AN78175

PSoC® 3 and PSoC 5LP - IEC 60730 Class B Safety Software Library

Author: Taras Kuzo, Dmytro Makara, Vasyl Parovinchak
Associated Project: Yes
Associated Part Family: CY8C3xxx, CY8C5xxx
Software Version: PSoC® Creator™ 2.2 SP1
Related Application Notes: AN79973

If you have a question, or need help with this application note, contact the authors at tark@cypress.com, dmtr@cypress.com, vasy@cypress.com.

AN78175 describes the PSoC® 3 and PSoC 5LP IEC60730 Class B Safety Software Library and includes an example project with self-check routines to ensure reliable and safe operation. Library routines and examples in the example project can be directly integrated with the end user's application. This application note also describes the API functions that are available in the Library.

Contents

Introduction 2
Overview of Annex H 2
Class B Requirements 3
API Functions for PSoC 3 and PSoC 5LP 4
CPU Registers Test 4
Program Counter 5
Program Flow Testing 5
Interrupt Handling and Execution 6
Clock 7
FLASH (Invariable Memory) 9
Error Correction Code (ECC) Method 9
Checksum Method 10
EEPROM (Invariable Memory) 11
SRAM (Variable Memory) 12
March Test 13
March C Test 13
March C Minus Test 13
MARCH C MINUS ALGORITHM: 13
March B Test 14
Stack Overflow Test 15
Digital I/O 16
A/D Converter 16
D/A Converter 18
Comparator 19
PGA 19
TIA 20
Opamp 21
Communications UART 23
Communications SPI 23
Recommendations and Examples 24
UDB Configuration Registers Test 24
Start-up Configuration Registers Test 26
Watchdog Test 27
Windowed Watchdog Timer 27
Example of Communications UART Data Transfer Protocol 30
Data Delivery Controlling 30
PSoC Implementation 31
Summary 34
References 34
Appendix A 35
Appendix B 37
Appendix C 38
Appendix D 39
MISRA Compliance 39
Worldwide Sales and Design Support 43

Introduction

Today, the majority of automatic electronic controls for home appliance products use single-chip microcontrollers. Manufacturers develop real-time embedded firmware that executes in the MCU and provides hidden intelligence to control home appliances. MCU damage due to overheating, static discharge, extremely high voltage, or other factors can cause the appliance to enter an unknown or unsafe state.

The International Electrotechnical Commission (IEC) has developed safety standard IEC 60730-1 that discusses mechanical, electrical, electronic, environmental endurance, EMC, and abnormal operation for home appliances.

This application note focuses on Annex H Class B: Requirements for Electronic Controls. This portion of the standard details test and diagnostic methods to ensure safe operation of embedded control hardware and software for home appliances.

Overview of Annex H

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

- Class A Control functions not intended to be relied upon for the safety of the equipment Examples: Humidity controls, lighting controls, timers, and switches.
- Class B Control functions intended to prevent unsafe operation of controlled equipment. Examples: Thermal cut-offs and door locks for laundry equipment.
- Class C Control functions intended to prevent special hazards (such as an explosion caused by the controlled equipment). Examples: Automatic burner controls and thermal cutouts for closed, unvented water heater systems.

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

The Class B Safety Software Library and the example project described in this application note implement the self-test and self-diagnostic methods that are 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:

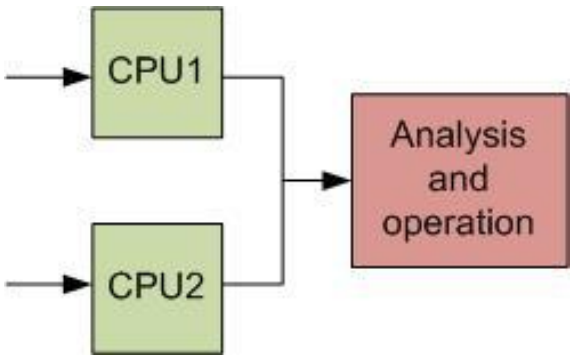
- Single channel with functional test
- Single channel with periodic self-test
- Dual channel without comparison (see Figure 1)

In the single-channel structure with 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 within the software, and the self-test occurs periodically while the software is in execution mode. The CPU is expected to regularly check the critical functions of the electronic control 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 the following figure.

Figure 1. Dual Channel without Comparison Structure



The dual-channel structure implementation is costlier because two CPUs (or two MCUs) are required. In addition, they are more complex because two devices are needed to regularly communicate with each other. The single-channel structure with a functional test is most commonly implemented today. However, appliance manufacturers are moving to the single-channel structure with the periodic self-test implementation.

Class B Requirements

The IEC60730-1 Class B Annex H Table H.11.12.7 has a list of the components that must be tested, depending on the software classification. Generally, each component offers optional measures to verify/test the corresponding components, and provides flexibility for manufacturers.

To provide Class B IEC 60730 compliance for the single-channel structures, manufacturers of electronic controls are required to test the components listed in the following table.

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

Class B IEC 60730 Components Required to be Tested on Electronic Control (see Table H.11.12.7 in Annex H)	Fault/Error
1.1 CPU registers	Stuck at
1.3 CPU program counter	Stuck at
2. Interrupt handling and execution	No interrupt or too frequent interrupt
3. Clock	Wrong frequency
4.1 Invariable memory	All single bit faults
4.2 Variable memory	DC fault
4.3 Addressing (relevant to variable/invariable memory)	Stuck at
5.1 Internal data path Data	Stuck at
5.2 Internal data path Addressing(for expanded memory MCU systems only)	Wrong address
6.1 External communications Data	Hamming distance 3
6.2 External communications Addressing	Hamming distance 3
6.3 Timing	Wrong point in time/sequence
7.1 I/O Periphery	Fault conditions specified in Appendix B: “IEC 60730-1, H.27”)
7.2.1 Analog A/D and D/A converters	Fault conditions specified in Appendix B: “IEC 60730-1, H.27”)
7.2.2 Analog multiplexor	Wrong addressing

In addition, the following system requirements are recommended to run the Class B Safety Software Library:

- The user application must correctly 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-tests routine, an unexpected change may occur in any of the registers. Therefore, when the Interrupt Service Routine (ISR) is executed, the contents of the register do not match the expected.
- The example project with Class B Safety Software Library shows where interrupts need to be disabled and enabled for correct self-testing.

Class B Safety Software Library

The Class B Safety Software Library described in this application note can be used with PSoC 3 and PSoC 5LP devices. The library includes APIs designed to maximize the application reliability through fault detection.

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

There are two types of self-test functions described and implemented in this application note:

1. Self-test functions to help meet the IEC 60730-1 Class B standard compliance :

- CPU Registers - Test for stuck bits
- Program counter - Test for jumps to the correct address
- Interrupt handling and execution - Test for proper interrupt calling and periodicity
- Clock. Test for wrong frequency
- FLASH (invariable memory). Test for memory corruption.
- EEPROM (invariable memory) - Test for memory corruption
- SRAM (variable memory) - Test for stuck bits and proper memory addressing
- Digital I/O - Test for pins short
- A/D- and D/A- convertor - Test for proper functionality
- COMPARATOR - Test for proper functionality
- Communications (UART, SPI) - Test for correct data reception

2. Additional self-test functions, which PSoC 3 and PSoC 5LP can support due to programmable interconnect. Often, the end-application also needs these self-tests even though they are not provided in Appendix B IEC 60730-1.

- TIA, PGA, and opamp tests
- Watchdog test - Test for chip reset
- Additional Windowed WDT to monitor FW execution
- UDB configuration registers test
- Start-up configuration registers test

All self-tests can be executed after device startup before the main loop. This provides an opportunity to check whether the chip is suitable for operation.

In addition, self-tests must be executed while the device is working in a determined period of time. This provides an opportunity to make sure the chip was not damaged during operation.

The following sections describe self-test and implementation details for each test according to the IEC 60730-1 Class B Annex H. In addition, each section lists the APIs required to execute a corresponding test for the supported architectures.

API Functions for PSoC 3 and PSoC 5LP

CPU Registers Test

PSoC 3 with an enhanced 8051 core has 256 bytes of internal data RAM as shown in Figure 2.

Figure 2. 8051 Internal Data Space

<table>
<tr><td>0xFF

0x80</td><td>RAM Shared with Stack Space (indirect addressing, idata space)</td><td>SFRs Special Function Registers (direct addressing, data space)</td></tr>
<tr><td>0x7F

0x30</td><td colspan="2">RAM Shared with Stack Space (direct and indirect addressing, shared idata and data spaces)</td></tr>
<tr><td>0x2F
0x20</td><td colspan="2">Bit Addressable Area</td></tr>
<tr><td>0x0F
0x00</td><td colspan="2">4 Banks, R0-R7 Each</td></tr>
</table>

All the registers and bit addressable locations are as follows (a failure of any of them can cause unpredictable behavior):

- Accumulator registers, ACC or A and B
- Stack pointer, SP
- Program counter, PC
- Four register banks (R0 – R7)
- Program status word, PSW
- Flag bit addressable locations
- 16 bit data pointer registers, DPTR0 and DPTR1
- Other SFRs including port data and select registers

PSoC 5LP with the Cortex-M3 has 16-bit and 32-bit registers:

- R0 to R12 – general purpose registers.
- R0 to R7 – can be accessed by all instructions.
- R8 to R12 – can be accessed by all 32-bit and some 16-bit instructions.
- 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. Stores the return program counter during function calls.
- R15 – Program counter. This register can be written to control the program flow.

The CPU Register test implements the functional test H.2.16.5 defined by the IEC 60730-1 standard. It detects stuck-at faults in the CPU registers by using the checkerboard test. This ensures that the bits in the registers are not stuck at value ‘0’ or ‘1’; this is a non-destructive test. This test 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 (length is dependent upon architecture) 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
SelfTest_CPU_asm.c
SelfTest_CPU_asm.h
```

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

For PSoC 3, the function checks the following CPU registers: ACC, DPL, DPH, DPL1, DPH1, DPS, DPX, DPX1, P2, IE, PSW, and B. It checks the entire internal data RAM by looping through the 256 bytes of memory.

For PSoC 5LP, the function checks all 16 CPU registers.

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

Program Counter

- DOC_PC – PSoC 3 CPU Program Counter register.
- R15 – PSoC 5LP CPU Program Counter register. Because Bit 0 is always '0', the instructions are always aligned to the word or half-word boundaries.

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

The Program Counter (PC) test implements the functional test H.2.16.5 defined by 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 PC test calls the functions that are located in the Flash memory at different addresses.
   - For PSoC 3, this can be done by using Linker commands. The Linker command for placing the functions in the above mentioned locations is: SEGMENTS(?PR?SelfTest_PC5555?SelfTest_CPU(C:0x5555),?PR?SelfTest_PCAAAA?SelfTest_CPU(C:0xAAAA))
   - The above linker command places a function named SelfTest_PC5555() placed in a file named SelfTest_CPU.c into location 0x5555 and SelfTest_PCAAAA() placed in a file named SelfTest_CPU.c into location 0xAAAA.
   - This linker command is placed in the following location: **Project** > **Build Setting** > **Linker** > **Command line**.
   - For PSoC 5LP, this can be done by using the Linker script in a **.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_cm3gcc.ld" file. This linker file is added in the following location: **Project** > **Build Setting** > **Linker** > **Custom Linker Script**.
2. These 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 will trigger 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 The Linker command for placing the functions in the previously mentioned locations for PSoC 3 must be added. SEGMENTS(?PR?SelfTest_PC5555?SelfTest_CPU(C:0x5555),?PR?SelfTest_PCAAAA?SelfTest_CPU(C:0xAAAA))

For PSoC 5LP, the "custom_cm3gcc.ld" file must be added to the linker.

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

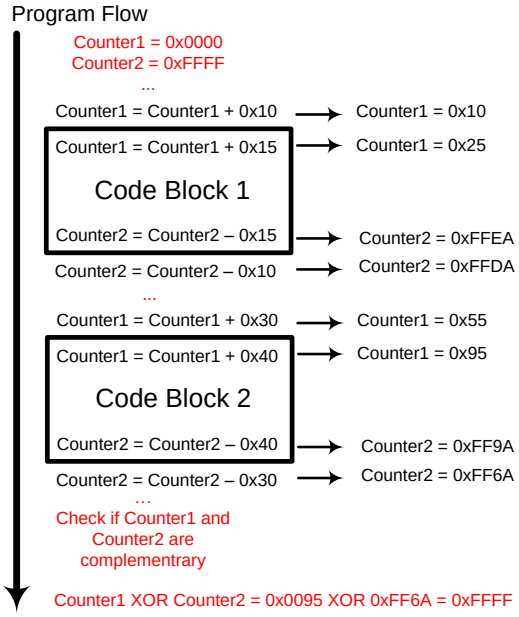
Program Flow Testing

A specific method is used to check the program execution flow for every critical execution code block. Unique numbers are added to or subtracted from the complementary counters before block execution, and immediately after execution. These procedures check if the code block is called correctly from the main program flow and 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 3.

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

Figure 3. Program Flow Test



Interrupt Handling and Execution

The PSoC 3 and PSoC 5LP Interrupt Controllers provide the mechanism for hardware resources to change the program address to a new location independent of the current execution in the main code. The interrupt controller also handles continuation of the interrupted code after completion of the interrupt service routine.

The Interrupt test implements independent time slot monitoring H.2.18.10.4 defined by 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 two interrupt sources. The test assumes that other interrupt vectors are also working when these two tested vectors are working.

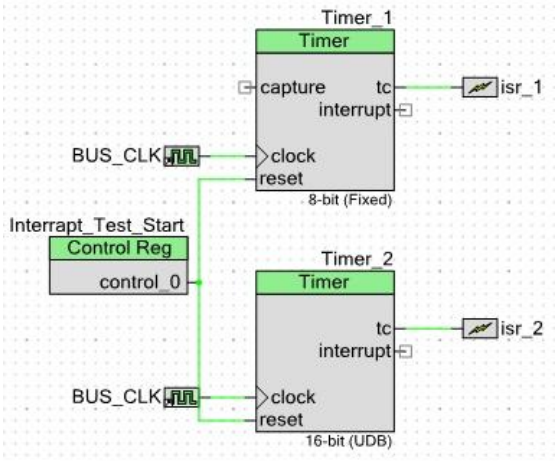
Function

```
uint8  SelfTest_Interrupt(void)

Returns:  0   No error
          1   Error detected

Located in:   SelfTest_Interrupt.c
              SelfTest_Interrupt.h
```

Figure 4. PSoC Creator Schematic for Interrupt Test



Note The component's name should be the same as in Figure 4. Global interrupts must be enabled for this test, but all interrupts except isr_1 and isr_2 must be disabled.

The SelfTest_Interrupt() function is called to check the Interrupt Controller operation. The interrupt sources are from Fixed Function and UDB blocks.

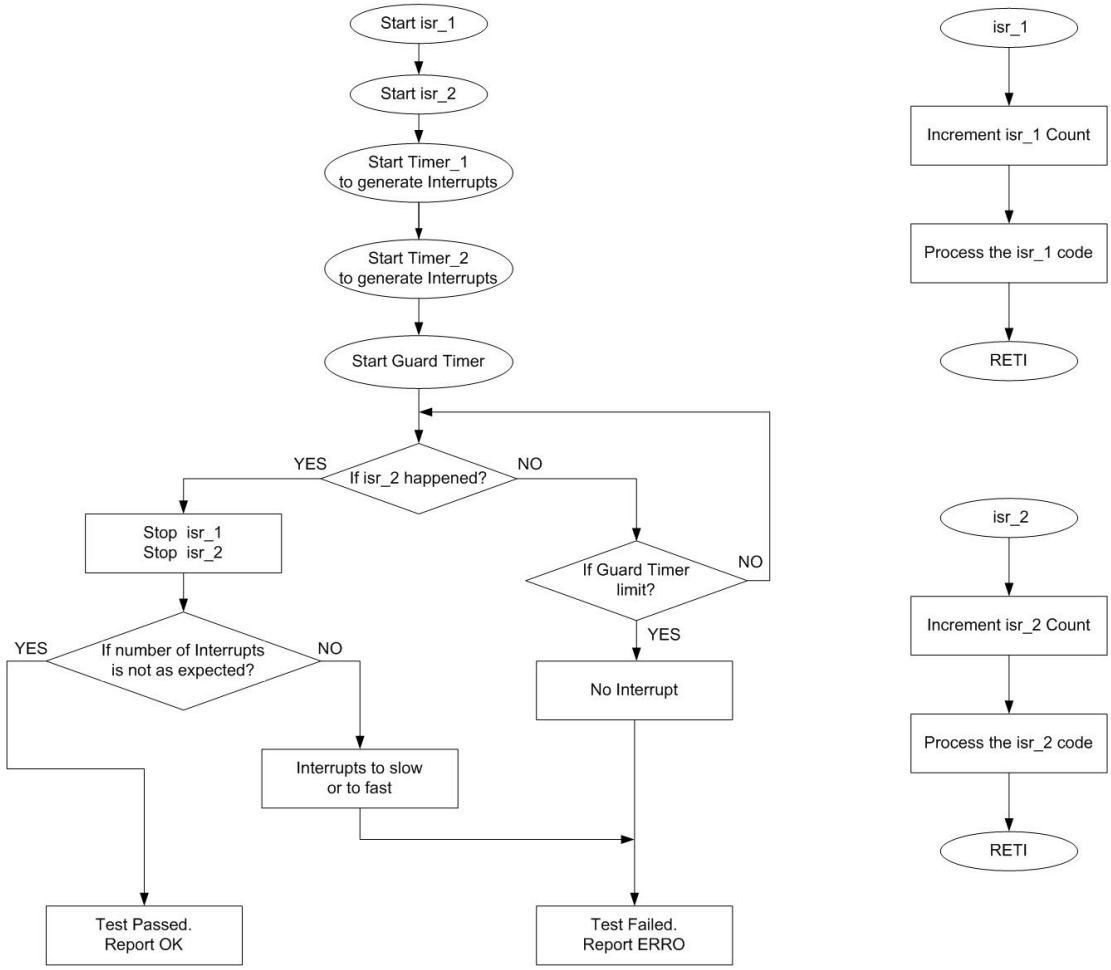
Calling of the function starts two timers.

Timer_1 is placed into a Fixed Function block and configured to generate an interrupt every 10.667 µs. isr_1 counts the number of interrupts that occurred.

Timer_2 is placed into a UDB block and configured to generate an interrupt every 213.333 µs.

Timer_1 must generate 20 interrupts during 213.333 µs When Timer_2 interrupt occurs, the isr_2 component stops Timer_1 interrupt counting. If the counted value in isr_1 is ≥ 18 and ≤ 22, the test is passed. The specific number of interrupts needed to pass this test depends on the application, and can be modified as required.

Figure 5. Flowchart for Interrupt Self Test



Clock

According to the IEC 60730 standard, only harmonics and sub-harmonics of the clock need to be tested. The Clock test implements an independent time slot, monitoring H.2.18.10.4 defined by the IEC 60730 standard. It verifies the reliability of the system clock (specifically, that the system clock should be neither too fast nor too slow).

Function

```
uint8  SelfTest_Clock(void)

Returns:  0   No error
          Not 0  Error detected (return the
          id of failed clock)

Located in:   SelfTest_Clock.h
              SelfTest_Clock.c
```

Note The tested clock sources and there accuracies are defined in the SelfTest_Clock.h. If the clock frequency is not defined, the test of the corresponding source will be skipped.

```
//      Id = 1
#define MASTER_CLK_FREQUENCY 24000000
       // Frequency in Hz
#define MASTER_CLK_ACCURACY_Minus 1000
       // Accuracy - in 0.001%
#define MASTER_CLK_ACCURACY_Plus 1000
       // Accuracy + in 0.001%
//      Id = 2
#define IMO_CLK_FREQUENCY 3000000
#define IMO_CLK_ACCURACY_Minus 1000
#define IMO_CLK_ACCURACY_Plus 1000
//      Id = 3
#define XTALM_CLK_FREQUENCY 24000000
#define XTALM_CLK_ACCURACY_Minus 1000
#define XTALM_CLK_ACCURACY_Plus 1000
```

```
//     Id = 4
#define ILO_CLK_FREQUENCY 1000
#define ILO_CLK_ACCURACY_Minus 50000
#define ILO_CLK_ACCURACY_Plus 100000
//     Id = 5
#define PLL_CLK_FREQUENCY 24000000
#define PLL_CLK_ACCURACY_Minus 1000
#define PLL_CLK_ACCURACY_Plus 1000
//     Id = 6
#define XTALK_CLK_FREQUENCY 32768
#define XTALK_CLK_ACCURACY_Minus 10
#define XTALK_CLK_ACCURACY_Plus 10
//     Id = 7
#define DSIG_CLK_FREQUENCY 10000000
#define DSIG_CLK_ACCURACY_Minus 100
#define DSIG_CLK_ACCURACY_Plus 100
//     Id = 8
#define DSID_CLK_FREQUENCY 10000000
#define DSID_CLK_ACCURACY_Minus 100
#define DSID_CLK_ACCURACY_Plus 100
//     Id = 9
#define DSIA_CLK_FREQUENCY 10000000
#define DSIA_CLK_ACCURACY_Minus 100
#define DSIA_CLK_ACCURACY_Plus 100
```

Figure 6 shows the PSoC schematic implementation of this test. The Clock input is connected to the measurement frequency that is tested through a divider to the count input of the UDB Counter. The counter counts the rising edges of the divided input frequency during a 1-sec time interval; after this time, the counter stores the counted value in the capture register. The divided frequency of the XTAL 32.768 kHz is used to provide the reference time interval for the counter.

This function returns the result after frequency measurement during the 1-second interval. The *SelfTests_Clock_Ready()* function checks when the hardware part finishes frequency measuring and the results are ready for comparison with the nominal value. It is not necessary to wait while the hardware part of the clock self-test operates. During this time, other tasks can be performed and the result of the clock test can be retrieved later.

Figure 6. Clock Test PSoC Implementation

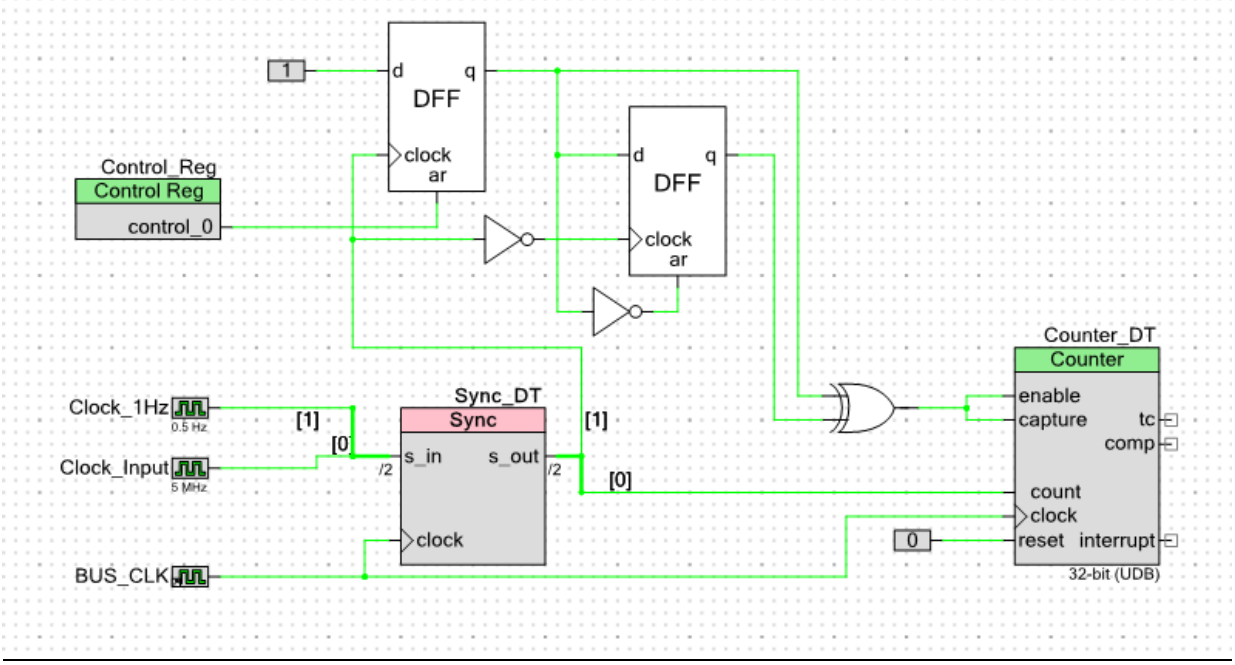
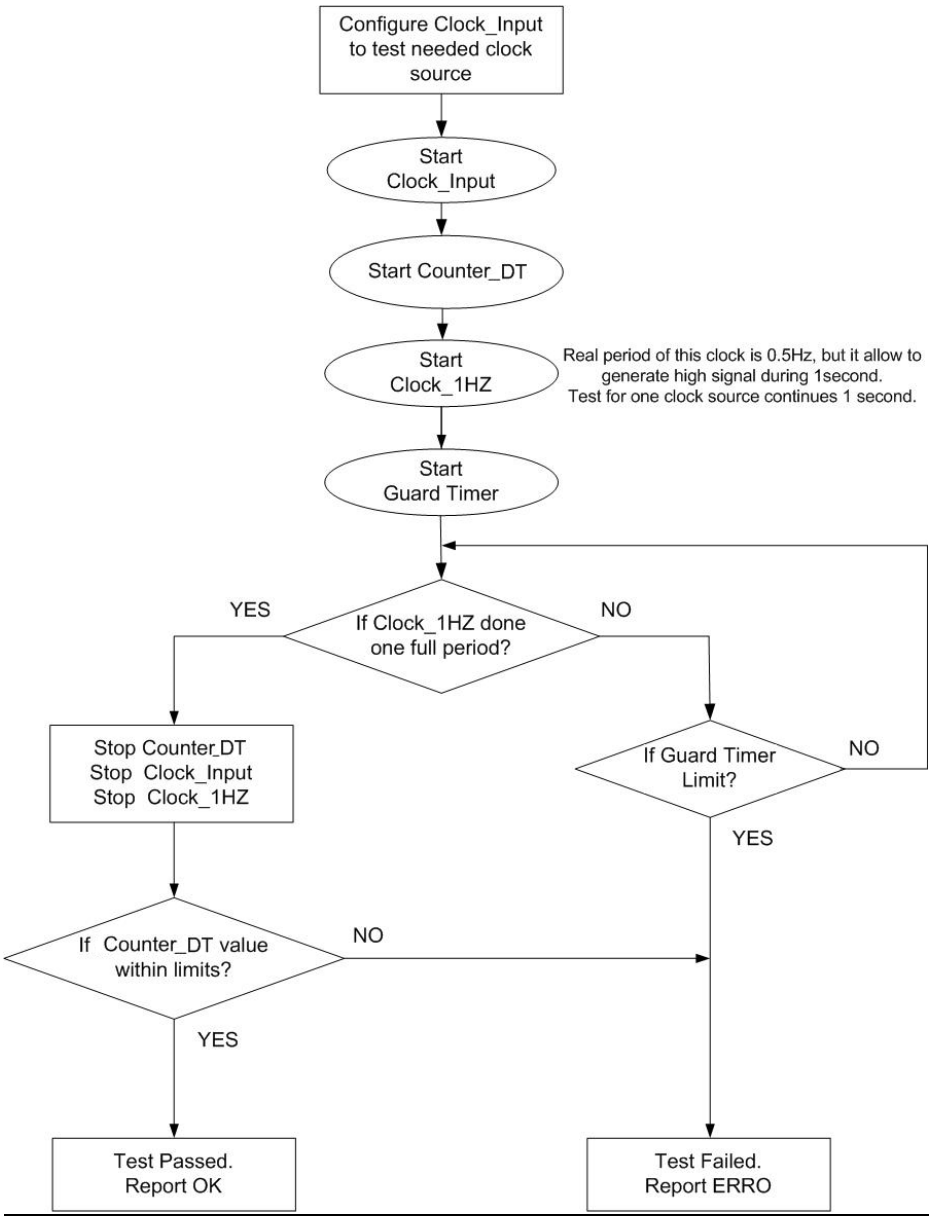


Figure 7. Flowchart for Clock Self-Test



FLASH (Invariable Memory)

PSoC 3 and PSoC 5LP devices include on-chip flash memory. These two families offer devices that range from 16 to 256 kilobytes. Additional flash is available for either error correction or data storage.

The PSoC 3 and PSoC 5LP flash memory is organized in rows where each row contains 256 data bytes plus 32 bytes. The additional 32 bytes per row can be configured for either error correcting codes (ECC) or additional data storage.

Error Correction Code (ECC) Method

To use the ECC method, you must enable ECC in the project settings by checking the box **System** > **Configuration** > **Enable Error Correcting Code (ECC)** in the projects **.cydwr* file. The ECC block checks the data read from flash. It is responsible for error detection and correction. The cache receives the error status from the ECC block and the error status is logged into the firmware visible registers. The ECC method works even when the flash memory contents are dynamically changed at run time, such as during bootloading.

There are two interrupts related to ECC detection and correction. The interrupts can be enabled by setting the appropriate bits of the CACHE_INT_MSK register.

- ECC-Single Bit: This interrupt occurs when a single bit error is encountered during read operation and is fixed. The CACHE_INT_LOG3 register is updated with the flash location address where the error occurred.
- ECC-Multiple Bits: This interrupt occurs when a multiple-bit error is encountered during a read operation. This error cannot be fixed. The CACHE_INT_LOG4 register is updated with the flash location address where the error occurred.

Function

```
uint8  SelfTest_FlashECC(void)

Returns:  0   No error
          1   Error detected(2 or more bits
              damaged)
          2   1-bit was corrected

Located in:   SelfTest_Flash.c
              SelfTest_Flash.h
```

The function SelfTest_FlashECC() is called to do the flash memory corruption test using the ECC hardware (Error Correction Code).

Note The global Interrupts must be enabled for this test.

Checksum Method

All devices can also use the checksum method if the ECC area is used for data rather than ECC.

To complete a full flash diagnostic, a checksum of all used flash needs to be calculated. PSOC_FLASH_SIZE in *SelfTest_Flash.h* file defines the flash size that needs to be monitored.

The Checksum Flash test reads each ROM or flash location and accumulates the values in a 16-bit variable to calculate a running checksum of the entire flash. The actual 16-bit checksum of the flash is stored in the last two bytes of the flash itself. When the test reaches the location 0xFFFE on 64-KB devices (the penultimate byte of the flash), it stops the checksum calculation and then compares this calculated value with the actual value stored in the last 2 bytes of the flash. A mismatch indicates ROM failure and the execution is frozen.

Programming Steps

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

   For PSoC 3:

   ```
   #pragma asm
        CSEG AT 0xFFFE
        DW   0x0000
   #pragma endasm
   ```

   For PSoC 5LPs (GCC compiler):

   ```
   const      uint16      CheckSum_from_rom
   __attribute__ ((section ("CheckSum"))) =
   0x0000u;
   ```

   For PSoC 5LPs (ARM MDK or RVDS compilers):

   ```
   const      uint32      CheckSum_from_rom
   __attribute__    ((at(0x0003FFFC)))    =
   0x00000000u;
   ```

2. Open PSoC Programmer and read Flashchecksum.
3. Copy this value and store it in the checksum location
4. Compile the project and program PSoC.

See Appendix A for a detailed description.

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 do 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 the parameters in SelfTest_Flash.h file:

```
/*Set size of one block in Flash test*/
#define WORDS_IN_BLOCK_FLASH (1024u)
```

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.

Note The check does not work if there is a change in the flash or ROM during run time using SPC commands. The checksum needs to be updated before calling the test.

EEPROM (Invariable Memory)

The PSoC 3 and PSoC 5LP devices have on-chip EEPROM memory. These two families offer devices that range from 512 bytes to 2 kilobytes.

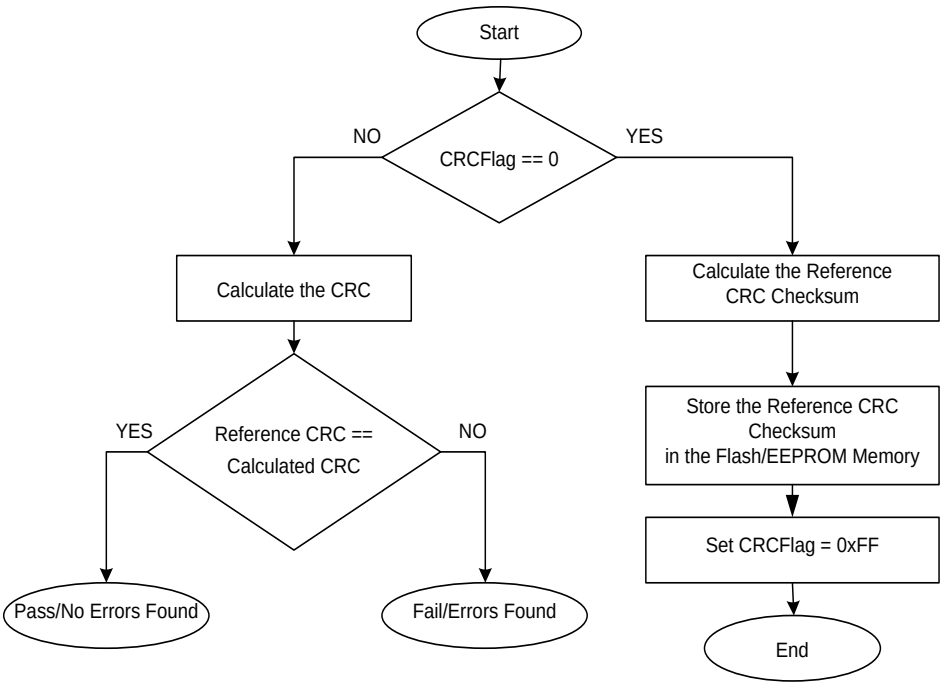
The PSoC 3 and PSoC 5LP EEPROM memories have the following organization:

- Organized in rows, where each row contains 16 bytes.
- Organized as one block of 32, 64, or 128 rows, depending on the device.
- Store nonvolatile data.
- Write and erase using the SPC commands.
- Byte read access by the CPU or DMA using the PHUB.

The EEPROM (Invariable Memory) test implements the periodic modified checksum H.2.19.3.1 defined by the IEC 60730 standard. It detects single-bit faults in the invariable memory. The invariable memory in a system such as the EEPROM contains data not intended to vary during program execution. The EEPROM invariable memory test computes the periodic checksum using a Cyclic Redundancy Check (CRC). This example uses the CRC-16 standard for the CRC calculation.

The characteristics of the CRC divisor vary from 8 to 32 bits depending on the polynomial that is used. You can change the polynomial in the CRC component configuration parameters.

Figure 8. Flowchart for Invariable Memory Test



The CRC16 calculation function returns the final CRC value that can be used to perform the following:

1. At system startup, the computed CRC checksum can be used as a reference checksum if the CRC_Flag is set to 0x00.
2. The reference checksum is stored in the Flash or EEPROM memory and the CRC flag is set to 0xFF.
3. The CRC16 calculation function can be called periodically if the CRC flag is set to 0xFF.
4. The checksum calculated from step 3 is compared to the reference checksum.
5. If both values match, a status bit can be set by the user application to indicate that the invariable memory has passed the test and no errors were found.

Function

```
uint8  SelfTest_EEPROM(void)

Returns:  0   No error
          1   Error detected

Located in:   SelfTest_Memory.c
              SelfTest_Memory.h
```

Schematic showed in Appendix B.

The function SelfTest_EEPROM () is called to do the EEPROM memory corruption test using CRC16 calculation.

There are two modes of CRC16 calculation: firmware calculation or hardware calculation.

You can define it using the following parameters in the `SelfTest_Memory.h` file:

```
#define CRC_CALCULATION_MODE
CRC_SOFTWARE_MODE

#define CRC_HARDWARE_MODE (1u)

#define CRC_SOFTWARE_MODE (0u)
```

Appendix B shows the implemented schematic for the EEPROM tests. The EEPROM CRC performs calculations using the hardware schematic and the CRC hardware component. Data from the EEPROM transfers to the Shift Register through the DMA. The Shift Register is used because the CRC component computes the CRC from a serial bit stream. Four D Flip Flops and Counter7 (based on the Status Register) are used for synchronization.

Note The component names should be the same as in Appendix B.

The example project uses the last row in the EEPROM to store the First Calculation Status Byte and two CRC bytes. CRC_EEPROM_SEMAPHORE_SHIFT, CRC_EEPROM_LO_SHIFT and CRC_EEPROM_HI_SHIFT define shift from end or row where this three bytes store.

These bytes are not used in the EEPROM CRC calculation.

`ClearEEPROMUserArea()` – this function clears CRC_CALC_STATUS, CRC_LO and CRC_HI defined bytes in EEPROM.

`SelfTest_EEPROM()` – Calculate the CRC if CRC_CALC_STATUS is not sets, store it to the EEPROM, and sets CRC_CALC_STATUS.

These two functions must be called after PSoC start and before the first test result from the `SelfTest_EEPROM()` function. In this case, if you need to update EEPROM periodically, follow these steps:

- Change data in EEPROM
- Call ClearEEPROMUserArea()function to clear CRC_CALC_STATUS status.
- Call SelfTest_EEPROM()function once to calculate and set CRC value and set CRC_CALC_STATUS.
- Call SelfTest_EEPROM()function in required place to do EEPOM self-test.

SRAM (Variable Memory)

PSoC 3 and PSoC 5LP devices include on-chip SRAM. These families offer devices that range from 2 to 64 kilobytes. The PSoC 3 devices offer an additional 4 kilobytes as a trace buffer that can be used as general purpose SRAM.

PSoC 3 and PSoC 5LP SRAM have these features:

- Organized as up to three blocks of 4 KB each, including a 4 KB trace buffer, for the CY8C38 family.
- Organized as up to 16 blocks of 4 KB each, for the CY8C55 family.
- Code can be executed out of portions of the SRAM, for the CY8C55 family.
- 8-bit, 16-bit, or 32-bit accesses. In PSoC 3 devices, the CPU has 8-bit direct access to the SRAM.
- Zero wait state accesses.
- Arbitration of SRAM accesses by the CPU and the DMA controller.
- Different blocks can be accessed simultaneously by the CPU and the DMA controller.

PSoC 3 has idata and xdata RAM.

The internal data space (idata) is 384 bytes, compressed within a 256-byte space. This space consists of 256 bytes of RAM and a 128-byte space for Special Function Registers (SFRs). See Figure 2. The lowest 32 bytes are

used for four banks of registers R0-R7. The next 16 bytes are bit-addressable.

In addition to the register or bit address modes used with the lower 48 bytes, the lower 128 bytes can be accessed with direct or indirect addressing. With the direct addressing mode, the upper 128 bytes map to the SFRs. With the indirect addressing mode, the upper 128 bytes map to RAM. The stack operations use indirect addressing.

The xdata space is 24-bit or 16 MB in size. The majority of this space is not “external” - it is used by on-chip components. It can only be accessed using indirect addressing. The 8051 core features dual DPTR registers for faster xdata transfer operations. The data pointer selects SFR and DPS, and selects which data pointer register (DPTR0 or DPTR1) is used for the following instructions:

MOVX @DPTR, A

MOVX A, @DPTR

MOVC A, @A+DPTR

JMP @A+DPTR

INC DPTR

MOV DPTR, #data16

The Variable Memory test implements the Periodic Static Memory test H.2.19.6 defined by the IEC 60730 standard. It detects single-bit faults in the variable memory. The variable memory contains data, which is intended to vary during the 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 some of the most widely used static memory algorithms to check for DC Faults. The following tests can be implemented using the Class B Safety Software Library:

- March Test
- March C Test
- March C Minus Test
- March B Test

March Test

March tests perform a finite set of operations on every memory cell in the memory array. Each operation performs the following tasks:

1. Writes ‘0’ to a memory cell (w0).
2. Writes ‘1’ to a memory cell (w1).
3. Reads the expected value ‘0’ from a memory cell (r0).
4. Reads the expected value ‘1’ from a memory cell (r1).

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)

March C Test

The March C test is a destructive test and used to detect the following types of Faults in the variable memory:

- Stuck-at Fault
- Addressing Fault
- Transition Fault
- Coupling Fault

This test complexity is 11n, where n indicates the number of bits in memory. This test is a destructive test (which means that memory contents are not saved). Therefore, it is designed to run at the system startup before initializing memory and the run-time libraries.

MARCH C ALGORITHM:

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

March C Minus Test

The March C Minus test is a destructive test and used to detect the following types of fault in the variable memory:

- Stuck-at Fault
- Addressing Fault
- Transition Fault
- Coupling Fault

This test complexity is 10n, where n indicates the number of bits in the memory. This test is a destructive test. Therefore, it is designed to run at the system startup before initializing the memory and the run-time libraries.

MARCH C MINUS ALGORITHM:

```
MarchC
{
```

<> (w0); > (r0, w1); > (r1, w0);
<> (r0); < (r0, w1); < (r1, w0);
}

March B Test

The March B is a destructive test that can detect the following types of fault:

- Stuck-at
- Linked Idempotent Coupling
- Inversion Coupling

This test complexity is 17n, where n indicates the number of bits in the memory.

MARCH B ALGORITHM:

MarchB
{
<> (w0); > (r0, w1, r1, w0, r0, w1); > (r1, w0, w1);
< (r1, w0, w1, w0); < (r0, w1, w0);
}

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 tests in run time without data corruption.

You can define March C Minus, March C, or March B using the following parameters in SelfTest_SRAM_March.a51 file for PSoC 3:

```
MARCH_C_MINUS          EQU      0x00
MARCH_C                EQU      0x01
MARCH_B                EQU      0x02

TYPE_OF_MARCH_TEST     EQU MARCH_C_MINUS
```

Or in SelfTest_SRAM_March.s file for PSoC 5LP:

```
.equ MARCH_C_MINUS,         0x00
.equ MARCH_C,               0x01
.equ MARCH_B,               0x02

.equ TYPE_OF_MARCH_TEST,   MARCH_C
```

Using the start address and end address pointers, the function performs runtime MARCH B/C/C Minus tests 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 the data is copied. The reserved area of SRAM must be located at the end of SRAM and can be set using the following parameters in the SelfTest_SRAM_March.a51 file for PSoC 3:

```
;Low bytes of addresses is 0x00.
START_BUFF_ADDR_H      EQU      0x1E ;00
END_BUFF_ADDR_H        EQU      0x20 ;00
```

Or in SelfTest_SRAM_March.s file for PSoC 5LP:

```
.equ START_BUFF_ADDR_H, 0x2000
.equ START_BUFF_ADDR_L, 0x7400

.equ END_BUFF_ADDR_H, 0x2000
.equ END_BUFF_ADDR_L, 0x7800
```

During the call, this function tests one block of SRAM. The size of the block must be the same as the size of the reserved buffer area in SRAM, and can be set using the following parameters in SelfTest_SRAM_March.a51 file for PSoC 3:

```
;High byte of block size in SRAM test
;Low byte of address is 0x00.
BLOCK_SIZE_BYTE_H      EQU      0x02;00
```

Or in SelfTest_SRAM_March.s file for PSoC 5LP:

```
/* Block size in SRAM test */
.equ BLOCK_SIZE_BYTE, 0x0400
```

This test does the test only on the SRAM area and does not cover 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.

Function

```
uint8  SelfTests_IRAM_March (void)

Returns:  0   No error
          1   Error detected

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

This function is called to do the March Internal RAM tests in run time without data corruption for PSoC 3.

More information about Internal RAM of PSoC 3 is described in the section CPU Registers Test.

You can define March C Minus, March C, or March B using the following parameters in SelfTest_IRAM_March.a51:

```
MARCH_C_MINUS          EQU      0x00
MARCH_C                EQU      0x01
MARCH_B                EQU      0x02

TYPE_OF_MARCH_TEST     EQU MARCH_C_MINUS
```

Using the start address and end address pointers, the function performs runtime MARCH B/C/C Minus tests by backing up the area of IRAM under test to the reserved part of SRAM and then restoring the data.

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

During the call, this function tests one block of IRAM. The size of the block must be the same as the size of the reserved buffer area in SRAM, and can be set using the following parameters in the SelfTest_IRAM_March.a51 file for PSoC 3:

```
;High byte of block size in SRAM test
;Low byte of address is 0x00.
BLOCK_SIZE_BYTE_H      EQU      0x02;00
```

This test does the test only on the IRAM. If the block is tested successfully and the end address of IRAM is reached, the test returns 0x03. If the block is tested successfully but the end address of IRAM is not reached, the test returns 0x02.

Function

```
uint8  SelfTests_Stack_March (void)

Returns:  0   No error
          1   Error detected

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

This function is called to do the March Stack tests in run time without data corruption for PSoC 5LP.

You can define March C Minus, March C, or March B using the following parameters in the SelfTest_SRAM_March.s file for PSoC 5LP:

```
.equ MARCH_C_MINUS,        0x00
.equ MARCH_C,              0x01
.equ MARCH_B,              0x02

.equ TYPE_OF_MARCH_TEST,   MARCH_C
```

Using the start address and end address pointers, this function performs runtime MARCH B/C/C Minus tests by backing up the area of Stack under test to the 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 that is used for the SRAM March test.

During the call, this function tests one block of Stack. The size of block must be the same as the size of the reserved buffer area in SRAM, and can be set using the following parameters in the SelfTest_IRAM_March.s file for PSoC 5LP:

```
/* Block size in SRAM test */
.equ BLOCK_SIZE_BYTE, 0x0400
```

This test does the test only on the Stack. If the Stack is tested successfully, the function returns 0x00. If not, it returns 0x01.

Stack Overflow Test

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

The Stack Pointer (SP) register is used to access the stack. The Stack pointer in PSoC 3 is only 8 bits wide, can address 256 bytes, and is incremented with every **PUSH** instruction (stack grows upwards) and decremented with every **POP**. The stack in PSoC 3 is located in idata space.

In PSoC 5LP, the stack is located at the end of RAM and grows downward: the stack pointer is 32 bits wide, is decremented with every **PUSH** instruction, and incremented with **POP.**

The purpose of the stack overflow test is to ensure that the stack did 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 and this should be treated as overflow error.

Functions

```
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 predefined pattern.

```
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 runtime 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
```

Digital I/O

The PSoC I/Os provide:

- Digital input sensing
- Digital output drive
- Pin interrupts
- Connectivity for analog inputs and outputs
- Connectivity for LCD segment drive and EMIF
- Access to internal peripherals:
  - Directly for defined ports
  - Through the Universal Digital Blocks (UDB) via the Digital System Interconnect (DSI)

The 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 the 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, actual resistance values are compared to the PSoC internal pull-up resistors.

To detect the pin-to-ground short, use the following next method.

The pin is configured in the resistive pull-up drive mode. In 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 Sensor-to-V_{CC} short, use a similar method.

In this case, the sensor pin is configured in the resistive pull-down drive mode. The input level is zero in normal conditions.

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

Function

```
uint8 SelfTests_IO()

Returns:  0   No error
          1   Error detected

Located in:   SelfTest_IO.c
              SelfTest_IO.h
```

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

For example:

```
uint8 CYXDATA * const CYCODE
Pin_pcTable[] =
{

        /* PORT0 */
        (uint8 CYXDATA *)CYREG_PRT0_PC0,
        (uint8 CYXDATA *)CYREG_PRT0_PC1,
        (uint8 CYXDATA *)CYREG_PRT0_PC2,
        (uint8 CYXDATA *)CYREG_PRT0_PC3,
        (uint8 CYXDATA *)CYREG_PRT0_PC4,
        (uint8 CYXDATA *)CYREG_PRT0_PC5,
        (uint8 CYXDATA *)CYREG_PRT0_PC6,
        .....
```

A/D Converter

The ADC test implements independent input comparison H.2.18.8, defined by the IEC 60730 standard. The test provides a fault/error control technique by which the inputs/outputs that are designed to be within specified tolerances are compared.

The purpose of the test is to test the ADC analog functions.

This test is easily implemented using the reconfigurable hardware of PSoC and by using the internal bandgap reference as the ADC input. For example, the PSoC internal bandgap reference delivers a 1.024-V voltage, which can be regularly sampled to test the ADC.

For this test, PGA, VREF, and analog multiplexers are needed.

If the PGA is already used in the design, the opamp is also needed to connect the VREF to the multiplexer through the opamp. VREF cannot be connected to the multiplexer directly. Therefore, the opamp connects the VREF to the multiplexer before the PGA input.

Figure 9 shows the schematic implementation of this test based on PSoC 3/5LP.

The PGA_Vin, ADC_Plus, and ADC_Minus are the user pins.

Before the test, the analog multiplexers disconect the ADC and PGA from the user pins, and reconfigure the internal connections necessary for the ADC self-tests. After the test, the user configuration is restored.

The ADC inputs switch between the corresponding pins (or other user modules, such as the PGA, TIA, and opamp) and the reference voltages, by using an analog multiplexer

The reference voltages can be connected to both polarities so that it is possible to calculate the ADC gain and ADC offset providing increased system accuracy.

The delta-sigma ADC has four configurations. One of these configurations is used for the test and named “ClassB”. The other three are used for user purposes. The “ClassB” configuration can measure ±2.048 V.

Function

```
uint8  SelfTest_ADC(void)

Returns:  0   No error
          1   Error detected

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

The schematic is shown in Figure 9.

The ADC accuracy is defined in “*SelfTest_Analog.h*”. A 10-bit ADC is used for testing:

```
#define ADC_TEST_ACC 10
            // +/- ADC result value
```

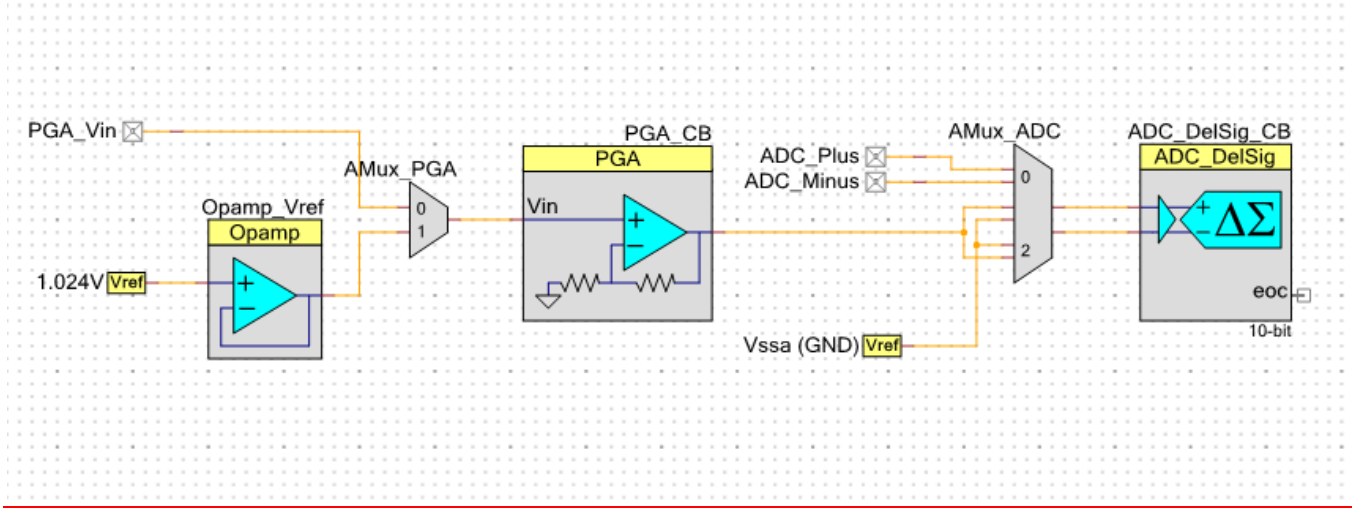
This function implements the ADC value test in four cases:

- Vin PGA – 1.024V, PGA Gain -1;
  ADCplus - 1.024V; ADCminus – GND;
  Expected ADC result: + 1.024V;
- Vin PGA – 1.024V, PGA Gain -1;
  ADCplus - GND; ADCminus - 1.024V;
  Expected ADC result: - 1.024V;
- Vin PGA – 1.024V, PGA Gain -2;
  ADCplus - 2.048V; ADCminus – GND;
  Expected ADC result: + 2.048V;
- Vin PGA – 1.024V, PGA Gain -2;
  ADCplus - GND; ADCminus - 2.048V;
  Expected ADC result: - 2.048V;

The test is a success if the digitalized input voltage value is equal to the required reference voltage value within the defined accuracy. If 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 9. ADC Test PSoC Implementation



D/A Converter

The DAC test implements independent output comparison H.2.18.8, defined by 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 DAC analog functions.

This test is easily implemented using the reconfigurable hardware of the PSoC and can be done using the ADC to measure the DAC output. For example, the DAC generates a set voltage, which can be regularly converted by the ADC to test the DAC.

The ADC and analog multiplexers are needed for the test.

Figure 10 shows the schematic implementation of this test based on PSoC 3/5LP.

Vout, ADC_Plus, ADC_Minus are the user pins.

Before the test, the analog multiplexers disconnect the DAC and ADC from the user pins, and reconfigure the internal connections needed for the DAC tests. After the test, the user configuration is restored.

The ADC inputs switch between the corresponding pins (or other user modules, such as the PGA, TIA, opamp) and the reference voltages by using an analog multiplexer.

The Track and Hold unit is used to maintain the DAC output value if a continuous voltage output (VDAC) is required by the end application. A continuous current output (IDAC) is not possible during testing without using a second IDAC during the test.

The delta sigma ADC has four configurations. One of these configurations is used for all the analog self-tests and is named “ClassB”. The other three are used for the user purposes. The “ClassB” configuration can measure ±2.048 V.

Function

```
uint8  SelfTest_DAC(void)

Returns:  0   No error
          1   Error detected

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

The schematic is shown in Figure 10.

The DAC accuracy is defined in “*SelfTest_Analog.h*”. A 10-bit ADC is used for testing.

```
#define DAC_TEST_ACC (15u)

    // +/- DAC result value
```

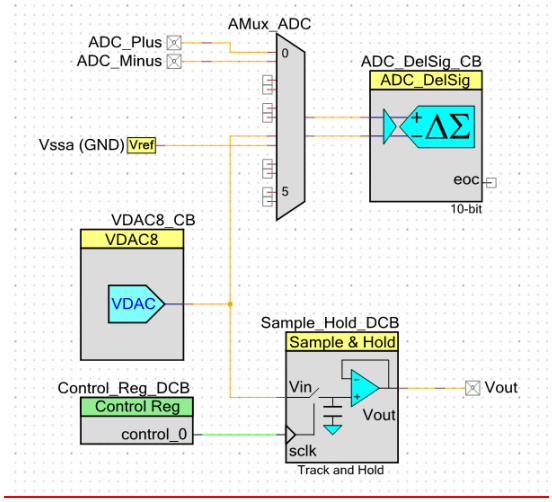
This function implements the DAC value test. The digital value converts to the analog voltage using the tested DAC, and this voltage converts to a digital value using the ADC. If both values are within the set tolerance, the test is passed. If the test is a success, the function returns 0, otherwise it returns 1.

Before the test, the DAC configures to the output range of 0 to 4.080 mV. After the test, the user configuration is restored.

The test range is defined in DAC_TEST_RANGE_1 (7u) and DAC_TEST_RANGE_2 (128u). The DAC step is 16 mV. The test range is from 112 mV to 2048 mV.

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

Figure 10. DAC Test PSoC Implementation



Comparator

This function implements the comparator functional test. Analog signals of opposite polarity are sequentially put into the input of the analog comparator, forcing the output value of the comparator to change. When the test is a success, the function returns 0; otherwise, it returns 1.

Function

```
uint8  SelfTest_Comparator(void)

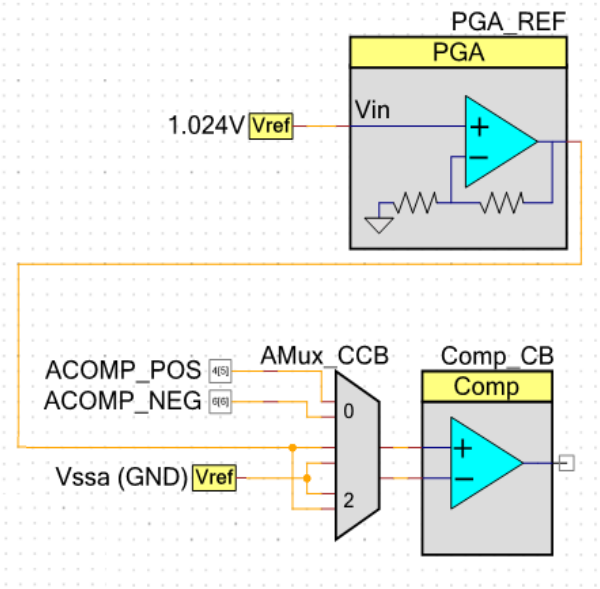
Returns:  0   No error
          1   Error detected

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

Figure 11 shows this test PSoC schematic implementation. The comparator input terminals switch between the corresponding pins (or other user modules, such as the PGA, TIA, and opamp) and the reference voltages using an analog multiplexer. 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 all the component configurations and non-retention registers before testing and restores them later.

Figure 11. Comparator Test PSoC Implementation



PGA

The Programmable Gain Amplifier (PGA) test is not provided in the IEC 60730 standard but is useful when the PGA is used in critical blocks.

The PGA test falls under the principle of testing “independent output comparison” defined in H.2.18.8 by 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 PGA analog functions.

This test is easily implemented using the reconfigurable hardware of the PSoC and can be performed by using the internal bandgap reference connected to the PGA input. The PGA delivers a voltage, which can be regularly converted by the ADC to test the PGA functions.

The ADC, opamp, VREF, and analog multiplexers are needed for the test. Because VREF cannot be connected to the multiplexer directly, the opamp is used to connect the VREF to the multiplexer before the PGA input.

Figure 12 shows the schematic implementation of this test based on PSoC 3/5LP.

PGA_Vin, ADC_Plus, and ADC_Minus are the user’s pins.

Before the test, the analog multiplexers disconect the ADC and PGA from the user’s pins, and reconfigure the internal connections needed for the ADC self tests. After the test, the user configuration is restored.

The ADC inputs switch between the corresponding pins (or other user modules, such as the PGA, TIA, and opamp) and the reference voltages by using an analog multiplexer

The delta-sigma ADC has four configurations. One of these configurations is used for the test and named “ClassB”. The other three are used for the user purpose. The “ClassB” configuration can measure ±2.048 V.

Function

```
uint8  SelfTest_PGA(void)

Returns:  0   No error
          1   Error detected

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

The schematic is shown in Figure 12.

The PGA test accuracy is defined in “*SelfTest_Analog.h*”. A 10-bit ADC is used for testing:

```
#define PGA_TEST_ACC 10
       // +/- PGA result value
```

This function implements the PGA test by measuring the PGA output value using the ADC in two cases:

- Vin PGA – 1.024 V, PGA Gain -1;

  Expected ADC result: +1.024V;

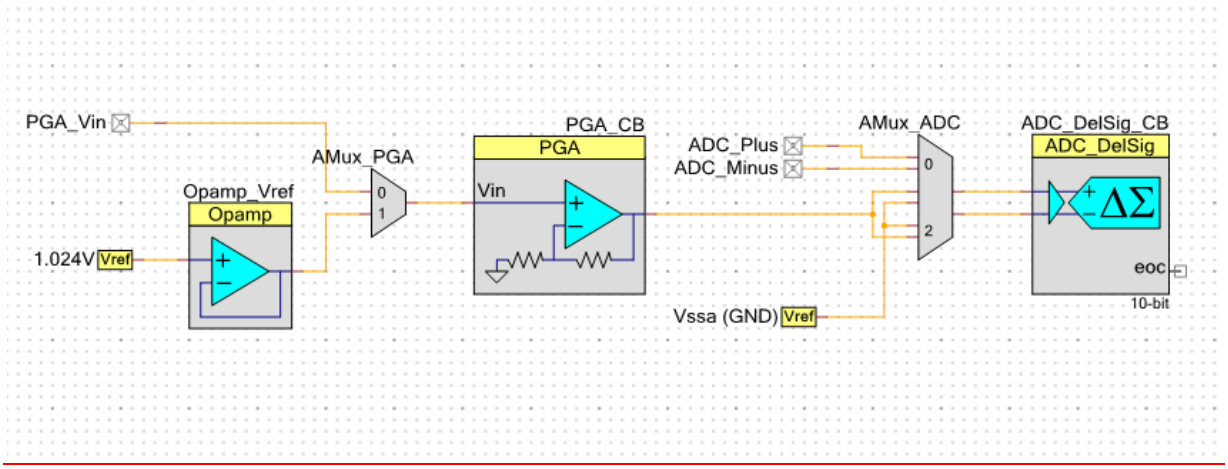
- Vin PGA – 1.024 V, PGA Gain -2;

  Expected ADC result: +2.048 V;

The test is a success if the digitalized ADC 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 12. PGA Test PSoC Implementation



TIA

The TIA test in not provided in the IEC 60730 standard but is useful when used in critical blocks.

The TIA test falls under the principle of testing “independent output comparison” defined in H.2.18.8 by 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 TIA analog functions.

This test is easily implemented using the reconfigurable hardware of the PSoC and can be performed using the ADC and IDAC to generate a current and measure the TIA output. For example, the IDAC generates a current and the TIA converts the current to a voltage, which can be regularly converted by the ADC.

The IDAC, ADC, and analog multiplexers are needed for the test.

Figure 13 shows the schematic implementation of this test based on PSoC 3 and PSoC 5LP.

The Iin, ADC_Plus, and ADC_Minus are user pins.

Before the test, the analog multiplexers disconect the TIA and ADC from the user pins and reconfigure the internal connections needed for the TIA self-tests. After the test, the user configuration is restored.

The ADC inputs switch between the corresponding pins (or other user modules, such as the PGA, TIA, and opamp) and reference voltages by using an analog multiplexer.

The delta-sigma ADC has four configurations. One of these configurations is used for all the analog self-tests and named “ClassB”. The other three are meant for the user. The “ClassB” configuration can measure ±2.048 V.

Function

```
uint8  SelfTest_TIA(void)

Returns:  0   No error
          1   Error detected

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

The schematic is shown in Figure 13.

The TIA test accuracy is defined in “*SelfTest_Analog.h*”. A 10-bit ADC is used for testing.

```
#define TIA_TEST_ACC (15u)
        // +/- TIA result value
```

This function implements the TIA value test and consists of four steps:

1. The digital value converts to a current using the IDAC.
2. This current converts to a voltage using the TIA.
3. The voltage converts to a digital value using the ADC.
4. If the resulting value matches the expected value within the set tolerance, the test is passed. When the test is a success, the function returns 0, else returns 1.

The TIA Vout is determined by the following equation, where the RFB is the resistive feedback:

$$VOUT = VREF - Iin \text{ x RFB}$$

$$VREF = \frac{VDD}{2} = 3.3\frac{V}{2} = 1.65\,V$$

Where RFB = 20 kΩ.

The IDAC is configured in the output range of 0 to 255 uA.

The TIA test range is defined in IDAC_TEST_RANGE_1 (0u) and IDAC_TEST_RANGE_2 (78u). IDAC step is 1 uA.

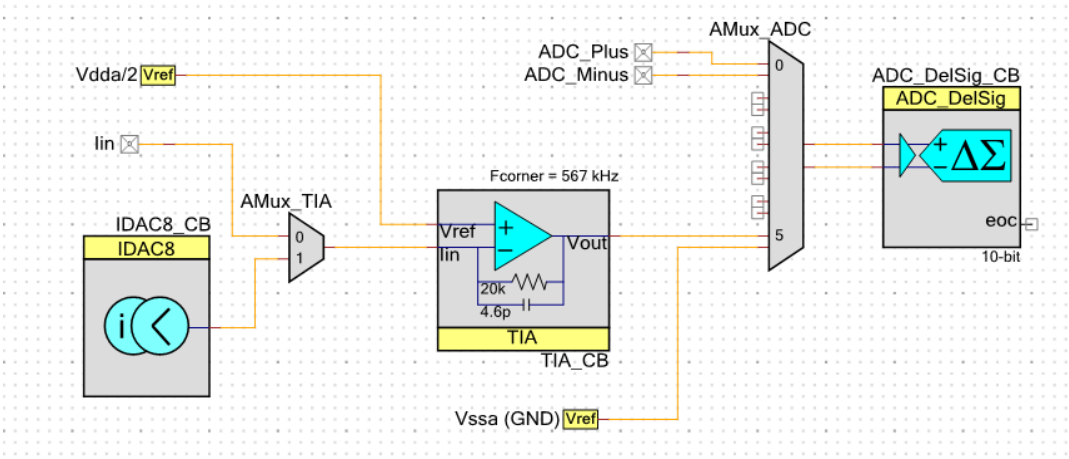
The TIA test range is:

[1.65 V – (IDAC_TEST_RANGE_2 - 1)uA * 20 kΩ ; 1.65 V - IDAC_TEST_RANGE_1 uA * 20 kΩ]

[110 mV; 1650 mV]

The test function saves the ADC configuration before testing and restores it after the test.

Figure 13. TIA Test PSoC Implementation



Opamp

The Opamp test is not provided in the IEC 60730 standard but is useful when the opamp is used in critical blocks.

The opamp test falls under the principle of testing “independent output comparison” defined in H.2.18.8 by 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 and can be performed by using the internal band-gap reference to the Opamp input. The opamp output signal can be regularly converted by the ADC to test opamp functions.

The PGA, ADC, and VREF, and the analog multiplexers are needed for the test.

If the PGA is already used in the design, the opamp is also needed to connect the Vref to the multiplexer through the Opamp. The Vref cannot be connected to the multiplexor directly. Therefore, the opamp is used to connect the Vref to the multiplexor before the PGA input. Figure 14 shows the schematic implementation of this test based on PSoC 3/5LP.

The Vin, ADC_Plus, ADC_Minus, Opamp_Plus, and Opamp_Minus are user pins.

Before the test, the analog multiplexers disconnect the opamp, ADC, and PGA from the user pins, and reconfigure the internal connections needed for the opamp self-tests. After the test, the user configuration is restored.

The ADC inputs switch between the corresponding pins (or other user modules, such as the PGA, TIA, and opamp) and reference voltages by using an analog multiplexer.

The delta-sigma ADC has four configurations. One of these configurations is used for the test and named “ClassB”. The other three are used for user purposes. The “ClassB” configuration can measure ±2.048 V.

Function

```
uint8  SelfTest_Opamp(void)

Returns:  0   No error
          1   Error detected

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

The schematic is shown in Figure 14.

The opamp test accuracy is defined in “*SelfTest_Analog.h*”. A 10-bit ADC is used for testing:

```
#define OPAMP_TEST_ACC 10
        // +/- Opamp result value
```

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

- Vin PGA – 1.024 V, PGA Gain -1;

  The opamp is configured for the following mode:

  Opamp Plus – 1.024 V

  The expected ADC result: + 1.024 V;

- Vin PGA – 1.024 V, PGA Gain -2;

The opamp is configured for the following mode.

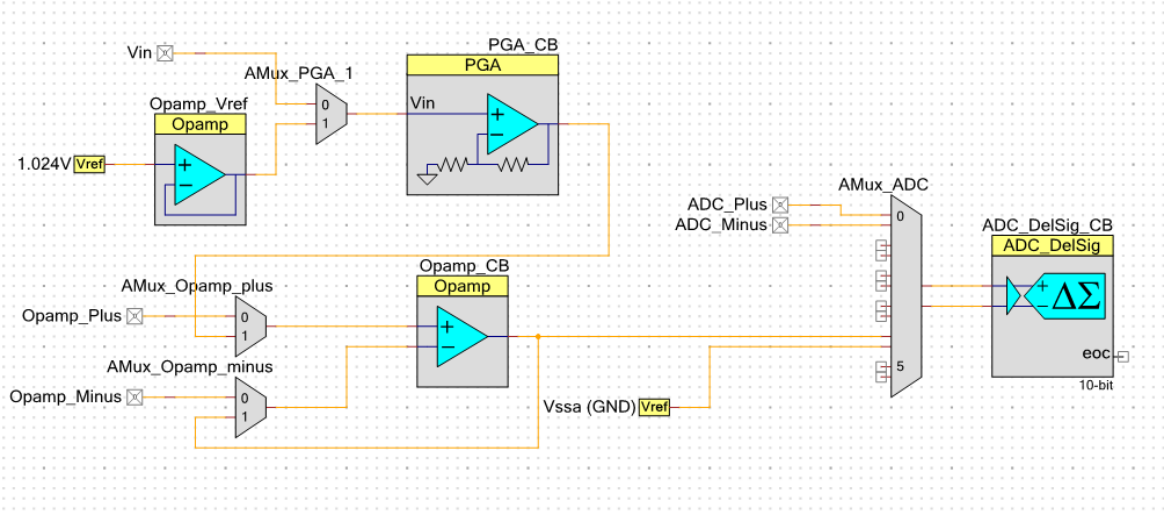
  Opamp Plus – 2.048 V

  The expected ADC result: + 2.048 V;

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

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

Figure 14. Opamp Test PSoC Implementation



Communications UART

This function implements the UART internal data loopback test. The test is a success if the transmitted byte is equal to the received one. If the test is a success, the function returns 0; otherwise, it returns 1.

Function

```
uint8  SelfTests_UART_Data(void)

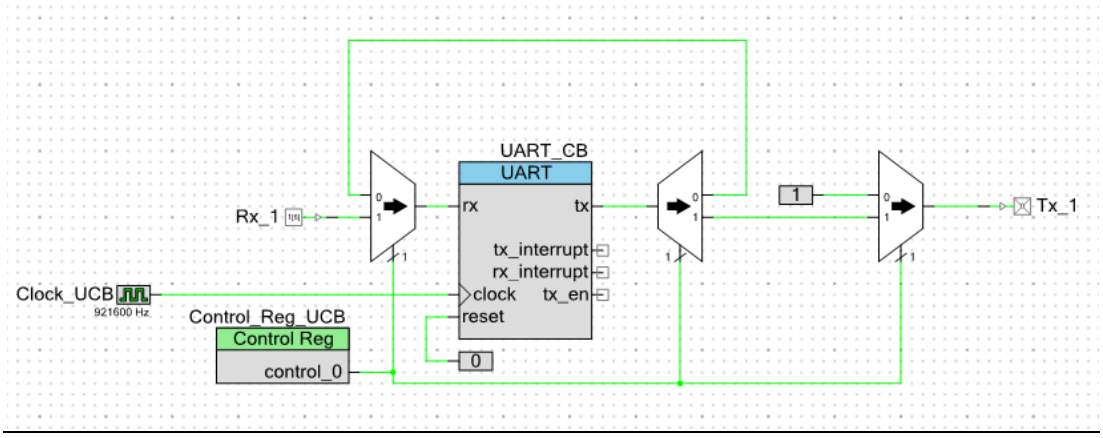
Returns:  1   Error detected
          2   Pass test with current value,
              but not all tests in range
              (from 0x00 to 0xFF) are
              complete
          3   Pass, completed testing with
              all tests range (from 0x00 to
              0xFF)
          4   ERROR_TX_NOT_EMPTY
          5   ERROR_RX_NOT_EMPTY
          6   ERROR_TX_NOT_ENABLE
          7   ERROR_RX_NOT_ENABLE

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

Figure 15 shows the PSoC schematic implementation. 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 `ERROR_RX_NOT_EMPTY` or `ERROR_TX_NOT_EMPTY` status.

The test function saves the component configuration and non-retention registers before testing and restores them later. During the call, the function transmits one byte. The transmitted value increments after each function call. The range of test values is from 0x00 to 0xFF.

Figure 15. UART Test PSoC Implementation



Communications SPI

This function implements the SPI internal data loopback test. The test is a success if the transmitted byte is equal to the received one. If the test is a success, the function returns 0; otherwise, it returns 1.

Function

```
uint8  SelfTest_SPI_Data(void)

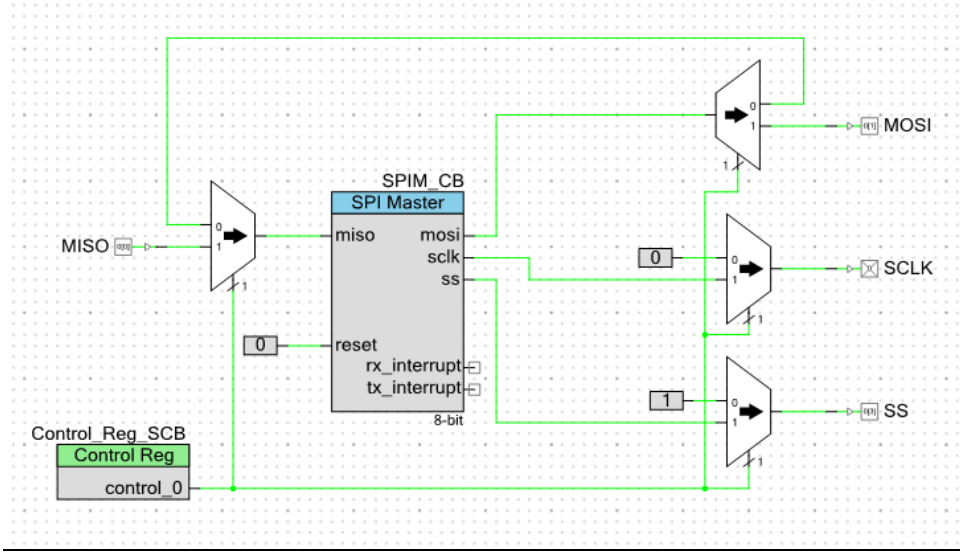
Returns:  1   Error detected
          2   Pass test with current
              values, but not all tests in
              range (from 0x00 to 0xFF)
              ware passed
          3   Pass, tested with all tests
              in range (from 0x00 to 0xFF)
          4   ERROR_TX_NOT_EMPTY
          5   ERROR_RX_NOT_EMPTY

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

Figure 16 shows this test's PSoC schematic implementation. 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 `ERROR_RX_NOT_EMPTY` or `ERROR_TX_NOT_EMPTY` status.

The test function saves all the component configuration and non-retention registers before testing and restores them later. During the call, the function transmits one byte. The transmitted value increments after each function call. The range of test values is from 0x00 to 0xFF.

Figure 16. SPI Test PSoC Implementation



Recommendations and Examples

The following section describes how to use PSoC's programmable interconnectivity for self-testing components.

UDB Configuration Registers Test

There are two main groups of UDB registers:

- UDB Working registers. Located in the main 64K page (Page 0)
- UDB Configuration registers. Located in the dedicated configuration page (Page 1)

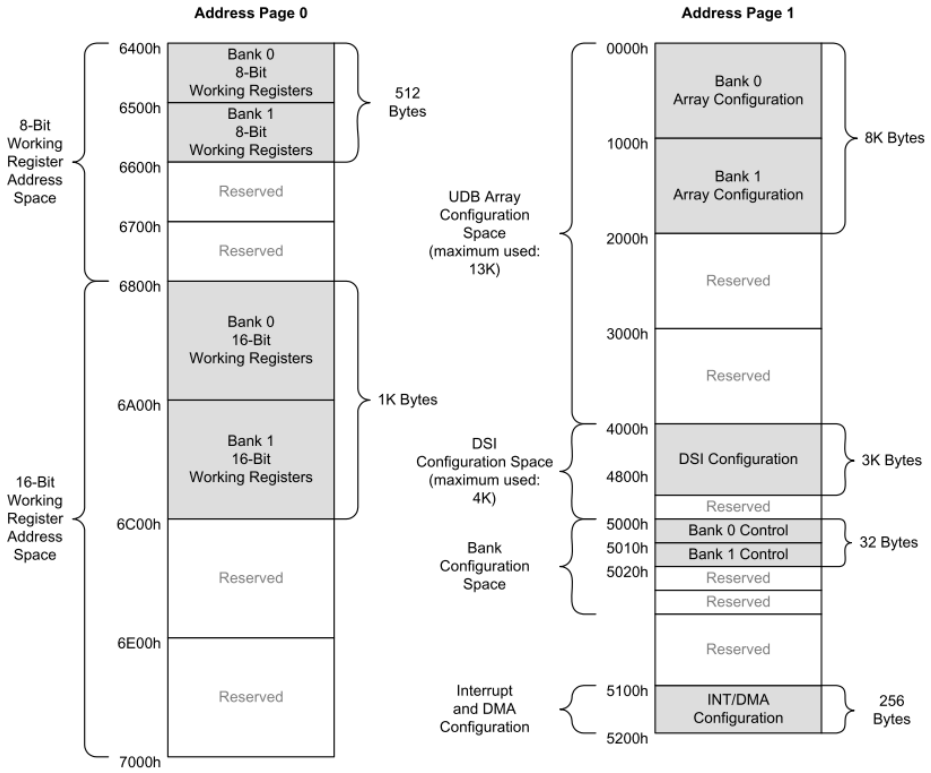
UDB Array Base Addresses are described in Figure 17.

The UDB configuration registers are static and configured during design build. Do not change these registers during device operation. You can check against the initial configuration.

The following functions allow you to implement the UDB configuration register tests in your design. There are two test modes implemented in the functions:

- Duplicates of UDB configuration registers are stored in the flash memory after device start-up. The configuration registers are periodically compared with the stored duplicates. Additionally, ECC can be enabled to monitor duplicate errors stored in flash. The corrupted registers can be restored from flash after checking.
- Compare the calculated CRC with the CRC previously stored in EEPROM if the CRC status semaphore is set. If the status semaphore is not set, the CRC must be calculated and stored to EEPROM and the status semaphore must be set.

Figure 17. UDB Array Base Addresses



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
              SelfTest_CustomFlash.c
              SelfTest_CustomFlash.h
```

Description:
This function stores the UDB configuration registers to flash memory. These registers are located in the memory address from 0x00010000u to 0x00012000u and occupy 8 KB, located in 32 blocks of flash memory. The number of flash blocks is defined in UDB_NUMBER_OF_BLOCKS. UDB_REG_FIRST_BLOCK and DB_REG_LAST_BLOCK define the location for the UDB configuration registers in the flash memory.

Note This function should be called once after PSoC initialization before entering the main program. It writes the correct UDB configuration register values to flash.

Function

```
uint8          SelfTests_UDB_ConfigReg(uint8
numberOfBlocks)
```

```
Parameters:  numberOfBlocks  -  number  of
blocks to be tested per 1 function call. 32
- used in demo project.
```

```
Returns:  1   Error detected
          2   PASS for one block
          3   PASS   for   all   tested   UDB
              registers
```

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

Description:

This function checks the UDB configuration registers.

The defined number of UDB register blocks is tested per one function call. The size of the block is 256 bytes. Because the size of one block of flash is the same, it simplifies working with flash.

The number of tested blocks during the call can be set using the following parameters in the SelfTest_UDB_CfgReg.h file:

```
#define BLOCKS_PER_TEST (2u)
```

There are two modes of checking:

- CRC16 calculation and checking
- Registers comparison with duplicates.

You can define the mode using the following parameters in the SelfTest_UDB_CfgReg.h file:

```
#define UDB_CFG_REGS_MODE
CFG_REGS_CRC_MODE/ CFG_REGS_TO_FLASH_MODE

#define CFG_REGS_TO_FLASH_MODE (1u)
```

This mode stores the duplicates of registers to FLASH and compares the registers with duplicates. Returns fail if values are different.

Registers can be restored in this mode.

SelfTests_Save_UBD_Cfg() function used to store duplicates to FLASH. UDB_REG_FIRST_BLOCK and DB_REG_LAST_BLOCK defines the location for UDB configuration registers in Flash memory.

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

In this mode, the function calculates a CRC16 of registers and stores the CRC to EEPROM. Later function calls recalculate the CRC16 and compare it with the saved value. Returns fail if values are different.

Start-up Configuration Registers Test

This test describes and shows an example of how to check the start-up configuration registers:

- Test digital clocks, ILO, IMO, PLL, and bus/master clock configuration registers
  - Test DMAC configuration registers (set to default values after start-up)
  - Test analog configuration registers (set to default values after start-up)
  - Test cyfitter configuration registers

These start-up configuration registers are static and are located in the *cyfitter_cfg.c* file after the design is built. This example covers tests cyfitter_cfg configuration functions:

- cfg_dma_init()
- ClockSetup()
- AnalogSetDefault()
- Other configuration registers generated in cyfitter_cfg()

There are two test modes implemented in these functions:

- Duplicates of startup configuration registers are stored in the flash memory after device start-up. The configuration registers are periodically compared with stored duplicates. ECC can also be enabled to monitor duplicate errors. Corrupted registers can be restored from flash after checking.
- Compare the calculated CRC with the CRC previously stored in EEPROM if the CRC status semaphore is set. If the status semaphore is not set, the CRC must be calculated and stored to EEPROM and the status semaphore must be set.

Note The following functions are examples and can only be applied to the example project. If the user makes changes in the schematic or configuration, other configuration registers may be generated in the *cyfitter_cfg.c* file. Do not change the list of required registers (recommended registers are generated in the cyfitter_cfg() function).

Note If the clock self-test is used, enable CLOCK_SELF_TEST_ENABLED defined in the *SelfTest_ConfigRegisters.c* file. It excludes checking of the CYREG_CLKDIST_DCFG0_CFG0, CYREG_CLKDIST_DCFG1_CFG0, and CYREG_CLKDIST_DCFG2_CFG0 registers, generated in the ClockSetup() function. This register is used for the Clock self-test and may change at run time causing a test error.

Function

```
cystatus SelfTests_Save_cfg(void)

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

Located in:  SelfTest_ConfigRegisters.c
             SelfTest_ConfigRegisters.h
             SelfTest_CustomFlash.c
             SelfTest_CustomFlash.h
```

Description:

This function stores all listed start up configuration registers to rowData array and writes this array to the flash block defined in CFG_REG_BLOCK.

Side Effects:

Call this function once after PSoC initialization before entering the main program. It writes the correct startup configuration registers to flash.

Function

```
uint8 SelfTests_StartUp_ConfigReg(void)
Returns:  0   No error
              1   Error detected
Located in:   SelfTest_ConfigRegisters.c
              SelfTest_ConfigRegisters.h
```

Description:

This function checks the listed startup configuration registers.

There are two modes of checking:

- CRC16 calculation and checking
- Registers comparison with duplicates.

The user 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 to flash and compares the registers with the duplicates. It returns a fail (1) if the values are different.

The registers can be restored in this mode.

The SelfTests_Save_cfg function is used to store duplicates to FLASH. CFG_REG_BLOCK, which defines the location of the Start Up configuration registers in the flash memory.

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

In this mode, the function calculates a CRC16 of registers and stores the CRC to EEPROM. Later, the function calls recalculate the CRC16 and compare it with the saved value. It returns a fail (1) if the values are different.

Watchdog Test

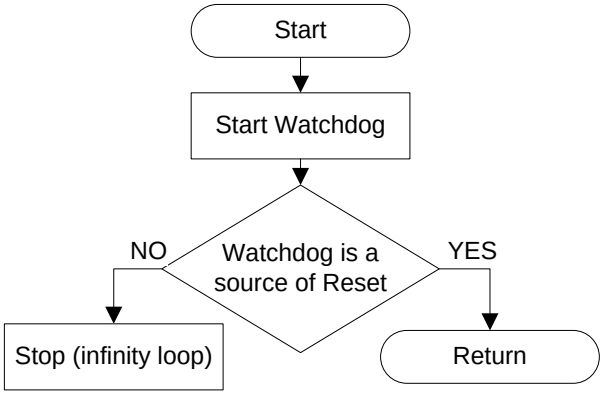
This function implements the watchdog functional test. The function starts the watchdog timer and runs an infinite loop. If the watchdog timer works, it generates a reset. After the reset, the function analyzes the reset source. If the watchdog is the source of the reset, the function returns; otherwise,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

Function

```
void   SelfTest_Watchdog(void)
Returns:  None
Located in:   SelfTest_WDT.h
              SelfTest_WDT.c
```

Figure 18 shows the test flowchart.

Figure 18. WDT Test PSoC Implementation



Windowed Watchdog Timer

The watchdog timer increases reliability in microprocessor-based systems. Window-selectable watchdog timers allow adjusting the watchdog time-out period, thus allowing 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 the 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.

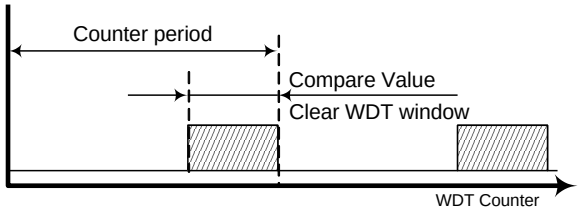
There is no Windowed Watchdog Timer in PSoC 3/PSoC 5LP, but it can be implemented using the reconfigurable hardware of PSoC.

Figure 21 shows the schematic implementation of this test based on the PSoC 3/5LP.

The Windowed Watchdog Timer (Windowed WDT) provides a way to demand that the ClearWDT instruction be executed (such as, only in the last quarter of the watchdog timeout period). Essentially this improves code flow monitoring to catch firmware bugs. For example, a user has an application bug that results in the clear watchdog timer repeatedly execute close to the beginning of the code flow. This can be interpreted as a normal operation in the Non Windowed Watchdog Timer mode. Most users just use the Non Windowed Watchdog Timer mode.

Figure 19. Windowed WDT Timing Diagram

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



The limitation in the Windowed Watchdog Mode is that the ClearWDT instruction must be called within a prescribed window. This limits the tolerance of the clock source that drives the watchdog timer. The tolerance of the clock source of the Windowed Watchdog also defines the minimum and maximum nominal watchdog period.

A counter and flip-flops are used to implement the Windowed WDT in PSoC 3/PSoC 5LP. Figure 20 shows the Windowed WDT counter configuration.

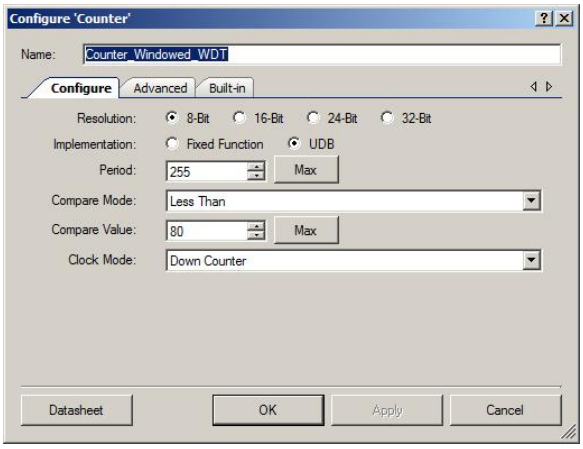
Clock_Count, and Counter Period are used to define the maximum waiting (maximum FW execution) time for clear command.

The Counter Compare value is used to define the width of the clearing window. The down counting counter output “comp” changes as follows during the count period:

- First Window: from (255u) to (80u) output “comp” is “0”
- Second Window: from (80u) to (0u) output “comp” is “1”

If the Windowed WDT clear happens in the first window or does not happen during the counter period, this means incorrect FW operation and PSoC requires reset. The clear Windowed WDT means trigger “1” in the Clear_Window_Watch_Dog Control Register.

Figure 20. Windowed WDT Counter Configuration.



The three stages of the Windowed WDT schematic operation are described in

Figure 21:

1. The Flip Flop DFF1 detects if Clear happens in the second window (The Clear_Window_Watch_Dog Control Register is triggered when Clock_Count is between 80u to 0u).
2. The Flip Flop DFF2 detects if Clear happens in the first window (The Clear_Window_Watch_Dog Control Register is triggered when Clock_Count is between 255u to 80u).
3. The Flip Flop DFF3 detects if Clearing does not happen during the specific period of time.

The ISR isr_Windowed_WDT is triggered in cases 2 or 3 (if clear happens in the first or second window).

The ISR is used to detect a reset in the example project for the purpose of demonstration.

In the customer design, a pin connected to the HW reset can be used instead of the ISR.

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

Integration of safety inputs, such as windowed WDT or interlock inputs, with output logic and pins can create 100% hardware safety control with no CPU processing.

Functions:

There are two functions to operate the Windowed Watchdog Timer.

`void Windowed_WDT_Start(void)` –This function starts operation of the Windowed Watchdog Timer.

`void Windowed_WDT_Clear(void)` –This function clears the Windowed Watchdog Timer.

These functions are located in the *SelfTest_Windowed_WDT.c* and *SelfTest_Windowed_WDT.h* files.

Figure 21. Windowed Watchdog Timer Implementation in PSoC 3/PSoC 5LP

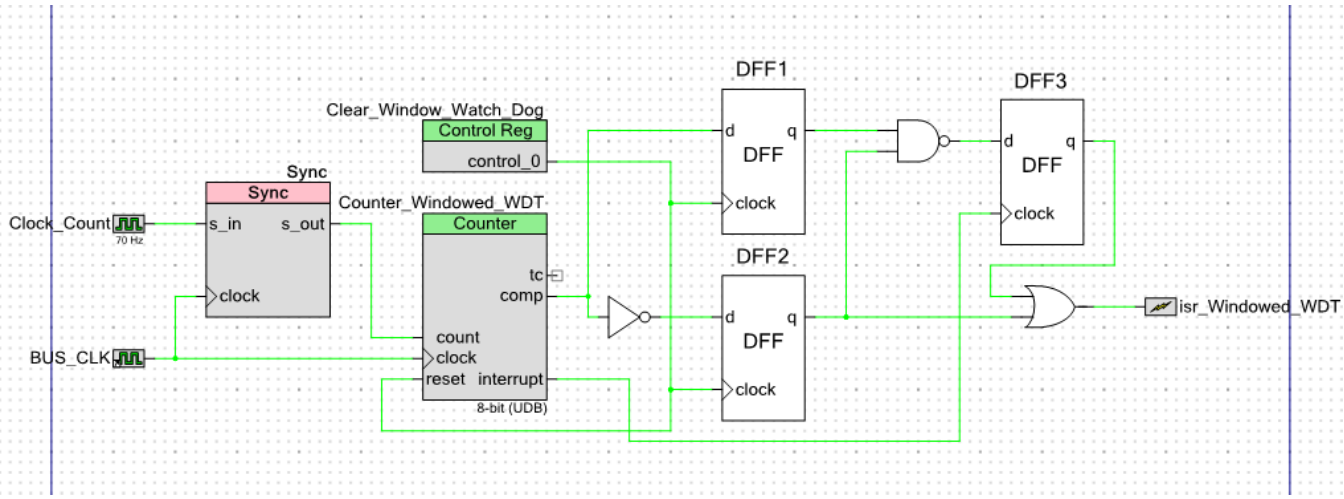
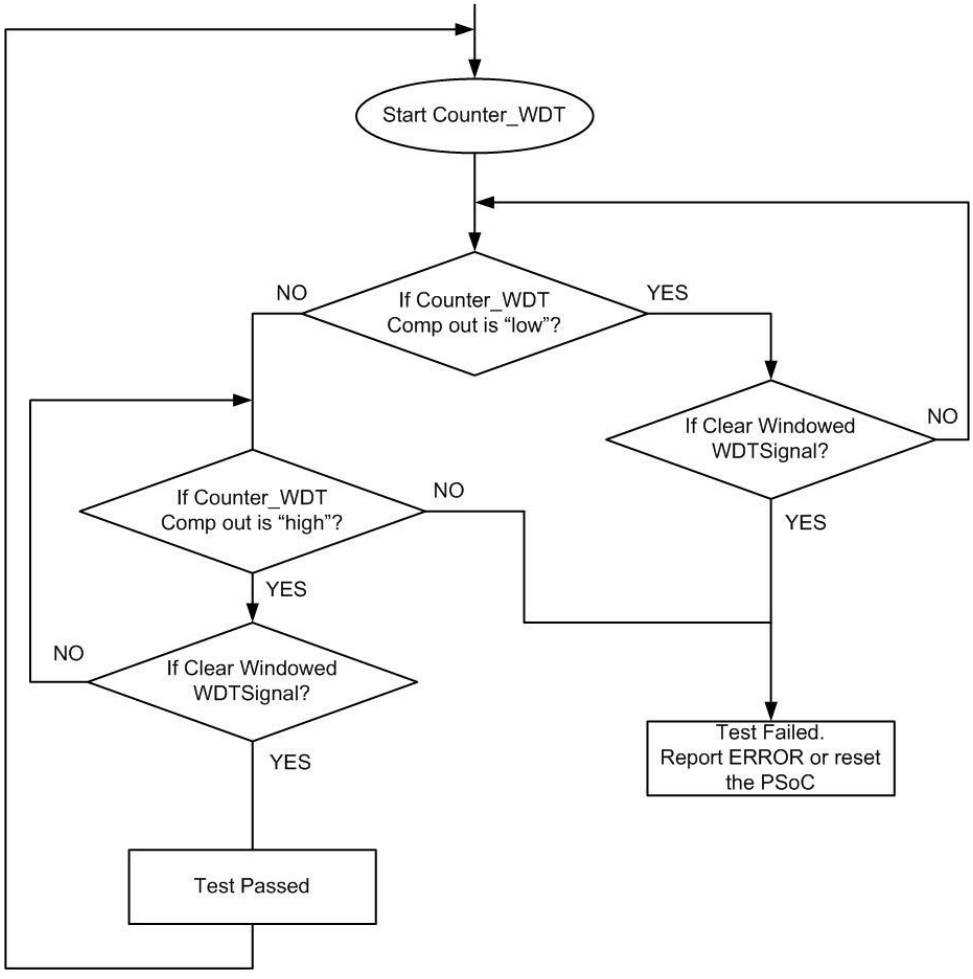


Figure 22. Flow Chart of Working Window WDT



Example of Communications UART Data Transfer Protocol

Additional system safety can be achieved for data transfer between system components, using communication protocols with CRCs and packet handling. An example of safety communication is described in this application note.

Data is placed in 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 23 shows the packet format.

Figure 23. Packet Structure

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

Channel controlling symbols

STX – Marker of the packet start. This is a unique byte that is always equal to ***0x02***.

ESC – Marker of a 2-byte sequence. When any byte in a packet is equal to **STX** or **ESC,** it changes to a 2-byte sequence (**ESC**)(**byte+1**). This procedure provides a unique packet start symbol. The **ESC** byte is always equal to ***0x1B***.

The following table shows the packet's field description.

Name	Length	Value	Description
STX	1 byte	0x02	Unique start packet marker
ADDR	1or 2 bytes	0x00..0xFF except 0x02	Slave address. If this byte is equal to **STX** it changes to 2 byte sequence: (**ESC**)+(**STX+1**) If this byte is equal to **ESC** it changes to 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 2 byte sequence: (**ESC**)+(**STX+1**) If this byte is equal to **ESC** it changes 2 byte sequence: (**ESC**)+(**ESC+1**)
D0..Dn (data)	1..510 bytes	0x00..0xFF except 0x02	Packet's data. If any byte in data is equal to **STX** it changes to 2 byte sequence: (**ESC**)+(**STX+1**) If any byte in data is equal to **ESC** it changes to 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 last data byte. CRC is calculated after the **ESC** changing procedure. If this byte is equal to **STX** it changes to 2 byte sequence: (**ESC**)+(**STX+1**) If this byte is equal to **ESC** it changes to 2 byte sequence: (**ESC**)+(**ESC+1**)
CRCL	1 or 2 byte	0x00.. 0xFF except 0x02	LSB of packet CRC. **CRC-16** is used. CRC is calculated for all packet's bytes from **ADDR** to last data byte including. CRC is calculated after the **ESC** changing procedure. If this byte is equal to **STX** it changes to 2 bytes sequence: (**ESC**)+(**STX+1**) If this byte is equal to **ESC** it changes to 2 bytes sequence: (**ESC**)+(**ESC+1**)

Data Delivery Controlling

Figure 24 shows the communication process-timing diagram. The communication procedure can be divided into three parts:

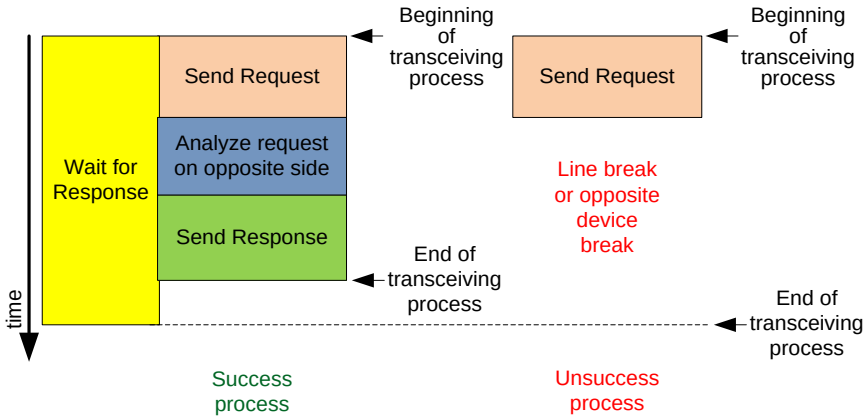
- Send Request (opposite side Receive Request)
- Wait for Response (opposite side Analyze Request)
- Receive Response (opposite side Send Respond)

The 'Send request' consists of sending the STX, sending the data length and data using the byte changing procedure, calculating the CRC, and sending the calculated result.

'Receive respond' consists of finding the STX and starting the CRC counter. If the received address is invalid, the STX byte is searched again. If the address is valid, the data length is received and then the data bytes are received. The CRC counter then stops and the two CRC bytes are received and these bytes are compared with the stored CRC value.

After sending a request, we start the guard timer to detect the end of the packet if a response is not sent.

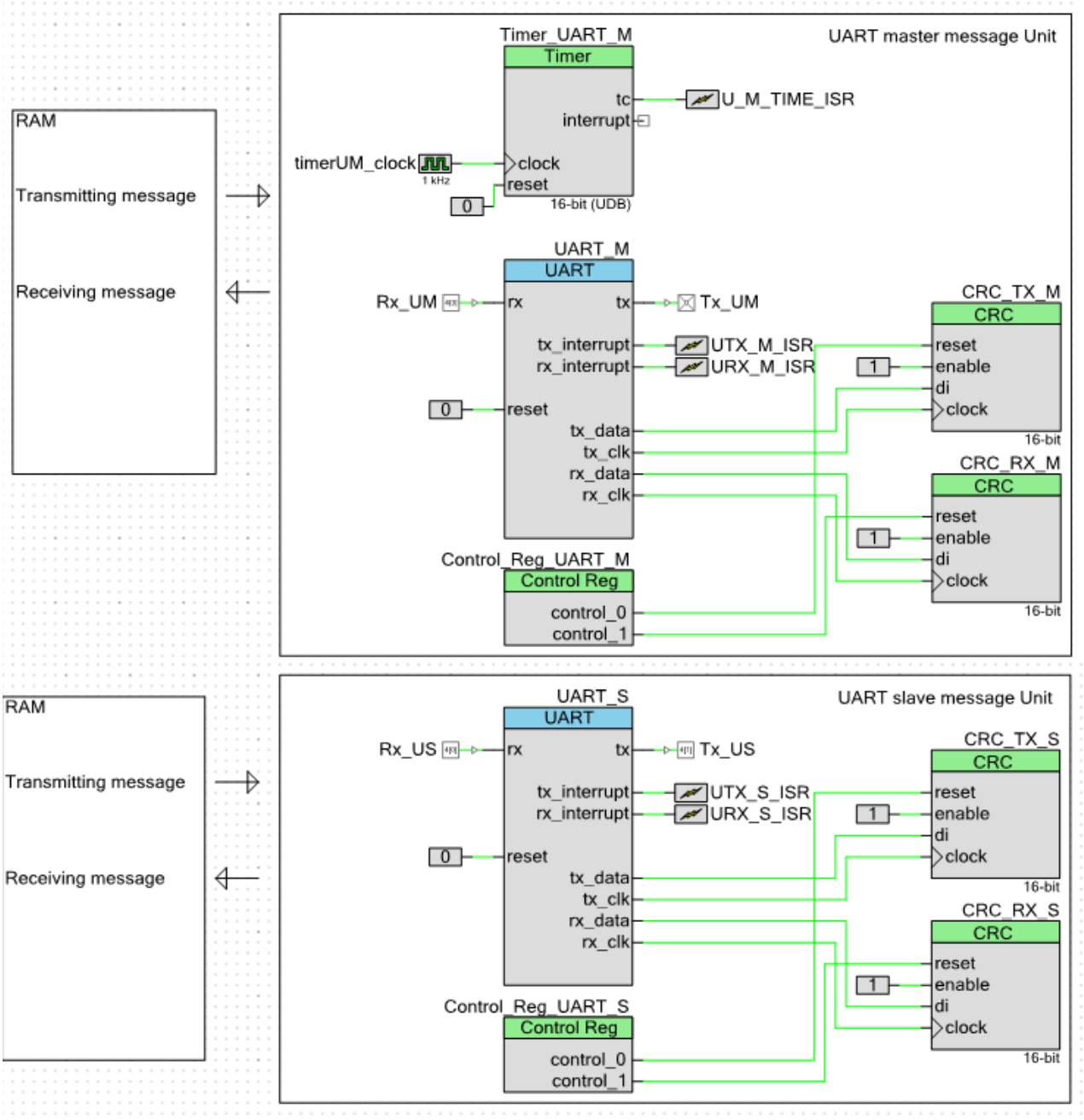
Figure 24. Timing Diagram of Communication Process



PSoC Implementation

Figure 25 represents the PSoC protocol implementation. The UART component is used to physically generate the signals. To provide the CRC-16 calculation, the CRC units are used (one for transmit and one for receive). To control the CRC calculation a control register is used. To detect an unsuccessful packet transaction the timer is used. There are three interrupts implemented in this project that 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 in the TX buffer and the Transmit complete event to start/stop the CRC calculation. The receive interrupt in the UART is configured for a FIFO not empty event to analyze the received data, control the CRC calculation and store the received data into RAM. The timer interrupt is used to detect the end of an unsuccessful transmission.

Figure 25. PSoC 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 provided in the background.

There are four functions for working with the protocol unit for the master:

Function

`void UartMesMaster_Init(void)`

`Returns: None`

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

This function initializes the UART message unit.

Function

```
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 be that 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 – the pointer to the transmitted data (Request data)
- tlen – the length of the request in bytes
- rxd – a pointer to the buffer where the received data is stored (Received data)
- rlen – the length of the received buffer in bytes

Function

```
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. There are three possible results:

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

Function

```
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.

There are five functions for working with the protocol unit for the slave.

Function

```
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 parameters are:

Address – the slave address.

Function

```
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 the response.

This function returns the result of the start of the process: if it succeeds, it returns 0 and 1 if there is an error (the unit has not received a marker).

The input parameters are:

- txd – the pointer to the transmitted data (Request data)
- tlen – the length of the request in bytes

Function

```
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 current state of the UART message unit. There are three possible results:

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

Function

```
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 stored in the receive buffer. If the unit state is not UM_PACKREADY, it returns 0.

Function

```
uint8 * UartMesSlave_GetDataPtr(void)

Returns: return pointer to data

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

This function obtains pointer to the received data.

Summary

This application note describes how to implement various diagnostic tests defined by the IEC 60730 standard. The introduction of IEC 60730 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 of PSoC 3 and PSoC 5LP, designers can comply with regulations while maintaining or reducing electronic systems cost. The use of PSoC and this Class B Safety Software Library enables creating a strong system-level development platform, achieving superior performance, fast time to market, and energy efficiency.

References

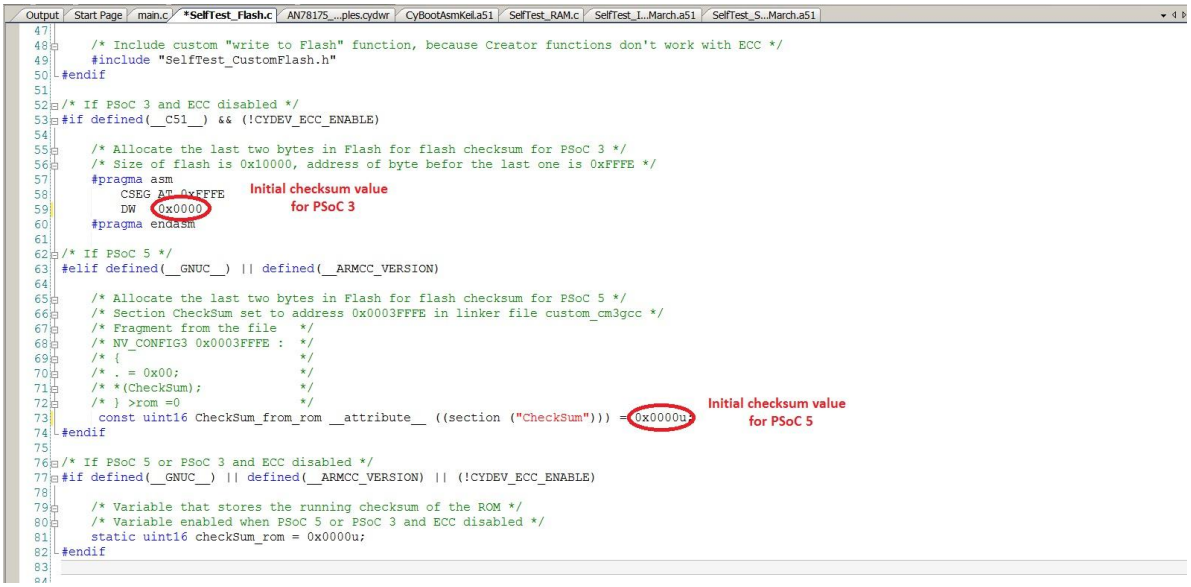
IEC 60730 Standard, “Automatic Electrical Controls for Household and Similar Use”, IEC 60730-1 Edition 3.2, 2007-03.

Appendix A

The following instructions help to program your part for proper flash/ROM diagnostic testing.

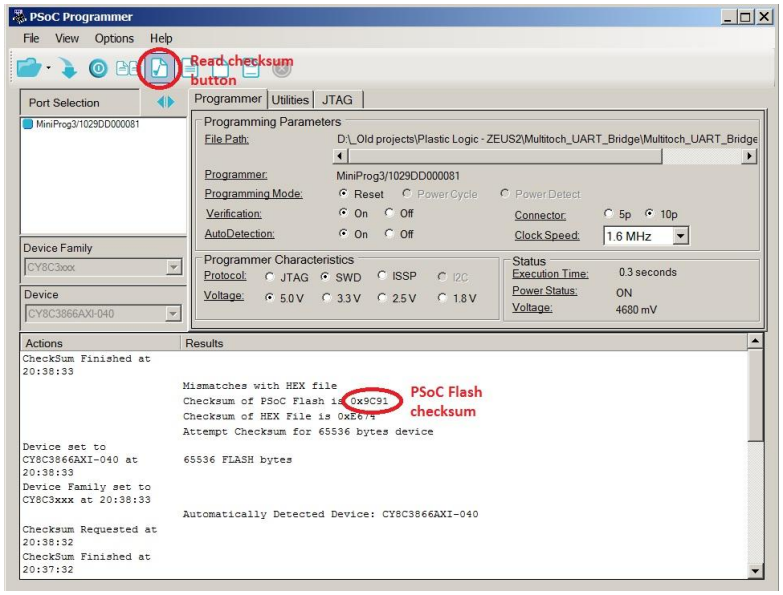
1. Build the project in PSoC Creator; make sure that the initial checksum value is set to '0'.

Figure 26. Build Project



2. Open PSoC Programmer and read the flash checksum as shown in the following figure.

Figure 27. Copy Checksum Value from PSoC Programmer



3. In PSoC Creator, assign the copied value to the checksum. Click the **Program** button.

Figure 28. Reassign Checksum Constant with Actual Checksum for PSoC 3

```
/* If PSoC 3 and ECC disabled */
#if defined(__C51__) && (!CYDEV_ECC_ENABLE)

    /* Allocate the last two bytes in Flash for flash checksum for PSoC 3 */
    /* Size of flash is 0x10000, address of byte befor the last one is 0xFFFE */
    #pragma asm
        CSEG AT 0xFFFE
        DW   0x9C91
    #pragma endasm

/* If PSoC 5 */
#elif defined(__GNUC__) || defined(__ARMCC_VERSION)
```

Figure 29. Reassign Checksum Constant with Actual Checksum for PSoC 5LP

```
/* If PSoC 5LP */
#else

    #if defined(__GNUC__)

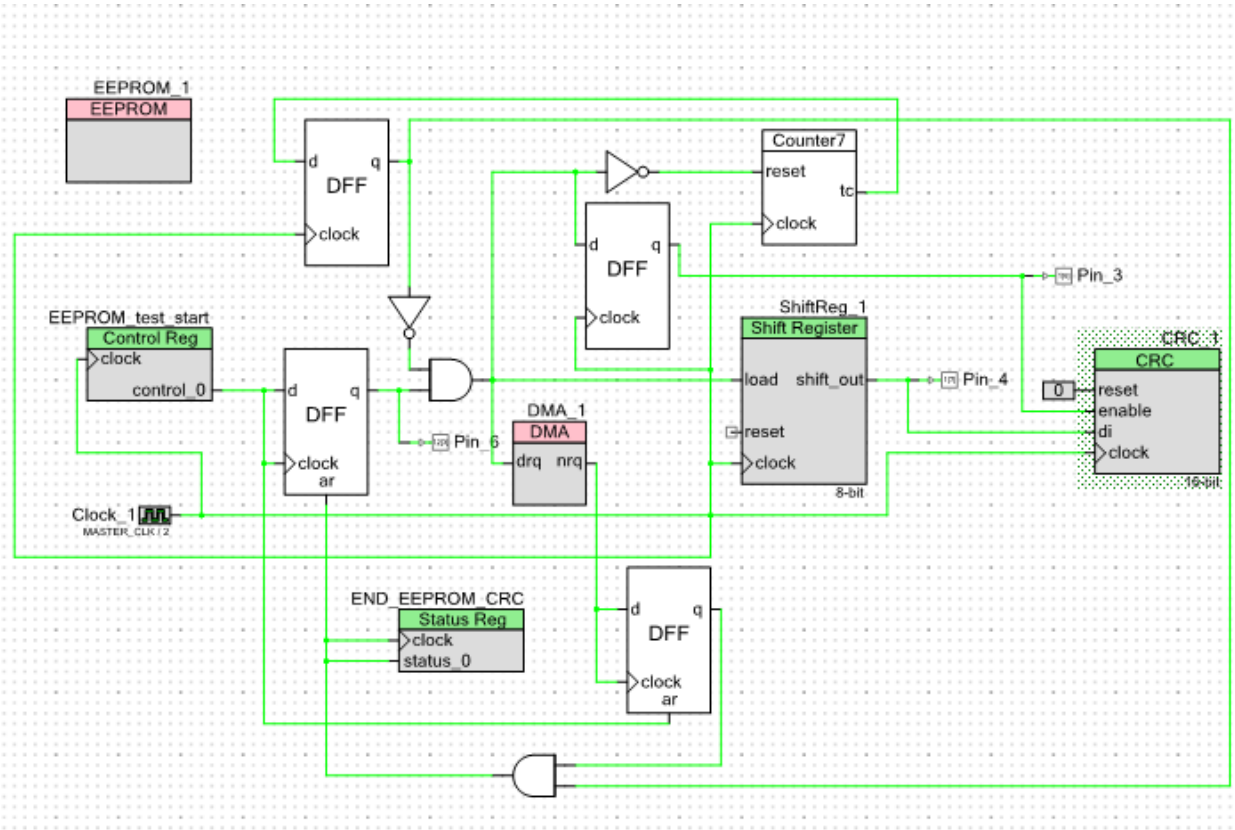
        /* Allocate the last two bytes in Flash for flash checksum for PSoC 5LP */
        /* Section CheckSum set to address 0x0003FFFE in linker file custom_cm3gcc */
        /* Fragment from the file   */
        /* NV_CONFIG3 0x0003FFFE :  */
        /* {                        */
        /* . = 0x00;                */
        /* *(CheckSum);             */
        /* } >rom =0                */
        const uint16 CheckSum_from_rom __attribute__ ((used, section("CheckSum"))) = 0x9C91;
    #elif defined(__CC_ARM)

        /* Allocate the last 2 bytes in flash for checksum. (ARM MDK and RVDS compilers) */
        const uint32 CheckSum_from_rom __attribute__ ((at(0x0003FFFC))) = 0x9C910000;
    #endif
```

Appendix B

EEPROM CRC Calculation at the Hardware Level using Hardware CRC Component

Figure 30. Hardware CRC Calculation



The EEPROM CRC calculates the CRC using a hardware CRC component freeing up CPU resources. Data from the EEPROM transfers to the Shift Register through the DMA. The Shift Register was used because the CRC component computes the CRC from a serial bit stream. Four D Flip Flops and a Counter7 (based on the Status Register) were used for synchronization.

Appendix C

Table 2. Supported Part Numbers

PSoC 3		PSoC 5LP
CY8C3244AXA-XXX	CY8C3446AXI-XXX	CY8C5265AXI-XXXXX
CY8C3244AXI-XXX	CY8C3446LTI-XXX	CY8C5265LTI-XXXXX
CY8C3244LTI-XXX	CY8C3446PVA-XXX	CY8C5266AXI-XXXXX
CY8C3244PVA-XXX	CY8C3446PVE-XXX	CY8C5266LTI-XXXXX
CY8C3244PVI-XXX	CY8C3446PVI-XXX	CY8C5267AXI-XXXXX
CY8C3245AXA-XXX	CY8C3645AXE-XXX	CY8C5267LTI-XXXXX
CY8C3245AXI-XXX	CY8C3646AXE-XXX	CY8C5268AXI-XXXXX
CY8C3245LTI-XXX	CY8C3646PVE-XXX	CY8C5268LTI-XXXXX
CY8C3245PVA-XXX	CY8C3665AXA-XXX	CY8C5465AXI-XXXXX
CY8C3245PVE-XXX	CY8C3665AXI-XXX	CY8C5465LTI-XXXXX
CY8C3245PVI-XXX	CY8C3665LTI-XXX	CY8C5466AXI-XXXXX
CY8C3246AXA-XXX	CY8C3665PVA-XXX	CY8C5466LTI-XXXXX
CY8C3246AXI-XXX	CY8C3665PVI-XXX	CY8C5467AXI-XXXXX
CY8C3246LTI-XXX	CY8C3666AXA-XXX	CY8C5467LTI-XXXXX
CY8C3246PVA-XXX	CY8C3666AXI-XXX	CY8C5468AXI-XXXXX
CY8C3246PVI-XXX	CY8C3666LTI-XXX	CY8C5468LTI-XXXXX
CY8C3444AXA-XXX	CY8C3666PVA-XXX	CY8C5666AXI-XXXXX
CY8C3444AXI-XXX	CY8C3845PVE-XXX	CY8C5666LTI-XXXXX
CY8C3444LTI-XXX	CY8C3846AXE-XXX	CY8C5667AXI-XXXXX
CY8C3444PVA-XXX	CY8C3865AXA-XXX	CY8C5667LTI-XXXXX
CY8C3444PVE-XXX	CY8C3865AXI-XXX	CY8C5668AXI-XXXXX
CY8C3444PVI-XXX	CY8C3865LTI-XXX	CY8C5668LTI-XXXXX
CY8C3445AXA-XXX	CY8C3865PVA-XXX	CY8C5866AXI-XXXXX
CY8C3445AXE-XXX	CY8C3865PVI-XXX	CY8C5866LTI-XXXXX
CY8C3445AXI-XXX	CY8C3866AXA-XXX	CY8C5867AXI-XXXXX
CY8C3445LTI-XXX	CY8C3866AXI-XXX	CY8C5867LTI-XXXXX
CY8C3445PVA-XXX	CY8C3866LTI-XXX	CY8C5868AXI-XXXXX
CY8C3445PVI-XXX	CY8C3866PVA-XXX	CY8C5868LTI-XXXXX
CY8C3446AXA-XXX	CY8C3866PVI-XXX	
CY8C3446AXE-XXX		

Appendix D

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the test projects.

MISRA stands for Motor Industry Software Reliability Association. The MISRA specification covers a set of 122 mandatory rules and 20 advisory rules that apply to firmware design. It has been put together by the automotive industry 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 3	Production
	PSoC 5LP	Production
Target Compiler	PK51	9.03
	GCC	4.4.1
Generation Tool	PSoC Creator	2.2 SP1
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

The list of deviated rules is provided in the following table.

Table 4. Deviated Rules

MISRA-C: 2004 Rule	Rule Class (R/A)	Rule Description	Description of Deviations
1.1	R	This rule states that code shall conform to the C ISO/IEC 9899:1990 standard.	Some C language extensions (such as interrupt keyword) relate to device hardware functionality and cannot be practically avoided. In the main.c file generated by PSoC Creator, the non-standard main() declaration is used: “void main()”. The standard declaration is “int main()”.
3.1	R	All usage of implementation-defined behavior shall be documented.	For the documentation on PK51 and GCC compilers, refer to the PSoC Creator Help menu, documentation sub-menu, Keil, and GCC commands respectively.
8.7	R	Objects shall be defined at block scope if they are only accessed 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 5LP, some objects are declared with external linkage in *.c files and these declarations are not in the 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 user application and might not be used in the 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,	File SelfTest_Analog.c: uint16 DACval variable is converted to a signed value to check its range (± constant value).

MISRA-C: 2004 Rule	Rule Class (R/A)	Rule Description	Description of Deviations
		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	
10.3	R	The value of a complex expression of integer type shall only be cast to a type of the same signedness that is no wider than the underlying type of the expression.	File SelfTest_Analog.c: uint16 DACval variable is converted to signed value to check it range (± constant value). Uint8 IDAC_Value variable is converted to signed int16 to be subtracted from the signed int16 constant value.
11.3	A	A cast should not be performed between a pointer type and an integral type.	See “List of pointer violations in demo projects” in 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 “List of pointer violations in demo projects” in 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 “List of pointer violations in demo projects” in 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: iblock variable is zeroed within loop body to start test for next flash block.
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 user application and might not be used in library demo code.
14.1	R	There shall be no unreachable code.	Library APIs are designed to be used in the user application and might not be used in the 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 the 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 require switch case usage without break statement.
16.9	R	A function identifier shall only be used with either a preceding &, or with a parenthesised 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, 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 which are not used in demo projects but could be used in customer’s projects.
17.4	R	Array indexing shall be the only allowed form of pointer arithmetic.	See “List of pointer violations in demo projects” table below.
21.1	R	Minimization of run-time failures shall be ensured by the use of at least one of: 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 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;	HW registers 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 parameter to function where is accessed as indexed array. reg8 * RegPointer and reg8 * RegPointer1 are used as indexed arrays to access registers set. static uint8 CYCODE *Flash_Pointer is used as array of flash data and is indexed via “++” operation. uint16 *stack is used as array and is indexed via “++” operation. uint8 CYCODE * CfgRegPointer is used as indexed array to access data stored in flash. uint8 * txptr is used as array and is indexed via “++” operation.

Document History

Document Title: AN78175 – PSoC® 3 and PSoC 5LP - IEC 60730 Class B Safety Software Library

Document Number: 001-78175

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3567592	TARK_UKR ANBI_UKR DIMA_UKR	04/02/2012	New Spec.
*A	3624352	TARK_UKR	08/06/2012	Updated Recommendations and Examples (Removed CapSense CSD (Removed Test on Sensor Shorts, Test on Sensor Disconnect, and Modulator Component (Cmod and Rb) Testing), added UDB Configuration Registers test and Start up Configuration Registers test).
*B	3721601	TARK_UKR	8/23/2012	Exclude CapSense CSD self-tests Added two additional self-tests: UDB configuration register test and Start up configuration register test located on page 15, 16.
*C	3822654	MATT	11/27/2012	Updated for PSoC 5LP.
*D	4280250	VASY	02/13/2014	Added Program Flow Testing description. Added Stack Overflow Test. Added PSoC 3 idata and xdata difference description. Added Windowed WDT timing diagram. Added List of supported part numbers to Appendix C. Added MISRA Compliance description to Appendix D
*E	4500971	TARK	09/12/2014	Corrected the attached project.
*F	5726362	AESATMP9	05/04/2017	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

©Cypress Semiconductor Corporation, 2012-2017. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.