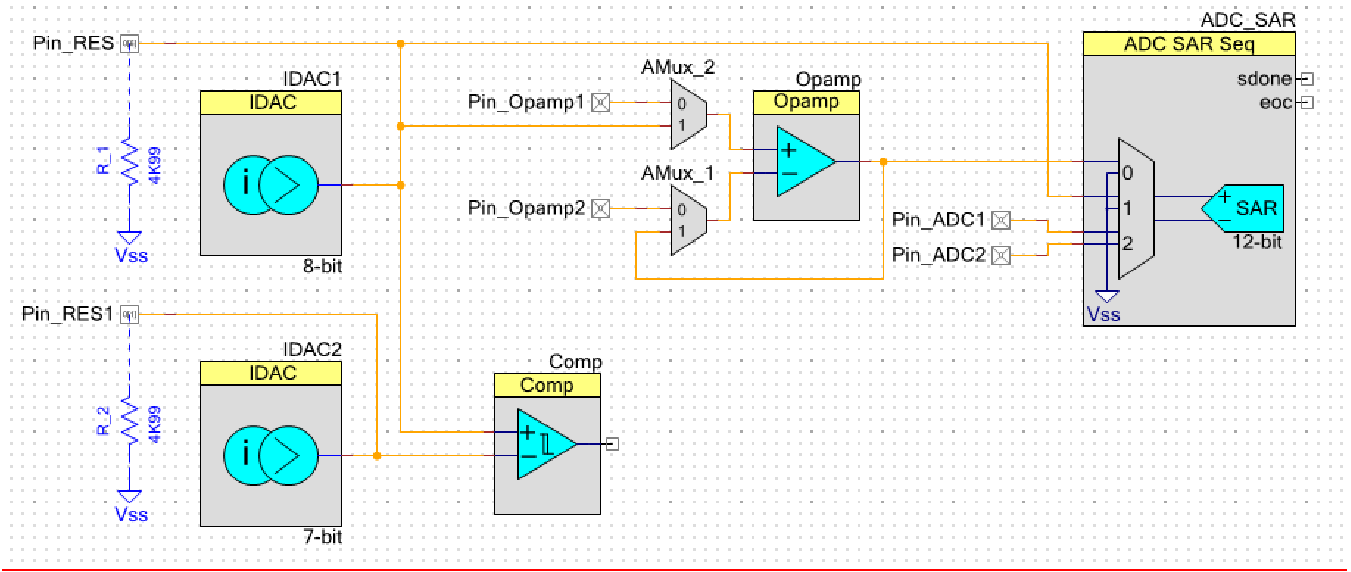
```
#define ADC_TEST_ACC 12                              // +/- ADC result value
```

To perform this test, the ADC is configured to scan channel “1”. A predefined (reference) voltage is generated using the IDAC and is sampled by the ADC.

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

The test function saves all the component configurations before testing and restores them after the test ends.

Figure 6. PSoC Implementation of ADC and DAC Tests



6.11 Comparator Test

Note: Applies to CY8C41xx and CY8C42xx only.

The comparator functional test is performed using IDAC1 and IDAC2 with external resisters to generate two voltage references. These reference voltages are sequentially put into the input of the analog comparator, forcing the output value of the comparator to change. The output state is read and compared with the expected value. When the test is a success, the function returns 0; otherwise, it returns 1.

Function

```
uint8  SelfTest_Comparator(void)

Returns:   0   No error
    1   Error detected

Located in:   SelfTest_Analog.h

        SelfTest_Analog.c
```

Figure 6 shows the PSoC schematic implementation of the comparator test and includes the optional ADC, DAC, and opamp tests. The output values of the comparator are analyzed at different polarities of the input signals. If the output signal changes during this operation, the test is passed.

The test function saves the ADC and DAC configurations before testing and restores them after the test ends.

6.12 Opamp Test

Note: Applies to CY8C41xx and CY8C42xx only.

The opamp test is not provided in the IEC 60730 standard, but it is useful when using the opamp in critical blocks.

The opamp test adheres to the principle of “independent output comparison” testing defined in section H.2.18.8 of the IEC 60730 standard. It provides a fault/error control technique by which inputs/outputs that are designed to be within specified tolerances are compared.

The purpose of the test is to test the opamp analog functions. This test is easily implemented using the reconfigurable hardware of the PSoC device and can be performed by using the IDAC with an external resistor as a predefined voltage reference. The opamp output signal can be regularly converted by the ADC to test opamp functions. Figure 6 shows the schematic implementation of the opamp test based on PSoC 4 and includes optional ADC, DAC, and comparator tests. Pin_Opamp1 and Pin_Opamp2 are the user pins. Before the test, the analog multiplexers disconnect the opamp from the user pins and reconfigure the internal connections as required for the opamp self-test. After the test is complete, the user pin connections are restored.

Function

```
uint8  SelfTest_Opamp(void)
```

```
Returns:  0   No error
    1   Error detected
```

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

The opamp test accuracy is defined in *SelfTest_Analog.h*. A 12-bit ADC is used for testing:

```
#define OPAMP_TEST_ACC 12                    // +/- Opamp result value
```

This function implements the opamp test by measuring the opamp output signal using the ADC in two cases:

- IDAC1 out: 0.096 V; expected ADC result: 0.096 V
- IDAC1 out: 1.52 V; expected ADC result: 1.52 V

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

The test function restores ADC and DAC configurations to the default after the test ends.

6.13 Communications UART Test

Note: Applies to CY8C41xx and CY8C42xx only.

This test implements the UART internal data loopback test. 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.

Function

```
uint8  SelfTest_UART(void)
```

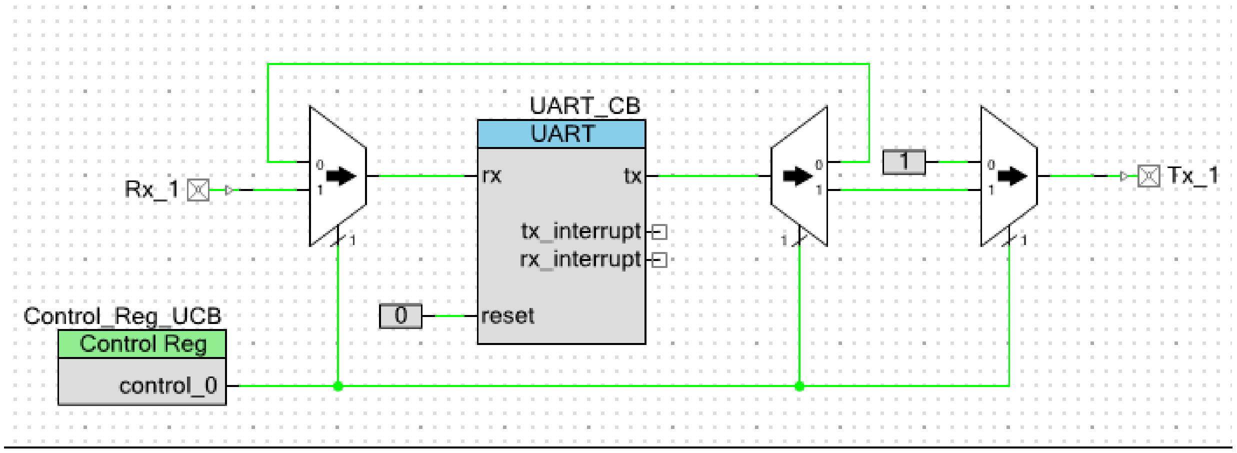
```
Returns:  1   Error detected
    2   Pass test with current value, but test is not complete testing full range from
        0x00 to 0xFF
    3   Pass, completed testing full range from 0x00 to 0xFF
    4   ERROR_TX_NOT_EMPTY
    5   ERROR_RX_NOT_EMPTY
    6   ERROR_TX_NOT_ENABLE
    7   ERROR_RX_NOT_ENABLE
```

```
Located in:   SelfTest_UART.h
        SelfTest_UART.c
```

Figure 7 shows the PSoC schematic implementation of the UART test. 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.

The test function saves the component configuration before testing and restores them after the test ends. 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.

Figure 7. PSoC Implementation of UART Test



6.14 Communications SPI Test

Note: Applies to CY8C41xx and CY8C42xx only.

This test implements the SPI internal data loopback test. 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.

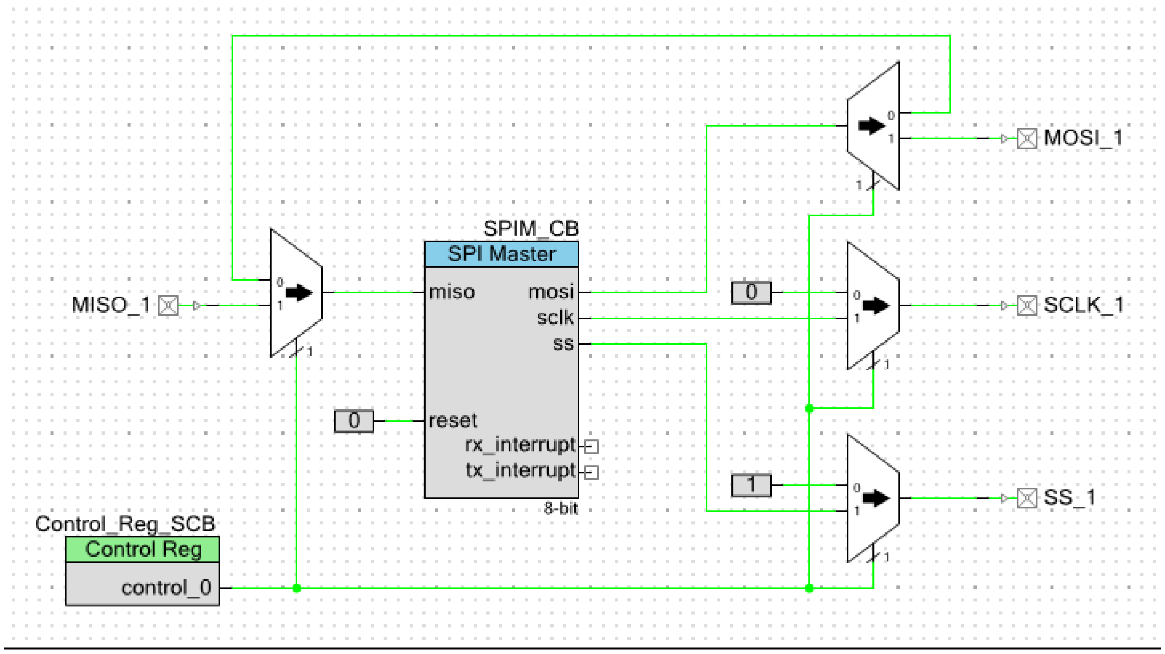
Function

```
uint8  SelfTest_SPI (void)

Returns:  1  Error detected
    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

Located in:  SelfTest_SPI.h
       SelfTest_SPI.c
```

Figure 8 shows the PSoC schematic implementation of the SPI test. 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.

The test function saves all component configurations before testing and restores them after the test ends. 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.

Figure 8. PSoC Implementation of SPI Test



6.15 UDB Configuration Registers Test

Note: Applies to CY8C41xx and CY8C42xx only.

UDB configuration registers are static and configured during design build. They should not be changed during device operation and can be checked against the initial configuration.

The following functions allow you to implement UDB configuration register tests in a design. They implement two test modes:

- Store duplicates of UDB configuration registers in flash memory after device startup. Periodically, the configuration registers are compared with the 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.

Function

```
cystatus SelfTests_Save_UDB_Cfg(void)

Returns:  0  write in flash is successful
    1  error detected during flash writing

Located in:  SelfTest_UDB_CfgReg.c
             SelfTest_UDB_CfgReg.h
```

This function stores UDB configuration registers in flash memory. UDB configuration registers are located in the memory addresses from 0x400F0000 to 0x40100000 and occupy 65K bytes. But the registers do not occupy all this memory—there are gaps with unused memory. This is why an array, located in `SelfTest_UDB_CfgReg`, is used with addresses of registers that need to be tested. The register array is generated by copying all of the UDB registers from the projects *cydevice_trm.h* file.

```
const uint16 UDB_ConfRegs[UDB_REGS_COUNT]

* Number of UDB configuration registers to be counted */
#define UDB_REGS_COUNT 1551
```

The number of registers to test is defined in *SelfTest_UDB_CfgReg.h*. `UDB_REGS_COUNT` is the total number of UDB registers copied from the `cydevice_trm.h` file.

Note: This function should be called once after the initial PSoC power up and initialization before entering the main program. It writes the correct UDB configuration register values to flash. After this initial write, typically executed during manufacturing with a test command, the register values are already stored and this function does not need to be called again.

Note: The flash test should only be called after the UDB configuration registers are saved or the CRC calculated, otherwise the flash test will fail.

Function

```
uint8 selfTests_UDB_ConfigReg(uint16 RegsToTest)

Parameters: RegsToTest – number of blocks to be tested per 1 function call. 128 – used
in demo project.

Returns:  1  Error detected
    2  Test in progress
    3  Test completed OK


Located in:   SelfTest_UDB_CfgReg.c
              SelfTest_UDB_CfgReg.h
```

This function checks the UDB configuration registers. The number of registers to be tested on each function call can be set using the UDB_REGISTERS_PER_TEST constant in the *SelfTest_UDB_CfgReg.h* file. This constant should be passed as an argument to the `selfTests_UDB_ConfigReg` function.

```
#define UDB_REGISTERS_PER_TEST (128u)
```

There are two modes of checking:

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

You can define the mode by defining UDB_CFG_REGS_MODE to one of the following constants in the *SelfTest_UDB_CfgReg.h* file:

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

This mode stores duplicates of registers in flash and compares the registers with duplicates. It returns a fail if the values are different. Registers can be restored in this mode. The `SelfTests_Save_UBD_Cfg()` function is used to store duplicates in flash. Registers are automatically saved to the last flash rows.

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

In this mode, the function calculates a CRC-16 of registers and stores the CRC in flash. Later, function calls recalculate the CRC-16 and compare it with the saved value. It returns a fail if the values are different.

6.16 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 cyfitter configuration registers.

These startup configuration registers are typically static and are located in the *cyfitter_cfg.c* 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 schematic or configuration, other configuration registers may be generated in the *cyfitter_cfg.c* file. You must change the list of required registers (the recommended registers are generated in the `cyfitter_cfg()` function).

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. If the UDB configuration test is configured for the `CFG_REGS_TO_FLASH_MODE` then the startup configuration registers are stored into the row(s) of flash immediately preceding the UDB configuration registers. **Note** This function should be called once after the initial PSoC 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.

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-16 calculation and verification
- Register comparison with duplicated copy

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

```
#define STARTUP_CFG_REGS_MODE              CFG_REGS_CRC_MODE / 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_cfg` 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 `SelfTests_Save_cfg.h`

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

In this mode, the function calculates a CRC-16 of registers and stores the CRC in flash. Later, function calls recalculate the CRC-16 and compare it with the saved value. It returns a fail (1) if the values are different.

6.17 Watchdog Test

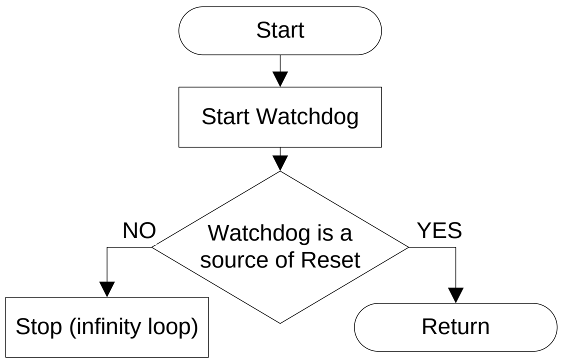
This function implements the watchdog functional test. The function starts the WDT and runs an infinite loop. If the WDT works, it generates a reset. After the reset, the function analyzes the reset source. If the watchdog is the source of the reset, the function returns; otherwise, the infinite loop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

Function

```
void   SelfTest_WDT(void)

Returns:   None

Located in:   SelfTest_WDT.h
        SelfTest_WDT.c
```

Figure 9 shows the test flow chart.

Figure 9. PSoC Implementation of WDT Test



6.18 Windowed Watchdog Timer

Note: Applies to CY8C41xx and CY8C42xx only.

The WDT increases the reliability of microprocessor-based systems. Window-selectable WDTs allow the watchdog timeout period to be adjusted, providing more flexibility to meet different processor timing requirements. The windowed watchdog circuits protect systems from running too fast or too slow.

Microprocessors executing critical or safety-related functions demand a high level of supervision to ensure proper fault detection and correction. A critical function can be defined as one for which downtime cannot be tolerated and (in many cases) one for which a repair is very costly. Such functions are found in almost every segment of the microprocessor market: patient-monitoring systems, process control plants, and safety-related automotive applications, for example.

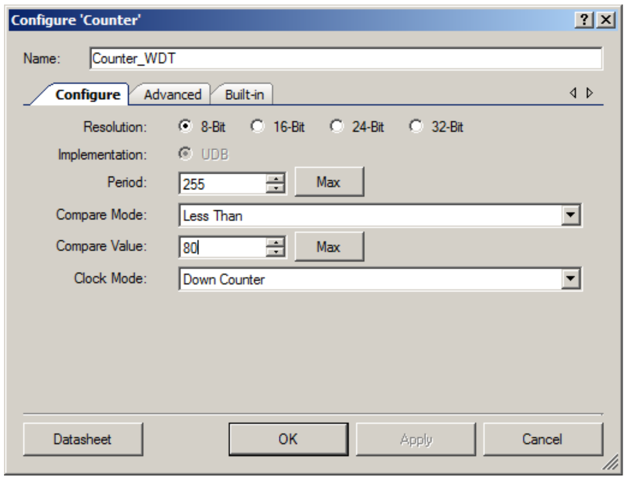
There is no windowed WDT in PSoC 4, but it can be implemented using PSoC reconfigurable hardware. The windowed WDT provides a way to demand that the ClearWDT instruction be executed, for example, only in the last quarter of the watchdog timeout period. Essentially, this enables better code flow monitoring to catch firmware bugs. For example, an application bug that results in the ClearWDT instruction repeatedly executing close to the beginning of the code flow could be interpreted as a normal operation in the non-windowed WDT mode. Most users use only the non-windowed WDT mode.

The caveat to the windowed WDT mode is that the ClearWDT instruction must be called within a prescribed window, which limits the tolerance of the clock source that drives the WDT. The tolerance of the clock source also defines the nominal watchdog period minimum and maximum.

A counter and flip-flops are used to implement the windowed WDT in PSoC 4.

Figure 10 shows the windowed WDT counter configuration. **Clock_Count** (Clock_2 in Figure 12) and **Period** (shown in Figure 10) are used to define the maximum waiting (maximum firmware execution) time for the clear command. **Compare Value** (shown in Figure 10) is used to define the width of the clearing window.

Figure 10. Windowed WDT Counter Configuration


Configure 'Counter'

Name: Counter_WDT

Configure | Advanced | Built-in

Resolution: [✓ 8-Bit] 16-Bit 24-Bit 32-Bit

Implementation: [✓ UDB]

Period: 255

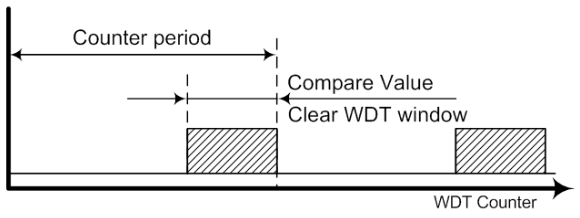
Compare Mode: Less Than

Compare Value: 80

Clock Mode: Down Counter

The down-counting counter output “comp” changes as follows during the count period, as shown in Figure 11:

Figure 11. Windowed WDT Timing Diagram



- First window: From (255u) to (80u), output “comp” is “0”.
- Second window: From (80u) to (0u), output “comp” is “1”.

The second window, from (80u) to (0u), is used to detect if the windowed WDT clear is correct.

If the windowed WDT clear happens in the first window or does not happen during the counter period, an incorrect firmware operation has occurred, and PSoC must be reset.

The clear windowed WDT means trigger “1” in the Control_Reg_ClearWDT control register.

Figure 12 shows the schematic implementation of the windowed WDT test based on PSoC 4. There are three stages in the schematic operation:

1. The flip-flop DFF1 detects if the clear happens in the second window.
2. The flip-flop DFF2 detects if the clear happens in the first window.
3. The flip-flop DFF3 detects if the clear does not happen during the period of time.

The ISR isr_Windowed_WDT is triggered in stages 2 or 3. It is used to detect a reset in the example project for the purpose of demonstration. In the customer design, a pin connected to the hardware reset can be used instead of the ISR.

Also, the windowed WDT output can be ANDed/ORed with the critical outputs to disable them in case of a WDT event.

Integrating safety inputs like windowed WDT or interlock inputs with output logic and pins can result in 100 percent hardware safety control with no CPU processing.

Figure 13 shows the flow chart of the windowed WDT operation.

Functions

There are two functions to operate the windowed WDT:

- `void Windowed_WDT_Start(void)`: This function starts operation of the windowed WDT.
- `void Windowed_WDT_Clear(void)`: This function clears the windowed WDT.

Both functions are located in the following files:

- SelfTest_Windowed_WDT.c
- SelfTest_Windowed_WDT.h

Figure 12. Windowed WDT Implementation in PSoC 4

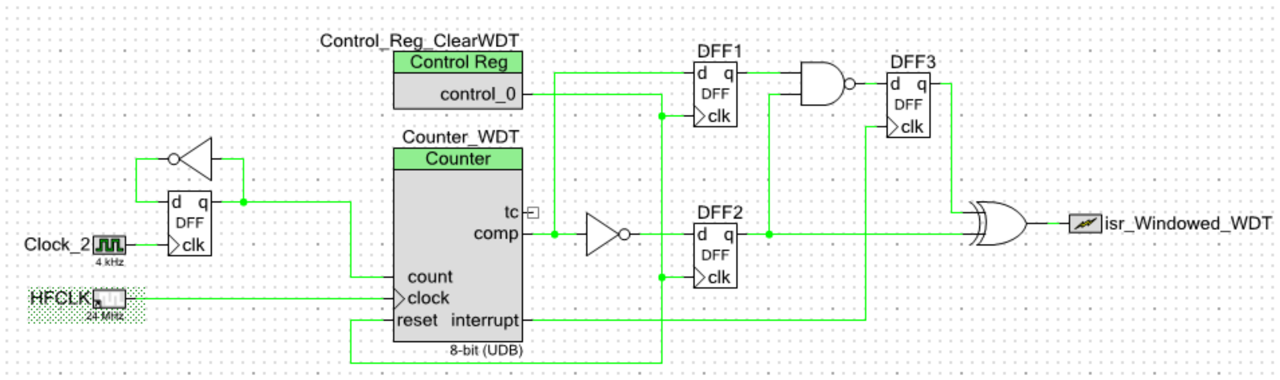
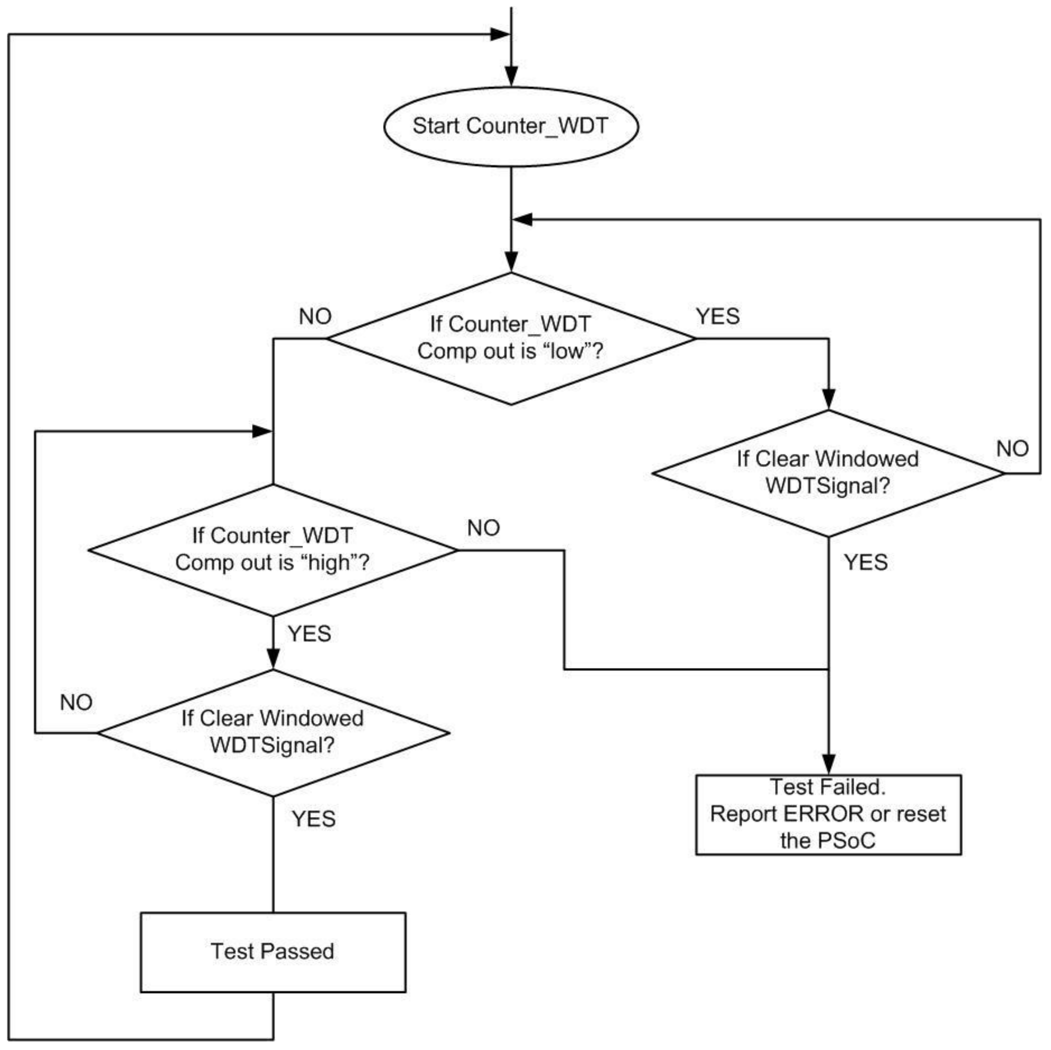


Figure 13. Flow Chart of Windowed WDT Operation


Start Counter_WDT
If Counter_WDT Comp out is “low”?
NO
YES
If Clear Windowed WDTSignal?
NO
YES
If Counter_WDT Comp out is “high”?
NO
YES
If Clear Windowed WDTSignal?
NO
YES
Test Failed. Report ERROR or reset the PSoC
Test Passed

6.19 Communications UART Data Transfer Protocol Example

Note: Applies to CY8C41xx and CY8C42xx only.

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 CRC calculated with the packet data to ensure the packet’s safe transfer. Figure 14 shows the packet format.

Figure 14. Packet Structure

STX	ADDR	DL	D0	……..(data bytes)	Dn	CRCH	CRCL

To allow the reserved start of packet maker (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 2 shows the packet field descriptions.

Table 2. Packet Field Descriptions

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

6.19.1 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.

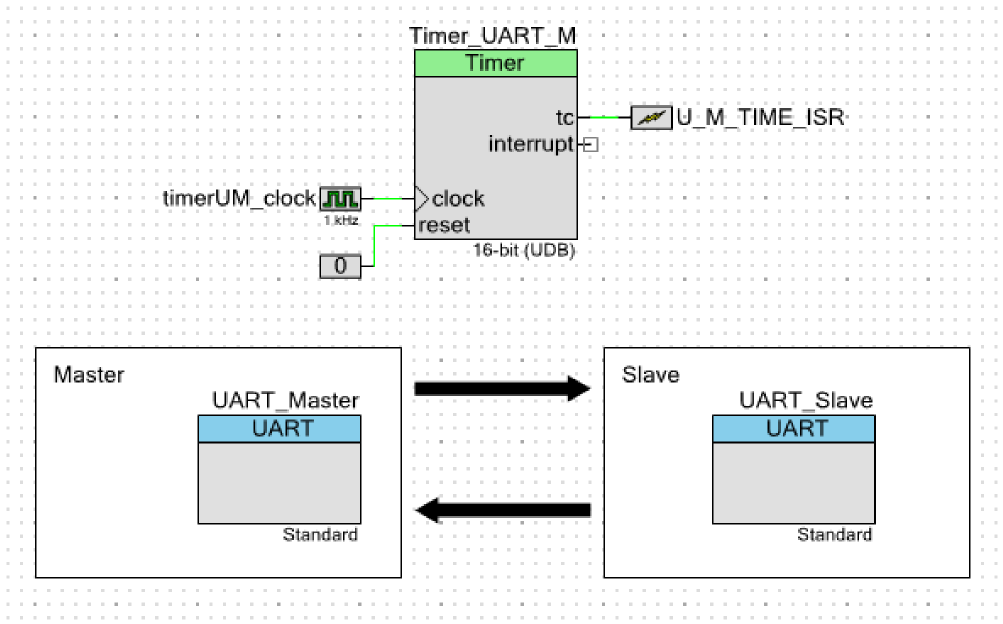
6.19.2 PSoC Implementation

Figure 15 represents the protocol implementation using PSoC. 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 15. PSoC 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(void)

Returns:   None

Located in:   UART_master_message.h
       UART_master_message.c
```

This function initializes the UART message unit.

Function 2

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

Returns:   0   No error
    1   Error detected

Located in:   UART_master_message.h
       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.

The input parameters are as follows:

- Address: Slave address for communication
- txd: Pointer to the transmitted data (request data)
- tlen: Length of the request in bytes
- rxd: Pointer to the buffer where the received data is stored (received data)
- rlen: Length of the received buffer in bytes

Function 3

```
uint8 UartMesMaster_State(void)
```

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

```
Located in:  UART_master_message.h
       UART_master_message.c
```

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

Possible results are the following:

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

Function 4

```
uint8 UartMesMaster_GetDataSize(void)
```

```
Returns: returns data size
```

```
Located in:  UART_master_message.h
       UART_master_message.c
```

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

Five Functions for Working with the Protocol Unit for the Slave

Function 1

```
void UartMesSlave_Init(uint8 address)
```

```
Returns: None
```

```
Located in:  UART_slave_message.h
       UART_slave_message.c
```

This function initializes the UART message unit. The input parameter is as follows:

- Address: Slave address

Function 2

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

```
Returns:  0  No error
    1  Error detected
```

```
Located in:  UART_slave_message.h
       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).

The input parameters are as follows:

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

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:   UART_slave_message.h

       UART_slave_message.c
```

This function returns the current state of the UART message unit. Possible results are the following:

- UM_IDLE: Last transaction process is finished.
- UM_PACKREADY: Unit has received a marker and there is received data in the buffer. The master waits for a response.
- UM_RESPOND: Unit is busy sending a response.

Function 4

```
uint8 UartMes_GetDataSize(void)
```

```
Returns:  returns data size
```

```
Located in:   UART_slave_message.h

       UART_slave_message.c
```

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

Function 5

```
uint8 * UartMesSlave_GetDataPtr(void)
```

```
Returns: return pointer to data
```

```
Located in:   UART_slave_message.h

              UART_slave_message.c
```

This function obtains a pointer to the received data.

7 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 4, designers can comply with regulations while maintaining or reducing electronic systems cost. Use of PSoC and the Safety Software Library enables the creation of a strong system-level development platform to achieve superior performance, fast time to market, and energy efficiency.

8 References

- IEC 60730 Standard, “Automatic electrical controls for household and similar use,” IEC 60730-1 Edition 3.2, 2007-03
- IEC 61508 Standard, “Functional safety of electrical/electronic/programmable electronic safety-related systems,” IEC 61508-2 Edition 2.0, 2010-04

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 PSoC Creator with the stored checksum value set to 0x0000 in the *SelfTest_Flash.c* file.

   For the GCC compiler:

```
volatile const uint64 flash_StoredCheckSum __attribute__ ((used, section
("CheckSum"))) = 0x0000000000000000u;
```

   For the MDK or RVD compiler:

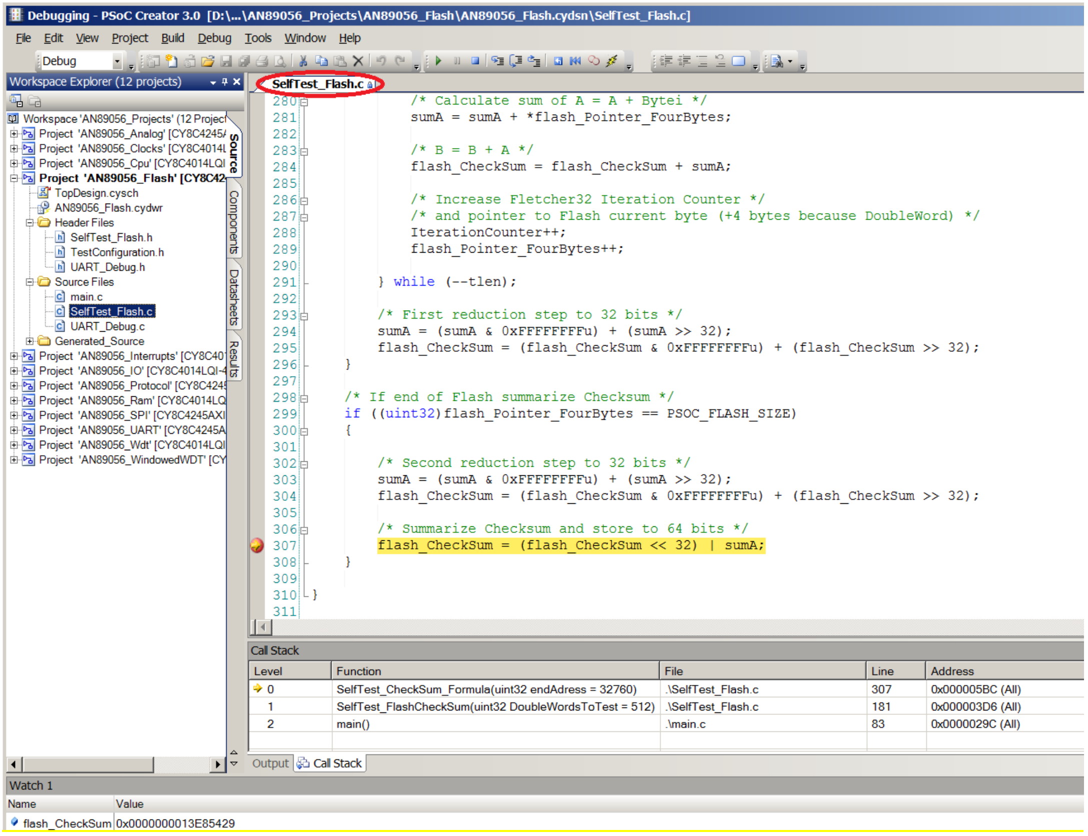
```
#if (CYDEV_CHIP_MEMBER_USED == CYDEV_CHIP_MEMBER_4A)
    volatile const uint64 flash_StoredCheckSum __attribute__ ((at(0x00007FF8))) =
0x0000000000000000;
#endif

    /* CY8C40XX */
#if (CYDEV_CHIP_MEMBER_USED == CYDEV_CHIP_MEMBER_4D)
    volatile const uint32 flash_StoredCheckSum __attribute__ ((at(0x00003FF8))) =
0x0000000000000000;
```

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

   a. Read the checksum in debug mode.

      i. Open the project file *SelfTest_Flash.c* and set the breakpoint in debug mode to the line shown in Figure 16.

Figure 16. Stored Checksum in Debug Mode.


Debugging - PSoC Creator 3.0 [D:\...\AN89056_Projects\AN89056_Flash\AN89056_Flash.cydsn\SelfTest_Flash.c]
File Edit View Project Build Debug Tools Window Help
Debug
Workspace Explorer (12 projects)
Workspace 'AN89056_Projects' (12 Project
Project 'AN89056_Analog' [CY8C4245
Project 'AN89056_Clocks' [CY8C4014L
Project 'AN89056_Cpu' [CY8C4014LQI
Project 'AN89056_Flash' [CY8C42
TopDesign.cysch
AN89056_Flash.cydwr
Header Files
SelfTest_Flash.h
TestConfiguration.h
UART_Debug.h
Source Files
main.c
SelfTest_Flash.c
UART_Debug.c
Generated_Source
Project 'AN89056_Interrupts' [CY8C40
Project 'AN89056_IO' [CY8C4014LQI-4
Project 'AN89056_Protocol' [CY8C4245
Project 'AN89056_Ram' [CY8C4014LQ
Project 'AN89056_SPI' [CY8C4245AXI
Project 'AN89056_UART' [CY8C4245A
Project 'AN89056_Wdt' [CY8C4014LQI
Project 'AN89056_WindowedWDT' [CY
Source
Components
Datasheets
Results
SelfTest_Flash.c
Output
Call Stack

```
            /* Calculate sum of A = A + Bytei */
            sumA = sumA + *flash_Pointer_FourBytes;

            /* B = B + A */
            flash_CheckSum = flash_CheckSum + sumA;

            /* Increase Fletcher32 Iteration Counter */
            /* and pointer to Flash current byte (+4 bytes because DoubleWord) */
            IterationCounter++;
            flash_Pointer_FourBytes++;

        } while (--tlen);

        /* First reduction step to 32 bits */
        sumA = (sumA & 0xFFFFFFFFu) + (sumA >> 32);
        flash_CheckSum = (flash_CheckSum & 0xFFFFFFFFu) + (flash_CheckSum >> 32);
    }

    /* If end of Flash summarize Checksum */
    if ((uint32)flash_Pointer_FourBytes == PSOC_FLASH_SIZE)
    {

        /* Second reduction step to 32 bits */
        sumA = (sumA & 0xFFFFFFFFu) + (sumA >> 32);
        flash_CheckSum = (flash_CheckSum & 0xFFFFFFFFu) + (flash_CheckSum >> 32);

        /* Summarize Checksum and store to 64 bits */
        flash_CheckSum = (flash_CheckSum << 32) | sumA;
    }

}
```

Call Stack

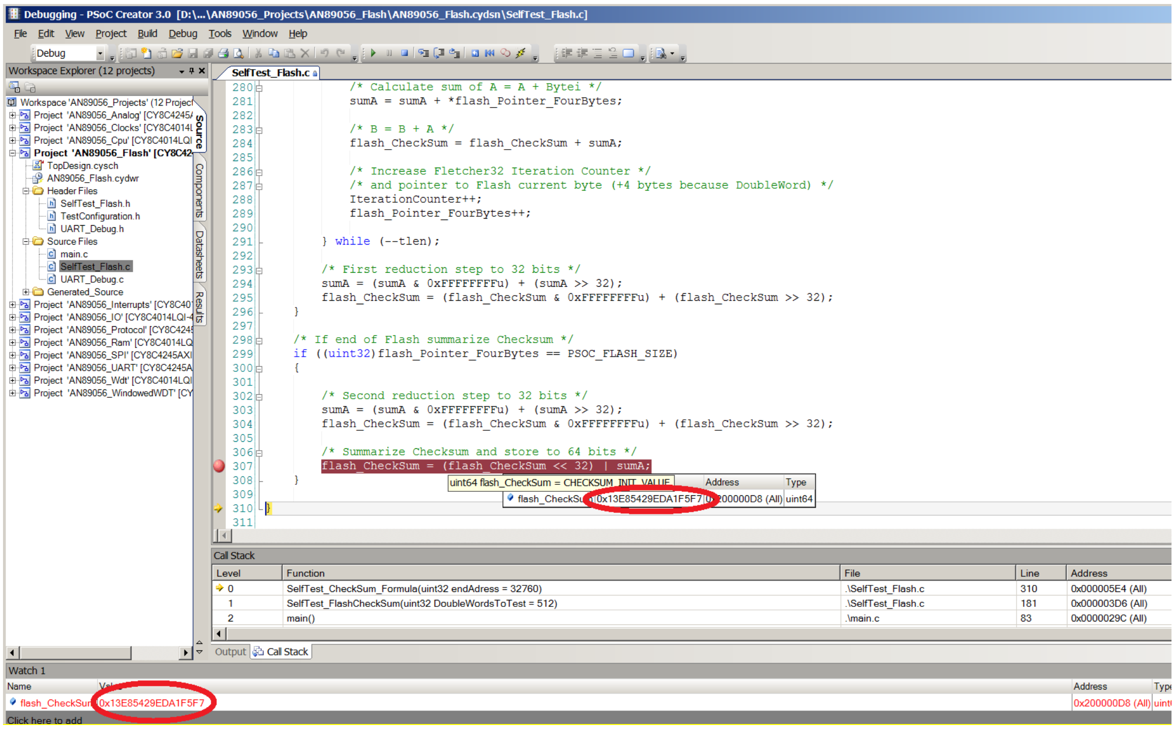
Level	Function	File	Line	Address
0	SelfTest_CheckSum_Formula(uint32 endAdress = 32760)	.\SelfTest_Flash.c	307	0x000005BC (All)
1	SelfTest_FlashCheckSum(uint32 DoubleWordsToTest = 512)	.\SelfTest_Flash.c	181	0x000003D6 (All)
2	main()	.\main.c	83	0x0000029C (All)

Watch 1

Name	Value
flash_CheckSum	0x0000000013E85429

      ii. 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 17.

Figure 17. Stored Checksum in Debug Mode.



Debugging - PSoC Creator 3.0 [D:\...\AN89056_Projects\AN89056_Flash\AN89056_Flash.cydsn\SelfTest_Flash.c]

```
                /* Calculate sum of A = A + Bytei */
                sumA = sumA + *flash_Pointer_FourBytes;

                /* B = B + A */
                flash_CheckSum = flash_CheckSum + sumA;

                /* Increase Fletcher32 Iteration Counter */
                /* and pointer to Flash current byte (+4 bytes because DoubleWord) */
                IterationCounter++;
                flash_Pointer_FourBytes++;

            } while (--tlen);

            /* First reduction step to 32 bits */
            sumA = (sumA & 0xFFFFFFFFu) + (sumA >> 32);
            flash_CheckSum = (flash_CheckSum & 0xFFFFFFFFu) + (flash_CheckSum >> 32);
        }

        /* If end of Flash summarize Checksum */
        if ((uint32)flash_Pointer_FourBytes == PSOC_FLASH_SIZE)
        {

            /* Second reduction step to 32 bits */
            sumA = (sumA & 0xFFFFFFFFu) + (sumA >> 32);
            flash_CheckSum = (flash_CheckSum & 0xFFFFFFFFu) + (flash_CheckSum >> 32);

            /* Summarize Checksum and store to 64 bits */
            flash_CheckSum = (flash_CheckSum << 32) | sumA;
        }

}
```

uint64 flash_CheckSum = CHECKSUM_INIT_VALUE

	Address	Type
flash_CheckSum 0x13E85429EDA1F5F7	0x200000D8 (All)	uint64

Call Stack

Level	Function	File	Line	Address
0	SelfTest_CheckSum_Formula(uint32 endAdress = 32760)	.\SelfTest_Flash.c	310	0x000005E4 (All)
1	SelfTest_FlashCheckSum(uint32 DoubleWordsToTest = 512)	.\SelfTest_Flash.c	181	0x000003D6 (All)
2	main()	.\main.c	83	0x0000029C (All)

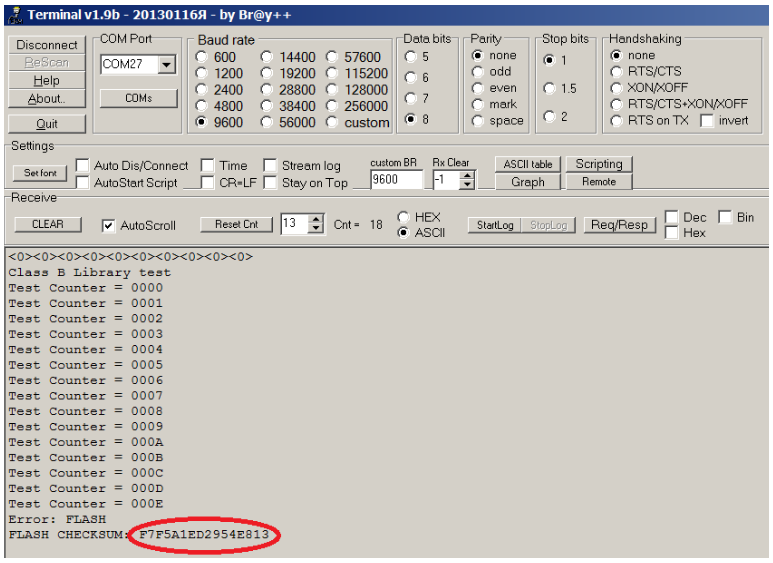
Watch 1

Name	Value	Address	Type
flash_CheckSum	0x13E85429EDA1F5F7	0x200000D8 (All)	uint64

b. Read the checksum using the communication protocol:

i. 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 18.

Figure 18. Checksum Output Using UART



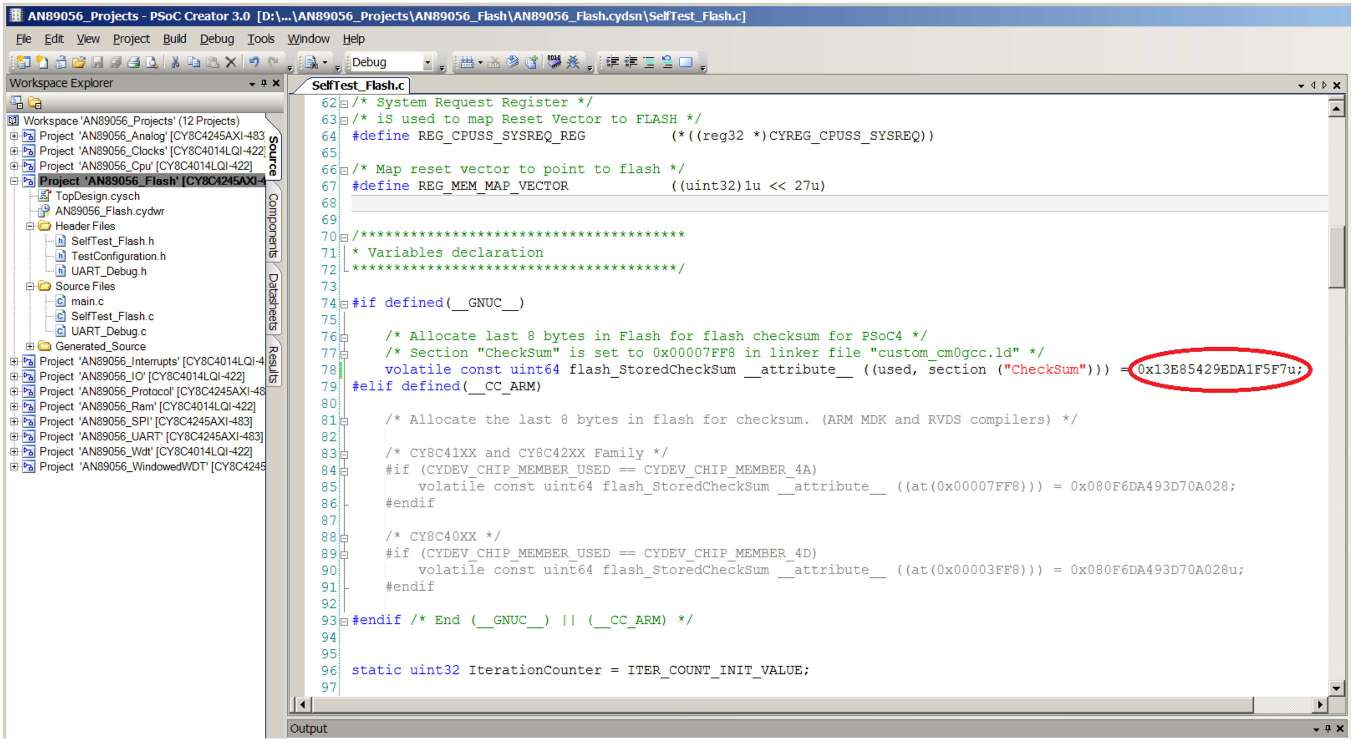
Terminal v1.9b - 20130116Я - by Br@y++

COM Port: COM27; Baud rate: [✓ 9600]; Data bits: [✓ 8]; Parity: [✓ none]; Stop bits: [✓ 1]; Handshaking: [✓ none]; custom BR: 9600; Rx Clear: -1; AutoScroll ✓; 13; Cnt = 18; [✓ ASCII]

```
<0><0><0><0><0><0><0><0><0><0><0><0>
Class B Library test
Test Counter = 0000
Test Counter = 0001
Test Counter = 0002
Test Counter = 0003
Test Counter = 0004
Test Counter = 0005
Test Counter = 0006
Test Counter = 0007
Test Counter = 0008
Test Counter = 0009
Test Counter = 000A
Test Counter = 000B
Test Counter = 000C
Test Counter = 000D
Test Counter = 000E
Error: FLASH
FLASH CHECKSUM: F7F5A1ED2954E813
```

3. Copy this checksum value and store it in the checksum location, *but remember that PSoC 4 uses little endian format*. The project for the GCC compiler is shown in Figure 19.

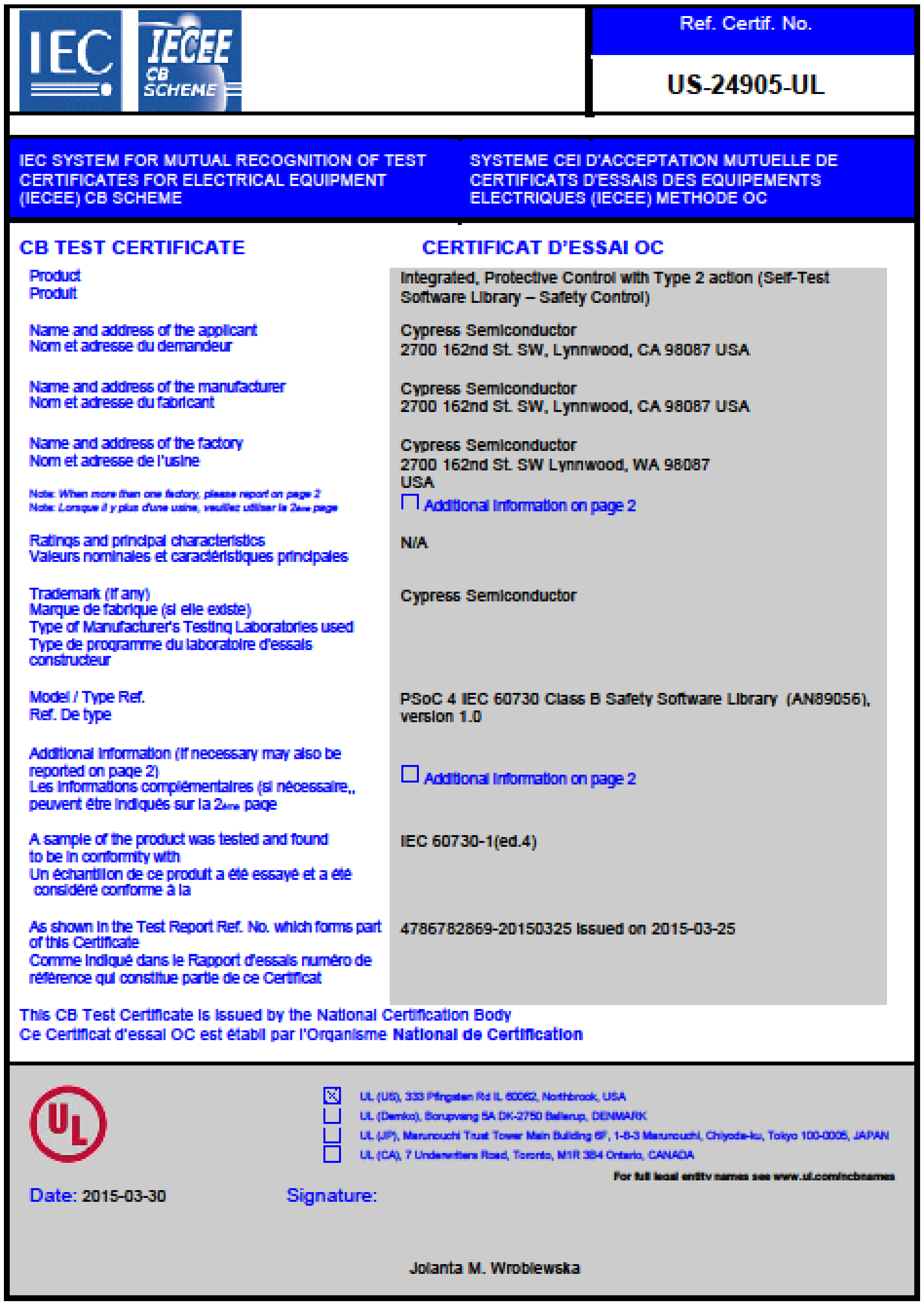
Figure 19. Reassign Checksum Constant with Actual Checksum


AN89056_Projects - PSoC Creator 3.0 [D:\...\AN89056_Projects\AN89056_Flash\AN89056_Flash.cydsn\SelfTest_Flash.c]
File Edit View Project Build Debug Tools Window Help
Debug
Workspace Explorer
Workspace 'AN89056_Projects' (12 Projects)
Project 'AN89056_Analog' [CY8C4245AXI-483
Project 'AN89056_Clocks' [CY8C4014LQI-422]
Project 'AN89056_Cpu' [CY8C4014LQI-422]
Project 'AN89056_Flash' [CY8C4245AXI-4
TopDesign.cysch
AN89056_Flash.cydwr
Header Files
SelfTest_Flash.h
TestConfiguration.h
UART_Debug.h
Source Files
main.c
SelfTest_Flash.c
UART_Debug.c
Generated_Source
Project 'AN89056_Interrupts' [CY8C4014LQI-4
Project 'AN89056_IO' [CY8C4014LQI-422]
Project 'AN89056_Protocol' [CY8C4245AXI-48
Project 'AN89056_Ram' [CY8C4014LQI-422]
Project 'AN89056_SPI' [CY8C4245AXI-483]
Project 'AN89056_UART' [CY8C4245AXI-483]
Project 'AN89056_Wdt' [CY8C4014LQI-422]
Project 'AN89056_WindowedWDT' [CY8C4245
Source
Components
Datasheets
Results
SelfTest_Flash.c
Output

```c
/* System Request Register */
/* iS used to map Reset Vector to FLASH */
#define REG_CPUSS_SYSREQ_REG            (*((reg32 *)CYREG_CPUSS_SYSREQ))

/* Map reset vector to point to flash */
#define REG_MEM_MAP_VECTOR              ((uint32)1u << 27u)


/****************************************
* Variables declaration
****************************************/

#if defined(__GNUC__)

    /* Allocate last 8 bytes in Flash for flash checksum for PSoC4 */
    /* Section "CheckSum" is set to 0x00007FF8 in linker file "custom_cm0gcc.ld" */
    volatile const uint64 flash_StoredCheckSum __attribute__ ((used, section ("CheckSum"))) = 0x13E85429EDA1F5F7u;
#elif defined(__CC_ARM)

    /* Allocate the last 8 bytes in flash for checksum. (ARM MDK and RVDS compilers) */

    /* CY8C41XX and CY8C42XX Family */
    #if (CYDEV_CHIP_MEMBER_USED == CYDEV_CHIP_MEMBER_4A)
        volatile const uint64 flash_StoredCheckSum __attribute__ ((at(0x00007FF8))) = 0x080F6DA493D70A028;
    #endif

    /* CY8C40XX */
    #if (CYDEV_CHIP_MEMBER_USED == CYDEV_CHIP_MEMBER_4D)
        volatile const uint64 flash_StoredCheckSum __attribute__ ((at(0x00003FF8))) = 0x080F6DA493D70A028u;
    #endif

#endif /* End (__GNUC__) || (__CC_ARM) */


static uint32 IterationCounter = ITER_COUNT_INIT_VALUE;

```

4. Compile the project and program PSoC.

B IECEE CB Scheme Test Certificate



Ref. Certif. No.

US-24905-UL

IEC SYSTEM FOR MUTUAL RECOGNITION OF TEST CERTIFICATES FOR ELECTRICAL EQUIPMENT (IECEE) CB SCHEME	SYSTEME CEI D'ACCEPTATION MUTUELLE DE CERTIFICATS D'ESSAIS DES EQUIPEMENTS ELECTRIQUES (IECEE) METHODE OC

CB TEST CERTIFICATE **CERTIFICAT D'ESSAI OC**

Product Produit	Integrated, Protective Control with Type 2 action (Self-Test Software Library – Safety Control)
Name and address of the applicant Nom et adresse du demandeur	Cypress Semiconductor 2700 162nd St. SW, Lynnwood, CA 98087 USA
Name and address of the manufacturer Nom et adresse du fabricant	Cypress Semiconductor 2700 162nd St. SW, Lynnwood, CA 98087 USA
Name and address of the factory Nom et adresse de l'usine Note: When more than one factory, please report on page 2 Note: Lorsque il y plus d'une usine, veuillez utiliser la 2ème page	Cypress Semiconductor 2700 162nd St. SW Lynnwood, WA 98087 USA ☐ Additional Information on page 2
Ratings and principal characteristics Valeurs nominales et caractéristiques principales	N/A
Trademark (if any) Marque de fabrique (si elle existe)	Cypress Semiconductor
Type of Manufacturer's Testing Laboratories used Type de programme du laboratoire d'essais constructeur	
Model / Type Ref. Ref. De type	PSoC 4 IEC 60730 Class B Safety Software Library (AN89056), version 1.0
Additional information (if necessary may also be reported on page 2) Les informations complémentaires (si nécessaire,, peuvent être indiqués sur la 2ème page	☐ Additional Information on page 2
A sample of the product was tested and found to be in conformity with Un échantillon de ce produit a été essayé et a été considéré conforme à la	IEC 60730-1(ed.4)
As shown in the Test Report Ref. No. which forms part of this Certificate Comme indiqué dans le Rapport d'essais numéro de référence qui constitue partie de ce Certificat	4786782869-20150325 issued on 2015-03-25

This CB Test Certificate is issued by the National Certification Body
Ce Certificat d'essal OC est établ par l'Organisme **National de Certification**

- ☒ UL (US), 333 Pfingsten Rd IL 60062, Northbrook, USA
- ☐ UL (Demko), Borupvang 5A DK-2750 Ballerup, DENMARK
- ☐ UL (JP), Marunouchi Trust Tower Main Building 6F, 1-8-3 Marunouchi, Chiyoda-ku, Tokyo 100-0005, JAPAN
- ☐ UL (CA), 7 Underwriters Road, Toronto, M1R 3B4 Ontario, CANADA

For full legal entity names see www.ul.com/ncbnames

Date: 2015-03-30 Signature:

Jolanta M. Wroblewska

1/1

C List of Supported Part Numbers

Note: The SPI, UART, and windowed WDT self-tests use UDB blocks and support only parts that are UDB equipped. PSoC 4 M-Series devices require PSoC Creator version 3.2 or higher which is not supported in the certified libraries.

PSoC 4	
CY8C4013LQI-411	CY8C4244LQI-443
CY8C4013SXI-400	CY8C4244LQQ-443
CY8C4013SXI-410	CY8C4244PVI-432
CY8C4013SXI-411	CY8C4244PVI-442
CY8C4014LQI-412	CY8C4244PVQ-432
CY8C4014LQI-421	CY8C4244PVQ-442
CY8C4014LQI-422	CY8C4245AXI-473
CY8C4014LQI-SLT1	CY8C4245AXI-483
CY8C4014LQI-SLT2	CY8C4245AXQ-473
CY8C4014SXI-411	CY8C4245AXQ-483
CY8C4014SXI-420	CY8C4245LQI-483
CY8C4014SXI-421	CY8C4245LQQ-483
CY8C4124AXI-433	CY8C4245PVI-482
CY8C4124AXQ-433	CY8C4245PVQ-482
CY8C4124LQI-433	CY8C4245AZI-M433 (Not supported by library projects)
CY8C4124LQQ-433	CY8C4245AZI-M443 (Not supported by library projects)
CY8C4124PVI-432	CY8C4245AZI-M445 (Not supported by library projects)
CY8C4124PVI-442	CY8C4245LTI-M445 (Not supported by library projects)
CY8C4124PVQ-432	CY8C4245AXI-M445 (Not supported by library projects)
CY8C4124PVQ-442	CY8C4246AZI-M443 (Not supported by library projects)
CY8C4125AXI-473	CY8C4246AZI-M445 (Not supported by library projects)
CY8C4125AXI-483	CY8C4246AZI-M475 (Not supported by library projects)
CY8C4125AXQ-473	CY8C4246LTI-M445 (Not supported by library projects)
CY8C4125AXQ-483	CY8C4246LTI-M475 (Not supported by library projects)
CY8C4125LQI-483	CY8C4246AXI-M445 (Not supported by library projects)
CY8C4125LQQ-483	CY8C4247LTI-M475 (Not supported by library projects)
CY8C4125PVI-482	CY8C4247AZI-M475 (Not supported by library projects)
CY8C4125PVQ-482	CY8C4247AZI-M485 (Not supported by library projects)
CY8C4244AXI-443	CY8C4247AXI-M485 (Not supported by library projects)
CY8C4244AXQ-443	

D MISRA Compliance

The tables 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 3. Verification Environment

Component	Name	Version
Test Specification	MISRA-C:2004 guidelines for the use of the C language in critical systems	October 2004
Target Device	PSoC 4	Production
	PSoC 4	Production
Target Compiler	PK51	9.03
	GCC	4.8.4
Generation Tool	PSoC Creator	3.1
MISRA Checking Tool	Programming Research QA C source code analyzer for Windows	8.1-R
	Programming Research QA C MISRA-C:2004 Compliance Module (M2CM)	3.2

Table 4. Deviated Rules

MISRA-C:2004 Rule	Rule Class (R/A)	Rule Description	Description of Deviation(s)
3.1	R	All use of implementation-defined behavior shall be documented.	For the documentation on PK51 and GCC compilers, refer to the PSoC Creator Help menu, Documentation submenu, and Keil and GCC commands, respectively.
8.7	R	Objects shall be defined at block scope if they are accessed only from within a single function.	Volatile global variables are accessed in ISR routines.
8.8	R	An external object or function shall be declared in one and only one file.	For PSoC 4, some objects are being declared with external linkage in **.c* files, and these declarations are not in header files.
8.10	R	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.	Library APIs are designed to be used in a user application and may not be used in a library API.
10.1	R	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a) it is not a conversion to a wider integer type of the same signedness, or b) the expression is complex, or c) the expression is not constant and is a function argument, or d) the expression is not constant and is a return expression.	File *SelfTest_Analog.c*: The uint16 `DAC_Offset` variable is used in arithmetic.
11.3	A	A cast should not be performed between a pointer type and an integral type.	See Table 5.
11.4	A	A cast should not be performed between a pointer to object type and a different pointer to object type.	See Table 5.
11.5	R	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.	See Table 5.
13.6	R	Numeric variables being used within a “for” loop for iteration counting shall not be modified in the body of the loop.	File *SelfTest_UDB_CfgReg.c*, function `SelfTests_UDB_ConfigReg`: The iblock variable is zeroed within the loop body to start the test for the next flash block.

MISRA-C:2004 Rule	Rule Class (R/A)	Rule Description	Description of Deviation(s)
13.7	R	Boolean operations whose results are invariant shall not be permitted.	Some Boolean operations are mistakenly treated by the analyzer as "invariant." These are all false positives. Library APIs are designed to be used in the user application and may not be used in library demo code.
14.1	R	There shall be no unreachable code.	Library APIs are designed to be used in user applications and may not be used in library code.
14.3	R	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.	The CyGlobalIntEnable macro has a null statement that is located close to other code.
15.2	R	An unconditional break statement shall terminate every non-empty switch clause.	File *SelfTest_Clock.c*, function `SelfTests_Clock`: State machine implementation requires switch case use without a break statement.
16.8	R	All exit paths from a function with non-void return type shall have an explicit "return" statement with an expression.	File *SelfTest_CPU_Regs.c*, function `SelfTest_CPU_Regs:` Returns the test result via ASM instruction.
16.9	R	A function identifier shall be used only with either a proceding "&" or with a parenthesized parameter list, which may be empty.	Files *uart_master_message.c* and *uart_slave_message.c:* Functions `UTX_M_ISR_SetVector,` `URX_M_ISR_SetVector,` `U_M_TIME_ISR_SetVector,` `UTX_S_ISR_SetVector,` and `URX_S_ISR_SetVector` take functions names as input parameters.
16.10	R	If a function returns error information, then that error information shall be tested.	Library functions return values that are not used in demo projects but can be used in customer projects.
17.4	R	Array indexing shall be the only allowed form of pointer arithmetic.	See Table 5.
21.1	R	Minimization of run-time failures shall be ensured by the use of at least one of the following: a) static analysis tools/techniques b) dynamic analysis tools/techniques c) explicit coding of checks to handle run-time faults	Some code generated by PSoC Creator in some specific configurations can contain redundant operations introduced because of a generalized implementation approach.

Table 5. Pointer Violations in Demo Projects

Pointer (variable name)	MISRA Rule	Files	Description
All register definitions in **.h* files that are used in library APIs	11.3 A cast should not be performed between a pointer type and an integral type.	**AN78175_Memory project:** *Main.c, SelfTest_cpu_asm.c, SelfTest_crc_calc.c, SelfTest_CustomFlash.c, SelfTest_EEPROM.c, SelfTest_Ram.c, SelfTest_Flash.c, SelfTest_Stack.c, SelfTest_UDB_CfgReg.c* **AN78175_Analog project:** *Main.c, idac8_cb.c, elfTest_Analog.c, SelfTest_Analog_Calibration.c, vdac8_cb.c* **AN78175_Digital project**: *Main.c, SelfTest_IO.c* **AN78175_Protocol project:** *Main.c, uart_master_message.c, Uart_slave_message.c* **AN78175_Wdt project:** *Main.c* **WDT_Window project:** *Main.c*	Hardware register access is implemented over pointers to these registers.
***SelfTest_Flash.c*:** Flash_Pointer ***Main.c*:** rxd	11.4 A cast should not be performed between a pointer to object type and a different pointer to object type.	**AN78175_Memory project:** *SelfTest_Flash.c* **AN78175_Protocol project:** *main.c*	Cast **static uint8 CYCODE *Flash_Pointer** to **(*((uint16 code *)Flash_Pointer))** to access uint16 word per one instruction. Cast **uint8 rxd[16u]** to **(char8*) rxd)** as it is required by **LCD_Char_PrintString** library function.
***uart_slave_message.c*:** UMS.message;	11.5 A cast shall not be performed that removes any “const” or “volatile” qualification from the type addressed by a pointer.	**AN78175_Protocol project:** *uart_slave_message.c*	Cast **uint8 message[MAX_MESSAGE_SIZE]** to **(uint8 *)UMS.message.**
SelfTest_CRC_calc.c: RegPointer ***SelfTest_CustomFlash.c*:** rowData ***SelfTest_EEPROM.c*:** RegPointer, RegPointer1 ***SelfTest_Flash.c*:** Flash_Pointer ***SelfTest_Stack.c*:** stack ***SelfTest_UDB_CfgReg.c*:** CfgRegPointer ***uart_master_message.c*:** UM.rxptr ***uart_slave_message.c*:** UMS.txptr	17.4 Array indexing shall be the only allowed form of pointer arithmetic.	**AN78175_Memory project:** *SelfTest_CRC_calc.c, SelfTest_CustomFlash.c, SelfTest_EEPROM.c, SelfTest_Flash.c, SelfTest_Stack.c, SelfTest_UDB_CfgReg.c* **AN78175_Protocol project:** *uart_master_message.c, uart_slave_message.c*	**uint8 * rowData** is passed as a parameter to the function where it is accessed as an indexed array. **reg8 * RegPointer** and **reg8 * RegPointer1** are used as indexed arrays to access the registers set. **static uint8 CYCODE *Flash_Pointer** is used as an array of flash data and is indexed via a “++” operation. **uint16 *stack** is used as an array and is indexed via a “++” operation. **uint8 CYCODE *** CfgRegPointer is used as an indexed array to access data stored in flash. **uint8 * txptr** is used as an array and is indexed via a “++” operation.

Document History

Document Title: AN89056 - PSoC® 4 – IEC 60730 Class B and IEC 61508 SIL Safety Software Library

Document Number: 001-89056

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	4697425	GJV	03/23/2015	New application note
*A	5276478	GJV	05/18/2016	Updated included projects to included missing assembly and linker files. Added IECEE test certificate Updated template.
*B	5698503	AESATP12	04/26/2017	Updated logo and copyright.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709