AN81828

PSoC® 1 - IEC 60730 Class B Safety Software Library

Author: Meenakshi Sundaram R

Associated Project: Yes

Associated Part Family: CY8C22x45, CY8C21x45

Software Version: PSoC Designer™ 5.4 CP1

Related Application Notes: AN75320, AN78175 and AN79973

AN81828 describes and includes the IEC 60730 Class B Safety Software Library. After reading this application note, you should be able to understand and easily integrate the IEC 60730 Class B Safety Software Library into your PSoC® 1 design.

Contents

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 a home appliance. However, MCU damage caused by overheating, static discharge, or other factors can cause the appliance to enter an unknown or unsafe state.

This application note focuses on Annex H, "Requirements for Electronic Controls," of the IEC 60730-1 standard. This portion of the standard details test and diagnostic methods to ensure the safe operation of embedded control hardware and software for home appliances. The PSoC 1 library files provided with this application note include self-test routines for ROM, RAM, CPU registers, interrupts, clocks, ADC, digital communication, comparator, watchdog timer (WDT), and CapSense®.

The application note assumes that the user is familiar with PSoC® 1 and PSoC® Designer IDE. Please refer Application note AN75320 - Getting Started with PSoC® 1 for getting started with PSoC® 1.

Refer IEC 60730 General Requirements document to know more about the safety standards covered in the application note.

Overview of Annex H

Annex H of the IEC 60730-1 standard classifies the appliance software into the following categories (see Appendix B: IEC 60730-1 Table H.11.12.7):

- Class A control functions, which are not required to check/detect unsafe operation of the equipment. Examples include humidity controls, lighting controls, timers, and switches.
- Class B control functions, which are intended to prevent unsafe operation of controlled equipment. Examples are thermal cut-offs and door locks for laundry equipment.
- Class C control functions, which are intended to prevent special hazards (such as an explosion caused by the controlled equipment). Examples include 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 into Class B. An exception is an appliance that might cause an explosion, such as a gas-fired controlled dryer, which falls into Class C.

The 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 the responses to them.

According to the IEC 60730-1 standard, a manufacturer of automatic electronic controls must design software using one of the following structures:

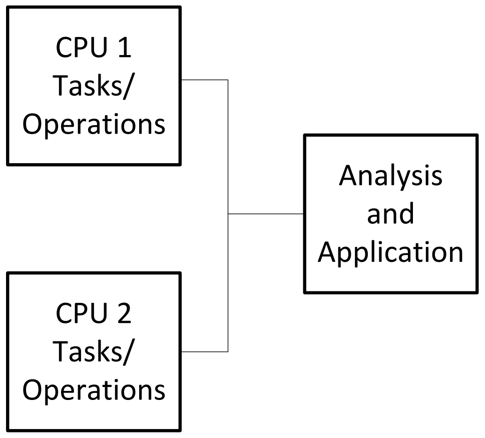
- Single channel with functional test
- Single channel with periodic self-test
- Dual channel without comparison

In the single-channel structure with the functional test, the software is designed using a single CPU to execute the functions as required. The functional test is executed before 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 executing. The CPU is expected to check regularly the critical functions of the electronic control without conflicting with the normal operation of the application.

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

Figure 1. Dual Channel Without Comparison Structure



The dual-channel structure implementation is more costly because two CPUs (or two MCUs) are required. In addition, it is more complex because the two devices need to communicate regularly with each other. The single-channel structure with the functional test is most commonly implemented today. However, appliance manufacturers are moving to the single-channel structure with the periodic self-test implementation for ensuring reliable operation during runtime.

Class B Requirements

Table H.11.12.7 in Annex H of the IEC 60730-1 standard specifies components that must be tested, depending on their software classification. Generally, each component offers various measures to verify or test the corresponding component(s). As a result, manufacturers have flexibility in selecting a suitable measure for their device.

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

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

Class B IEC 60730-1 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 interrupts
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 Table H.11.12.7
7.2.1 Analog A/D and D/A converters	Fault conditions specified in Appendix B: IEC 60730-1 Table H.11.12.7
7.2.2 Analog multiplexer	Wrong addressing

Class B Safety Software Library

You can use the Class B Safety Software Library described in this application note with CY8C22x45 and CY8C21x45 PSoC 1 devices. This library includes APIs that are intended to maximize application reliability through fault detection.

There are self-tests—for RAM, ROM, CPUs, registers, and so on—as mentioned in Table 1, that can be performed independent of the end application. These tests are provided as APIs that can be used as such or with minor modifications. There are other tests that depend on end-user applications, such as interrupt tests, communication protocol tests, and I/O periphery tests. For these tests, the concept and the implementation in the application are explained in this application note.

To run most of the self-test APIs, invoke an appropriate API function from the **.asm* or **.c* and **.h* and **.inc* files in the Class B Safety Software Library. Other self-tests can be applied by using an appropriate API function from the **.asm* or **.c* and **.h* and **.inc* files along with an appropriate user module placement in the project.

This application notes describes two types of self-test functions:

- The self-test functions to help meet the IEC 60730-1 Class B standard compliance are as follows:
  - CPU registers: Test on stuck bits.
  - Program counter: Test on jumps to the right address.
  - Interrupt handling and execution: Test on proper interrupt calling and periodicity.
  - Clock: Test on wrong frequency.
  - Flash (invariable memory): Test on memory corruption.
  - SRAM (variable memory): Test on stuck bits and proper memory addressing.
  - Digital I/O: Test on stuck bits.
  - A/D convertor: Test on proper functionality.
  - Communications (UART, SPI): Test on correct data receiving possibility.
  - Watchdog test: Test on chip reset possibility.
- Additional self-test functions that PSoC 1 can support due to programmable interconnectivity. The user application frequently needs these self-tests even though they are not provided in Appendix B: IEC 60730-1 Table H.11.12.7:
  - CapSense Sigma Delta (CSD): Test on sensor shorts, sensor disconnect, and modulator component (Cmod and Rb/IDAC) testing
  - Comparator test

- Checkerboard RAM test and redundant inverse storage test for global variable

All self-tests can be executed after device startup, before the main loop, where the application code resides. The self-tests can determine whether the chip is suitable to function.

In addition, self-tests must be executed while the device is functioning in a determined period of time. This lets you ensure that the chip was not damaged while executing application functions.

The following sections describe the self-tests and the implementation details for each test according to IEC 60730-1 Class B Annex H. In addition, each section lists the APIs provided with this application note to execute the corresponding test for the supported architecture. Any mention of PSoC 1 refers to the CY8C22x45 and CY8C21x45 family of PSoC 1 devices.

API Functions for PSoC 1

CPU Self-Test (Table 1 – 1.1)

PSoC 1 relies on five registers for program execution:

- Accumulator (A)
- Index (X)
- Program counter (PC)
- Stack pointer (SP)
- Flags (F)

To check for stuck bits within these registers, perform a checkerboard test. A checkerboard test writes alternate ‘1’s and ‘0’s to a register and verifies the value; then it writes alternate ‘0’s and ‘1’s and verifies the value again. This is done by first writing 0xAA and then 0x55 for each register. The switching of these values resembles the alternate-colored squares on a checkerboard, hence the name. The checkerboard method is implemented for all CPU registers except the program counter, for which testing is covered in the next section.

Function

```
BYTE   SelfTest_CPU_Registers(void)

Returns:

0   No error

1   Error in temporary register used for
    RAM tests

x   Does not return/Halts, if error in CPU
    registers

Include:

Source file:   selfTest.asm

Header file:  selfTest.h, selftest.inc
```

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

Program Counter (Table 1 – 1.3)

The program counter register is part of the CPU register set. To test these registers explicitly using a checkerboard test, the addresses of 0x1555 and 0x2AAA must be allocated exclusively for this test. Because it is only a few bytes of flash, it appears to be a good investment of resources. However, this breaks the flash into segments of memory that are now smaller. This condition is not favorable for efficient compilation and can inhibit some programs from fitting into flash, although their absolute size is smaller than the actual flash size. See the ImageCraft C Compiler Guide, Section 7.2.1, or the PSoC Designer help files located at **Help > Documentation > Compiler and Programming Documents** > ***C Language Compiler User Guide.pdf*** for placing your code in absolute code location and avoiding errors because of these reserved flash addresses.

Function

```
BYTE   SelfTest_PC(void)

Returns:

0   No error

1   Error in PC

x   Does not return, if error in PC

Include:

Source file:   selfTest_pc.asm

Header file:  selfTest_pc.h,
selftest_pc.inc
```

To avoid this code fragmentation and to ensure proper PC functioning, use a watchdog timer (WDT), which can reset the system in case of a stuck PC value. Also, you can use a global variable with the WDT. This global variable is incremented in the main code by a predefined value at various places in the main code flow. The value is then monitored in an interrupt, such as a sleep timer interrupt service routine (ISR), for a proper increment/change from its previous value. If the change is too small or is less than expected, the code may be stuck somewhere. If this condition is detected, an infinite loop, which does not clear the WDT, can be entered to cause a soft watchdog reset.

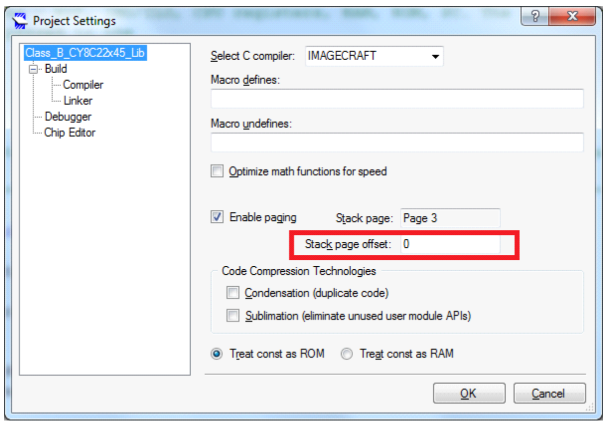
Stack Overflow Test

The stack is one of the key components, especially when there are many function calls, ISRs, and local variables in the code. During code execution, take care to ensure that the stack does not get full or overflow, or else it will lead to undesired code behavior. When a function is called or an interrupt occurs, the program counter value is pushed to the stack space. Also, local variables in the code are typically used from the stack space. Therefore, an overflow in the stack will result in data corruption of these variables or registers and in erroneous behavior.

In CY8C22x45 or any PSoC 1 family, the stack is a part of RAM, and by default, the last RAM page (one RAM page in PSoC 1 is equal to 256 bytes) is used for the stack (after execution of the boot code, when the stack resides in Page 0 regardless of the device). The starting address or offset of the stack in the RAM page on which it resides can be set in compiler settings, as shown in Figure 2. This setting is available at **Project** > **Settings** > **Compiler**. Regardless of this offset, the last address of the stack space is the address of the last byte in that particular RAM page where the stack resides.

To detect overflow, the last two bytes of the stack are reserved to hold special bytes of data. These special bytes are compared periodically, for example, in a timer interrupt, for any change in their value. If there is a change, it can be asserted that there was a stack overflow. At that point, stop code execution and reset the device, either by watchdog or external trigger.

Figure 2. Stack Offset Under Project Settings


Project Settings

Class_B_CY8C22x45_Lib
- Build
  - Compiler
  - Linker
- Debugger
- Chip Editor

Select C compiler: IMAGECRAFT

Macro defines:

Macro undefines:

☐ Optimize math functions for speed

☑ Enable paging   Stack page: Page 3

Stack page offset: 0

Code Compression Technologies

☐ Condensation (duplicate code)

☐ Sublimation (eliminate unused user module APIs)

[✓ Treat const as ROM]   Treat const as RAM

OK   Cancel

The following functions implement the stack overflow check:

```
void   SelfTest_StackOverflowChk_Init(void)
Returns:     None


BYTE   SelfTest_StackOverflowChk (void)
Returns:
0   No Error
1   Error detected
Include:
```

Source File: selftest_stackoverflow.asm

Header File: selftest_stackoverflow.inc, selftest_stackoverflow.h

You can modify the special bytes stored and the location of the special bytes in the stack page in the *selftest_stackoverflow.inc* header file, according to user requirements. The SelfTest_StackOverflowChk_Init() API initializes the last bytes of the stack to the user-specified special bytes. The SelfTest_StackOverflowChk() API checks the values stored in those bytes for any data corruption or stack overflow.

WDT Test

This function implements the watchdog functionality test (see Figure 3). The function starts the WDT and runs an infinite loop. If the WDT is working fine, it generates a reset. After the reset, the function analyzes the reset source. If the watchdog is the source of reset, the function returns to main code execution; otherwise, it waits for four sleep clock timer intervals and then returns watchdog fail/error. Because the WDT counter is three sleep timer ticks, a fourth tick without a reset would mean either WDT failure or a disabled state.

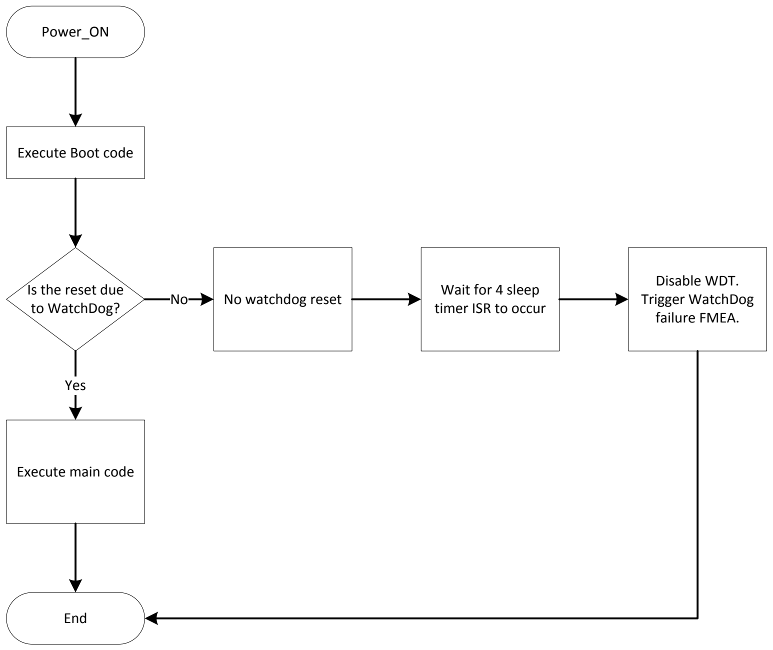
Function

```
BYTE   SelfTest_WDT(void)
Returns:
        0 - No Error
        1 - WDT disabled/Error
Include:
```

Source File: SelfTest_WDT.c

Header File: SelfTest_WDT.h

Figure 3. WDT Self-Test


Power_ON
Execute Boot code
Is the reset due to WatchDog?
No
No watchdog reset
Wait for 4 sleep timer ISR to occur
Disable WDT. Trigger WatchDog failure FMEA.
Yes
Execute main code
End

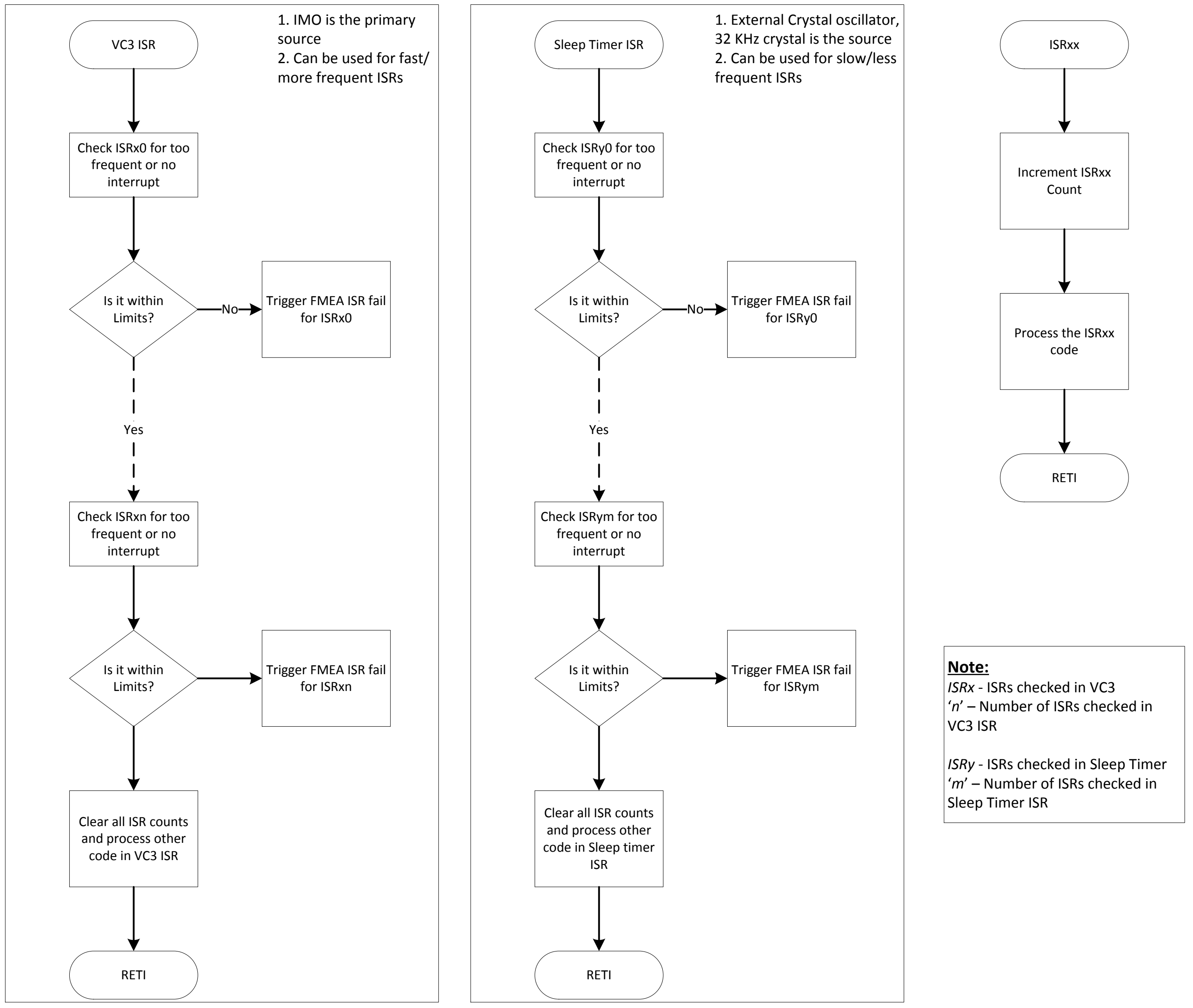
Interrupt Handling and Execution (Table 1 – 2)

Interrupts, which are integral to most embedded designs, require attention to ensure proper operation. An interrupt that occurs too often or too infrequently can compromise a system's safety and reliability. For instance a frequently occurring interrupt can utilize the CPU completely without giving the required/desired control to the main code. Similarly an interrupt that occurs infrequently can cause the system to lag or perform out of specification.

If interrupts occur frequently, enable the WDT. If the WDT is not refreshed because code execution is stuck inside the interrupt handler, then the WDT causes a reset.

In PSoC 1, Time-slot monitoring of the ISRs using one or two independent interrupt sources can be used to implement this safety feature. The VC3 divider ISR derived from IMO and/or the sleep timer ISR derived from an external crystal oscillator (ECO) or internal low-speed oscillator (ILO) can be used for time-slot monitoring. All the ISRs running in the system should have a count variable associated with them. A limit (number of minimum and maximum occurrence of each between two VC3 ISRs and/or two sleep timer ISRs) for each is defined in the form of macros. Whenever a VC3 or sleep timer ISR occurs, the number of occurrences of all the ISRs is checked for their limits and an error flag is raised, and/or the ISR is disabled in case of any discrepancies. Figure 4 shows the flow chart for implementing the same in PSoC 1.

Figure 4. Interrupt Self-Test – No or Too Frequent Interrupt Check


VC3 ISR
1. IMO is the primary source
2. Can be used for fast/ more frequent ISRs
Check ISRx0 for too frequent or no interrupt
Is it within Limits?
No
Trigger FMEA ISR fail for ISRx0
Yes
Check ISRxn for too frequent or no interrupt
Is it within Limits?
Trigger FMEA ISR fail for ISRxn
Clear all ISR counts and process other code in VC3 ISR
RETI
Sleep Timer ISR
1. External Crystal oscillator, 32 KHz crystal is the source
2. Can be used for slow/less frequent ISRs
Check ISRy0 for too frequent or no interrupt
Is it within Limits?
No
Trigger FMEA ISR fail for ISRy0
Yes
Check ISRym for too frequent or no interrupt
Is it within Limits?
Trigger FMEA ISR fail for ISRym
Clear all ISR counts and process other code in Sleep timer ISR
RETI
ISRxx
Increment ISRxx Count
Process the ISRxx code
RETI
Note:
ISRx - ISRs checked in VC3
'n' – Number of ISRs checked in VC3 ISR
ISRy - ISRs checked in Sleep Timer
'm' – Number of ISRs checked in Sleep Timer ISR

Clock (Table 1 – 3)

According to the IEC 60730 standard, failure modes resulting in the clock oscillating at harmonics and subharmonics of the set frequency need to be tested. To ensure that the clock is not oscillating at a harmonic or subharmonic, the clock accuracy limits are set to +/-10%. The clock test implements the 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). There are two types of implementation for this type of test. One is based on firmware and the other on hardware. The advantage of the firmware-based test is that it consumes no hardware resource, although it is less accurate than the hardware implementation. It is recommended to use a hardware timer if one is available. If not, use the firmware-based accuracy check.

Function (Firmware Based)

```
BYTE   SelfTest_IMO_Check(void)

Returns:  0(SelfTest_IMO_OK)      No error
          1 (SelfTest_IMO_ERR)    Error

Include:
```

Source File: `SelfTest_Clock.asm`

Header File: `SelfTest_Clock.inc, SelfTest_Clock.h`

Important Note
The clock limits are defined in the *SelfTest_Clock.inc* file. The method to calculate the numbers is described in the comment above the limits. You can accordingly calculate the accuracy and replace the values with their requirement.

Function (Hardware Based)

```
BYTE   SelfTest_Clock_HW(void)

Returns:  0(SELF_TEST_CLK_OK)    No error
          1 (SELF_TEST_CLK_ERR)  Error

Include:
```

Source File: `selfTest_clock_hw.c`

Header File: `selfTest_clock_hw.h`

The hardware-based function checks the IMO/ILO for proper frequency using a 16-bit HW timer user module named “IMO_Timer” with the following:

Period = 65535

Clock sync = Use Sysclk Direct

Interrupt = TerminalCount

Capture = Low

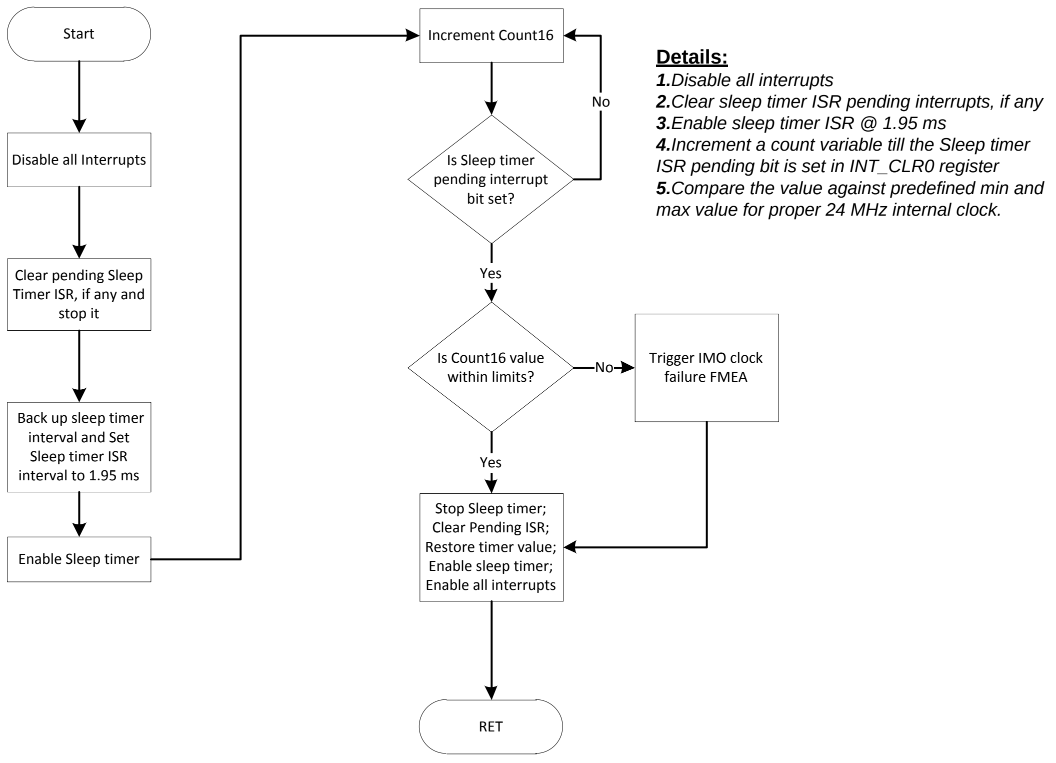
Other parameters are “Don't care.”

The limits for the clock accuracy are defined in *selftest_clock_hw.h* with an explanation in the comment defining them. The *selftest_clock_hw.c* file has function definitions for sleep timer ISR and IMO_Timer ISR. These ISRs must be placed in their respective locations in the *boot.tpl* file. If you have already defined the sleep timer ISR, then both of the ISRs need to be merged, and care must be taken to ensure that the clock test part of the code executes only when called from this function.

Important Note
Both the hardware- and firmware-based clock tests need a 32.768-kHz ECO to be present for strong accuracy.

Figure 5 shows the PSoC implementation of the test in firmware.

Figure 5. Clock Self-Test


Start
Disable all Interrupts
Clear pending Sleep Timer ISR, if any and stop it
Back up sleep timer interval and Set Sleep timer ISR interval to 1.95 ms
Enable Sleep timer
Increment Count16
Is Sleep timer pending interrupt bit set?
No
Yes
Is Count16 value within limits?
No
Trigger IMO clock failure FMEA
Yes
Stop Sleep timer; Clear Pending ISR; Restore timer value; Enable sleep timer; Enable all interrupts
RET
Details:
1.Disable all interrupts
2.Clear sleep timer ISR pending interrupts, if any
3.Enable sleep timer ISR @ 1.95 ms
4.Increment a count variable till the Sleep timer ISR pending bit is set in INT_CLR0 register
5.Compare the value against predefined min and max value for proper 24 MHz internal clock.

Flash (Invariable Memory, Table 1 – 4.1)

PSoC 1 includes 4 KB to 16 KB of flash on-chip memory. PSoC 1 flash memory is organized in blocks, with each one containing 64 data bytes. According to the IEC 60730 standard, an acceptable measure for Class B compliance is to detect all single-bit faults. A 16-bit checksum (sum of all bytes in flash until the address is specified) of the entire flash content is computed whenever requested and compared against the stored checksum value for any data corruption. The checksum is stored in last two bytes of the flash area. (To learn how to obtain the checksum for a program, see Appendix A.)

Function

```
BYTE   SelfTest_ROM(WORD noOfBlocks)
Arguments: WORD noOfBlocks;
```

Number of ROM blocks to be appended to the current checksum; the value is capped to flash size. If greater than flash size, checksum is calculated until the end of flash, and function returns the result of the test.

```
Returns:   0   No error
           1   Checksum not complete
           2   1-bit Error
Include:
Source File: selfTest.asm,
             selftest_ram_rom.c
Header File: selfTest.h,
             selftest_ram_rom.h
```

The *selftest.asm,‘selftest.inc,* and *selftest.h* files contain the lower level API, which calculates the checksum for a single block. The *selftest_ram_rom.c* and *selftest_ram_rom.h* files contain the API, which does the checksum verification for the entire flash by calling the low-level API for all the flash blocks. During run time, the user can call the function with a smaller number of bytes every self-test cycle to reduce the time spent on ROM calculation. The function return value can be monitored for any error or checksum completion status.

SRAM (Variable Memory, Table 1 – 4.2)

All PSoC 1 devices include on-chip SRAM. All the families offer devices that range from 512 bytes to 1 KB of RAM, which includes the stack. The entire RAM is split into pages of 256 bytes each, with the last 256 bytes or page allocated for the stack.

The Variable Memory test implements the Periodic Static Memory test H.2.19.6 defined by the IEC 60730 standard. It detects the 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 is one of the widely used static memory algorithms to check DC faults. You can implement the following tests using the Class B Safety Software Library provided with this application note:

- March C/C Minus test
- March B test

March Test

The March test performs 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’ to a memory cell)
w1	Indicate write operation (writes ‘1’ to a memory cell)

March C Test

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

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

This test is destructive, which means that memory contents are not saved. Therefore, it is designed to run at the system startup before initializing the memory and the runtime libraries.

March C Algorithm

The March C algorithm is implemented as follows:

1. Write ‘0’ to RAM cells being tested starting from first (ascending) or last (descending) RAM address.
2. Read ‘0’ from and write ‘1’ to RAM cells being testing starting from the first RAM address.
3. Read ‘1’ from and write ‘0’ to RAM cells being tested starting from the first RAM address.
4. Read ‘0’ from the RAM cells starting from either the first or the last RAM address.
5. Read ‘0’ from and write ‘1’ to RAM cells being tested from the last (to the first) RAM address.

6. Read ‘1’ from and write ‘0’ to RAM cells from the last RAM address.
7. Read ‘0’ from RAM cells from the first RAM address.

An error is reported if any read operation returns an error bit (‘1’ instead of ‘0’ and vice versa). The following notation explains the steps.

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

March C minus

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

The same functions used for March C can be conditional compiled to perform the March C minus test. By clearing the macro NOT_MARCH_C_MINUS available in the *selftest_ram_march_c.inc* file and then calling, the function will perform the March C minus test. Using the start address and end address pointers, you can do runtime March C/C minus tests by backing up the area of RAM under test to other parts of RAM not included in the test and then restoring the data. This test coves only the RAM area, not the stack area. If the start or end address is found to be out of the RAM boundary (excluding the stack size), then the test returns 0xFF. If the start and end address are equal, the test is done on the entire RAM area (excluding the stack). If the end address is less than the start address, the test rolls over to address ‘0’ at the end of the RAM area and continues the test until it reaches the end address.

Important Note
The test covers the RAM area from the start address to the (end address – 1)th location.

Function

```
BYTE   SelfTest_RAMMarchC_Periodic(BYTE
*startAddr, BYTE *endAddr);

Parameters:
BYTE *startAddr, BYTE *endAddr
Start and End address of the RAM area to
scanned

Returns:
0 (SelfTest_RAM_OK)                No error
2 (SelfTest_ERR_RAM_FAILURE)       Error
FFh (OUT_OF_BOUNDS)
```

Stack Page Test

```
BYTE   SelfTest_bSTKPG_Test_March_C(BYTE)

Returns:
0 (SelfTest_RAM_OK)                No error
2 (SelfTest_ERR_RAM_FAILURE)       Error


Include:
Source    File:    selftest_ram_march_c.asm,
            Selftest_ram_march_c_periodic.c,
            selftest_ram_march_low_lvl.asm;
Header File: selftest_ram_march_c.h,
              selftest_ram_march_c.inc,
            Selftest_ram_march_c_periodic.h,
            selftest_ram_march_low_lvl.h;
```

March B Test

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

- Stuck-at
- Linked idempotent coupling
- Inversion coupling

March B Algorithm

MarchB

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

Function

```
BYTE   SelfTest_bRAM_Test_March_B(void)
Returns:
0 (SelfTest_RAM_OK)               No error
2 (SelfTest_ERR_RAM_FAILURE)      Error
```

Stack Page Test

```
BYTE   SelfTest_bSTKPG_Test_March_B(void)
Returns:
0 (SelfTest_RAM_OK)               No error
2 (SelfTest_ERR_RAM_FAILURE)      Error
```

```
Include:
Source File: selftest_ram_march_b.asm
Header File: selftest_ram_march_b.h
```

SelfTest_RAMMarchC_Periodic, SelfTests_bRAM_Test_March_B—One or both of these functions can be chosen and called immediately after the PSoC start. These functions check the RAM destructively. After passing the test, these functions reset the PSoC RAM to '0'. The stack page in RAM is tested by storing the top three bytes of the stack in temporary registers (backing up the program counter value to return execution properly). Because only the top three bytes of the stack are saved, the user needs to take care that the function is called from main. In other cases, the user needs to back up the required amount of the stack depending on the depth of the calling function. Also, it is recommended that interrupts be disabled before calling the stack check function, as interrupts use the stack page. If the function is called in either case—interrupt enabled or called from a function other than main—the stack test will clear the entire stack except the first three bytes, which will result in the calling function not returning at all. In the case of a fault, the function returns '2', or if the first three bytes are corrupted, the function halts and does not return. A WDT can be used as a reset mechanism in such scenarios.

Digital I/O (Table 1 – 7.1)

The PSoC I/Os provide the following:

- Digital input sensing
- Digital output drive
- Pin interrupts
- Connectivity for analog inputs and outputs
- Connectivity for LCD segment drive
- Access to internal peripherals:
  - Directly for defined ports
  - Through the global input/output bus inside PSoC

The 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, I^2C port, crystal oscillator). Special functions are enabled using control registers associated with the specific functions.

The test goal is to ensure that I/O bits are not stuck at '1' or '0'. The I/O test tests all the I/O. It is done by successive writing, reading, and checking 0x55 and 0xAA values into the I/O registers.

Important Note
This test is application-dependent and can be run only if the hardware allows it.

Because writing to and reading from the I/O registers can cause unwanted behavior on the I/O pin states, this test needs to be application dependent. I/O states can be periodically read and compared to the expected values during runtime to do the plausibility check.

A/D Converter (Table 1 – 7.2.1)

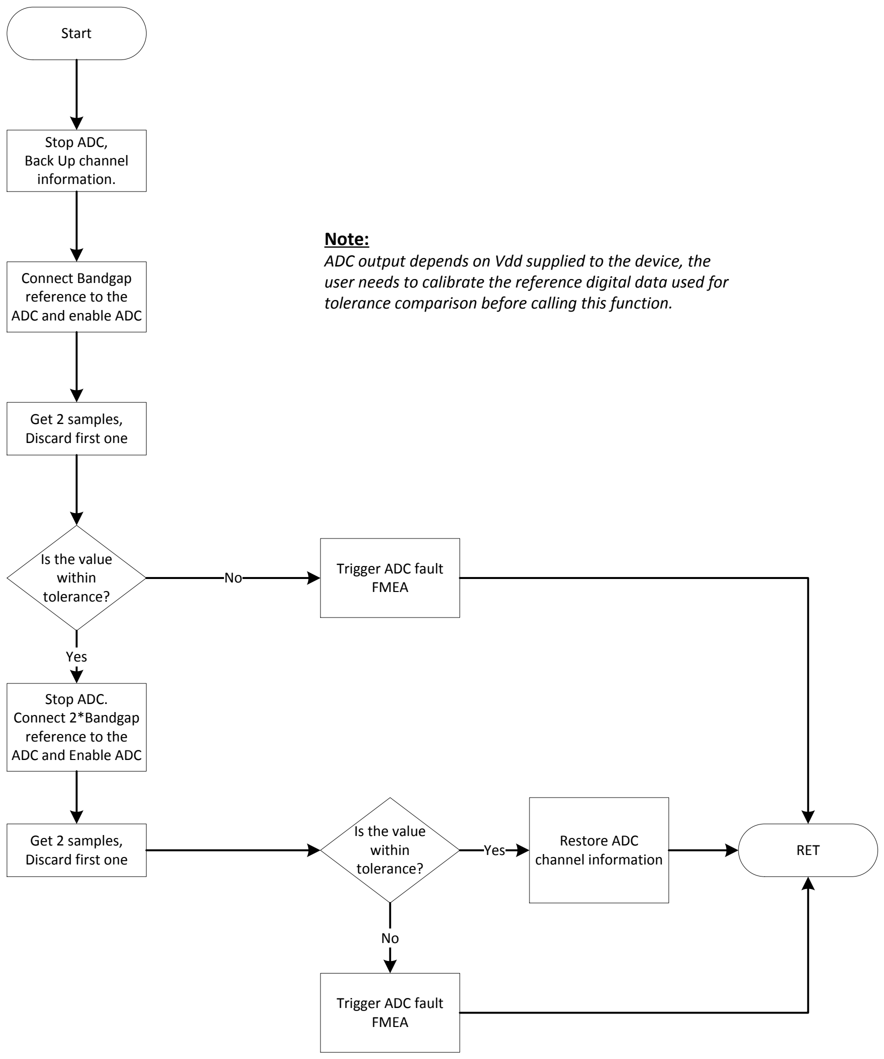
This function tests the SAR ADC in the CY8C22x45 and CY8C21x45 family of devices. In this test (see Figure 6), the SAR input is temporarily connected to bandgap reference (Vbg) and 2 times bandgap reference (2Vbg), and the ADC result is tested for plausibility. The test is a success if the digitalized input voltage value is equal to the required reference voltage value to within the defined accuracy. When the test is a success, the function returns 0; otherwise, it returns 0xFF.

Function

```
BYTE   SelfTest_SAR_ADC(void)

Returns:

0      No error
0xFF   Error detected

Include:

Source File: selftest_sar_adc.asm
Header File: selftest_sar_adc.h,
             selftest_sar_adc.inc
```

The ADC accuracy is defined in *selftest_sar_adc.inc*. Based on supply voltage, the value changes as the SAR ADC input range is fixed (it is from Vss to Vdd).

Figure 6. Self-Test SAR ADC


Start
Stop ADC, Back Up channel information.
Connect Bandgap reference to the ADC and enable ADC
Get 2 samples, Discard first one
Is the value within tolerance?
No
Trigger ADC fault FMEA
Yes
Stop ADC. Connect 2*Bandgap reference to the ADC and Enable ADC
Get 2 samples, Discard first one
Is the value within tolerance?
Yes
Restore ADC channel information
RET
No
Trigger ADC fault FMEA
Note:
ADC output depends on Vdd supplied to the device, the user needs to calibrate the reference digital data used for tolerance comparison before calling this function.

Additional Safety Measures

The following routines are additional safety routines that are not directly required by the IEC 60730 standard but are useful to ensure reliable operation of the system.

Comparator

This function implements the comparator functionality test. To execute it, connect the bandgap reference to one terminal and Vss to another terminal (internally). Then check that the comparator output is working properly. The test is then repeated by swapping the inputs. When the test is a success, the function returns 0, SelfTest_COMP_OK; otherwise, it returns 1, SelfTest _COMP_ERR.

Function

```
BYTE   SelfTest_Comparator(void)
```

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

Source File: `SelfTest_comparator.c`

Header File: `SelfTest_comparator.h`

The test requires you to place the comparator user module with the name “CMP” in the project or change the reference “CMP” name in the *selftest_comparator.c* and *selftest_comparator.h* files to be able to use the function properly. The CMP_BUS_BIT available in *selftest_comparator.h* should be modified per the column in which the comparator is placed.

The test function saves all the user module configurations and nonretention registers before testing. They are restored afterward. You have to call the CMP_Start() API after the test is complete, because it stops the comparator after completing the test.

Communications UART (Table 1 – 6.1/6.2)

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

Function

```
BYTE   SelfTest_UART_LoopBack(BYTE parity)
```

```
Arguments: BYTE parity
```

```
          Parity with which UART needs to
          enabled after test; Pass 0xFF to
          disable UART after test.
```

```
Returns:
0   No error
1   Error detected
2   Test not done
```

```
Include:
```

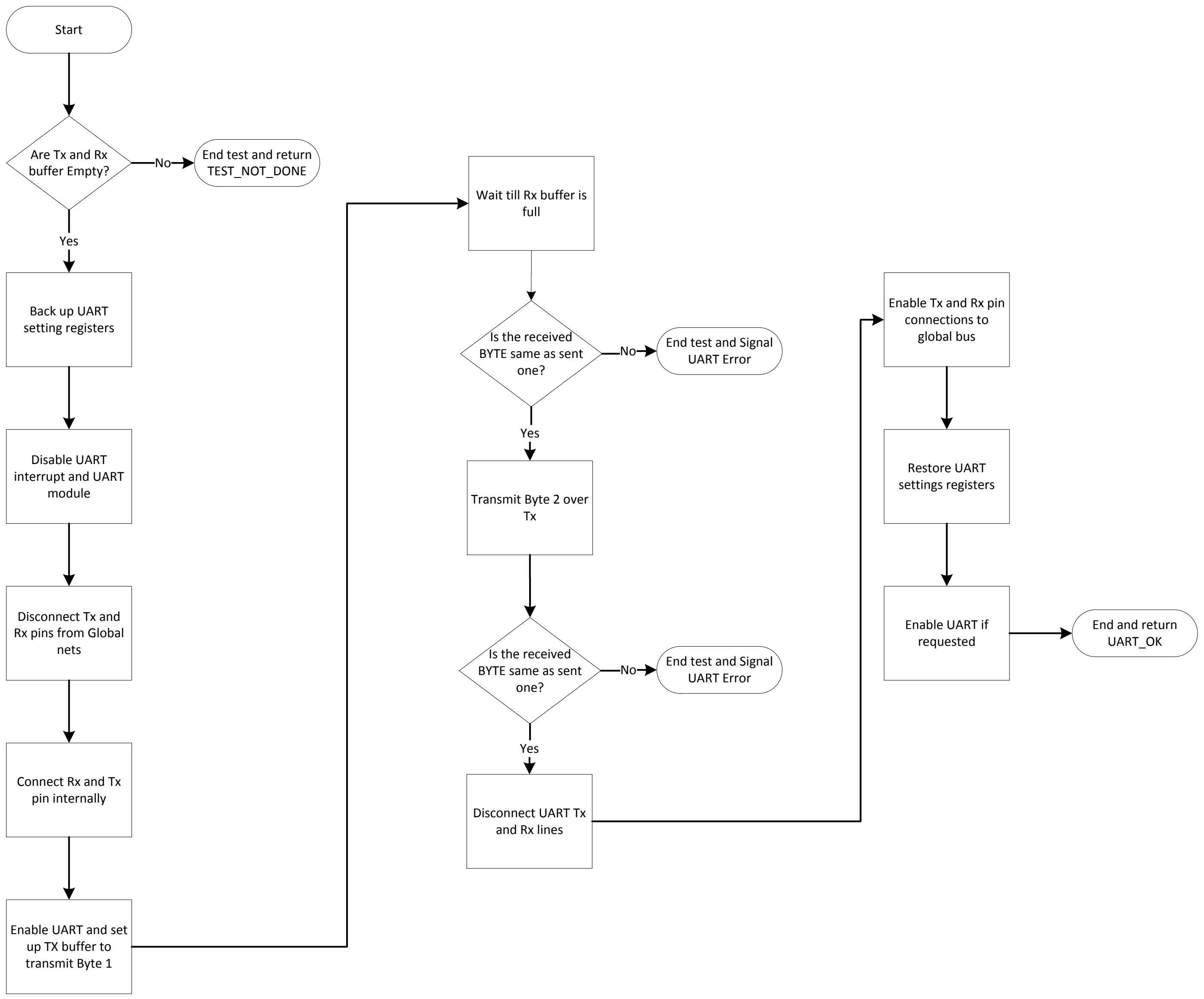
Source File: `SelfTest_UART.c`

Header File: `SelfTest_UART.h`

Figure 7 shows this test PSoC implementation. The UART Tx and Rx lines are shorted internally during the test. If there are any bytes in the receive buffer, the test is not executed and it returns 2, indicating that the test was not done because of a full Rx or Tx register.

The test function saves all the user module configuration registers before testing and restores them afterward. During the test, this function transmits two special checkerboard bytes (0x55 and 0xAA) over the Tx line and checks the data received over the Rx line. The function requires you to provide details, such as the port global select register and the port pin number of the Rx and Tx pins. These details are used to connect and disconnect the pins during the test and are available in the *selftest_uart.h* file. This function can be called periodically to ensure that the UART line is functioning properly.

Figure 7. UART Loopback Self-Test



Start
Are Tx and Rx buffer Empty?
No
End test and return TEST_NOT_DONE
Yes
Back up UART setting registers
Disable UART interrupt and UART module
Disconnect Tx and Rx pins from Global nets
Connect Rx and Tx pin internally
Enable UART and set up TX buffer to transmit Byte 1
Wait till Rx buffer is full
Is the received BYTE same as sent one?
No
End test and Signal UART Error
Yes
Transmit Byte 2 over Tx
Is the received BYTE same as sent one?
No
End test and Signal UART Error
Yes
Disconnect UART Tx and Rx lines
Enable Tx and Rx pin connections to global bus
Restore UART settings registers
Enable UART if requested
End and return UART_OK

Communications SPI

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

Function

```
BYTE   SelfTest_SPIS_LoopBack(BYTE mode)
```

```
Arguments: BYTE mode
```

```
          Mode with which SPI slave has to
          be tested and started again, if
          0xFF is passed then SPI is tested
          in mode 2 and disabled after
          test.
```

```
Returns:
0  No error
1  Error detected
2  Test not done
```

```
Include:
```

Source File: `SelfTest_SPI.c`

Header File: `SelfTest_SPI.h`

The PSoC implementation for SPI loopback is similar to the one explained for UART. The SPI slave MOSI and

MISO lines are shorted internally during the test and VC2 is used as SCLK source. If there are bytes in the receive buffer, the test is not executed and it returns 2, which indicates the test was not done because of a full receive or transmit register.

The test function saves all the user module configuration registers before testing and restores them afterward. During the test, this function transmits two special checkerboard bytes (0x55 and 0xAA) over the MISO line and checks the data received over the MOSI line. The function requires you to provide details, such as the port global select register and port pin number of the MOSI and MISO pins. These details are used to connect and disconnect the pins during the test and are available in the *selftest_spis.h* file. This function can be called periodically to ensure proper SPI slave functioning.

CRC-8 for Communications Protocols

This function implements a firmware 8-bit cyclic redundancy check (CRC) algorithm, which can be used by communications protocols to perform CRC-8. The same function can be used to verify CRC-8 checksum as well. The flowchart to perform CRC-8 using this function is shown in Figure 8. The calculated 8-bit CRC is available in the global variable “rc_checksum” in the file and can be accessed by including the *crc.asm*, *crc.inc*, and *crc.h* files in the project. The CRC polynomial is provided as a macro in the *crc.inc* file, which you can modify per the polynomial requirement. The comments near the macro describe how to define the polynomial.

To pass Class B compliance for external communication, a hamming distance of 3 needs to be satisfied, which means an error detection capability of up to two bits. The polynomial of 0x14D (or 0xA6 in implicit “+1” format) supports a hamming distance of 3 for a data length up to 246 bits, excluding 8-bit CRC checksum, as explained in Table 4 of the paper titled Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks. This can be enough for many applications involving up to 29 bytes of data, 1 byte of address, and 1-byte checksum. You can use the CRC16 User Module available in PSoC Designer to implement a better hamming distance error check code or to support packet lengths above 246 bits for hamming distance 3, required for Class B compliance.

Function

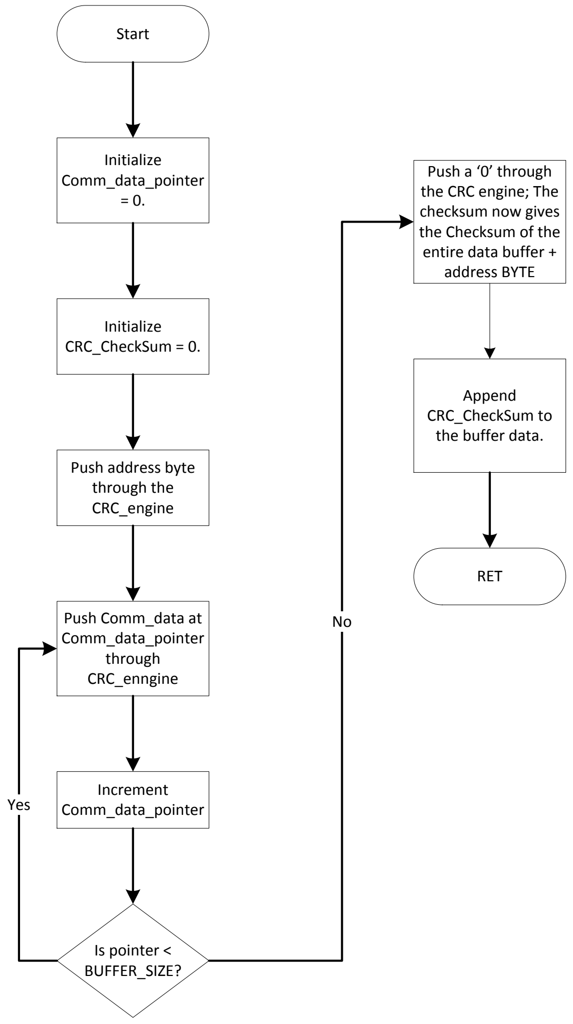
```
void   CRC_Calculate(BYTE)
```

```
Arguments: BYTE crc_nextByte

          Next Byte to be pushed into the
          CRC engine.

Returns:  None

Include:
```

Source File: `crc.asm`

Header File: `crc.inc, crc.h`

Figure 8. CRC-8 for Communication Interface


Start
Initialize Comm_data_pointer = 0.
Initialize CRC_CheckSum = 0.
Push address byte through the CRC_engine
Push Comm_data at Comm_data_pointer through CRC_enngine
Increment Comm_data_pointer
Is pointer < BUFFER_SIZE?
Yes
No
Push a '0' through the CRC engine; The checksum now gives the Checksum of the entire data buffer + address BYTE
Append CRC_CheckSum to the buffer data.
RET

CSD

The CY8C22x45 and CY8C21x45 family of devices have a capacitive sensing feature called CapSense. It allows users to activate touch-sensitive buttons, sliders, and wheels using the capacitance of their fingers. Touchpads and touchscreens are common examples of capacitive sensing interfaces. The underlying principle of these technologies is the measurement of capacitance between a plate (the sensor) and its environment. To learn more about Cypress's CapSense technology, see Getting Started with CapSense.

This section demonstrates how to use the unique hardware reconfiguration possibilities available in PSoC devices to detect errors in capacitive sensing measurements at run time. The errors include the following:

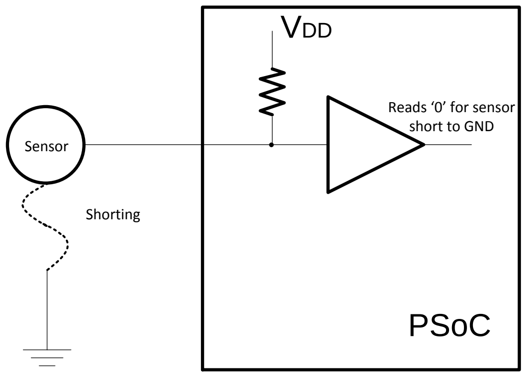
- Shorts to V_{CC} or ground
- Sensor-to-sensor shorts
- Sensor disconnects
- Faults in the sigma-delta modulator external components:
  - Modulation capacitor (C_{mod})
  - Discharge resistor (R_b) or current DAC

Use these diagnostics to provide fail-safe functions in capacitive sensing devices, where sensor faults lead to safety concerns. White goods, automotive, and industrial electronic applications are examples of the devices that require a sensor fault diagnosis for safe operation.

Test on Sensor Shorts

In normal operating conditions, the sensor-to-ground, sensor-to-sensor, and sensor-to-V_{CC} resistances are very high. To detect any shorts, actual resistance values are compared to the PSoC internal pull-up resistors. Figure 9 shows a simplified schematic for sensor-to-ground short detection.

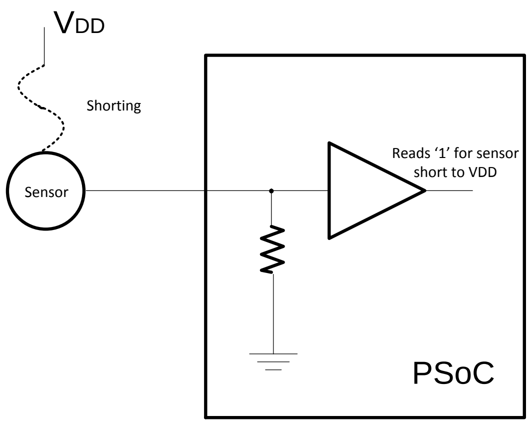
Figure 9. Sensor-to-Ground Short Detection Schematic



The sensor pin is configured in the resistive pull-up drive mode. In normal conditions, the CPU reads a logic 1 because of the pull-up resistor and due to the high impedance of the sensor input. Due to a fault, if the sensor is shorted to ground, the CPU will read a logic 0 on the input and can recognize this as a fault.

Sensor-to-V_{CC} shorting is detected by a similar method. Figure 10 shows the corresponding schematic.

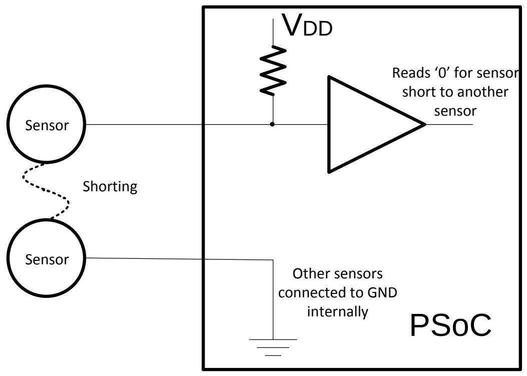
Figure 10. Sensor-to- V_{CC} Short Detection Schematic



In this case, the sensor pin is configured in the resistive pull-down drive mode. The input level is 0 in normal conditions and a logic HIGH if the sensor is shorted to V_{CC}.

Figure 11 shows the schematic for the sensor-to-sensor short check.

Figure 11. Sensor-to-Sensor Short Detection Schematic


VDD
Reads '0' for sensor short to another sensor
Sensor
Shorting
Sensor
Other sensors connected to GND internally
PSoC

All of the sensor pins are connected to ground internally by writing '0' to the data registers in the strong drive mode. Every sensor input is now individually configured to pull-up mode, and the level on the sensor input is read. If the sensor is good, then a logic HIGH will be read. If the sensor is shorted to any other sensor, then a logic 0 will be read. The following function does all the short tests mentioned in this section.

Function

```
BYTE   SelfTest_CSD_CheckShorts(void)
```

```
Returns:
'0'    No error
'N'    Error detected, N - number of
       sensor with error
```

```
Include:
```

Source File: `SelfTest_CSD.asm`

Header File: `SelfTest_CSD.h`

Important Note
The CSD User Module must be named the CSD to use the functions as such and the table CSD_Shade_Table_Short_Test in selftest_csd.asm file should be replaced with the table CSD_Shade_Table from CSD.asm file. This should be done whenever sensor pin or number of sensors is changed.

The function `SelfTest_CSD_CheckShorts` () is called to detect sensor shorts to ground, V_{CC}, or other sensors.

Test on Sensor Disconnect

Another task is to detect the sensor disconnects. Unlike detecting shorts, this task cannot be accomplished with hardware tricks. The detection method is based on observing the sensor baseline data. If a sensor is disconnected from the PSoC, the area of copper attached to the sensing IC is smaller than expected. This leads to a significantly lower raw count and baseline values.

To detect the sensor disconnects, store the baseline values under normal conditions in the internal EEPROM. The actual baseline value is compared to the stored value. If the actual value is less than the stored value and the difference exceeds the threshold, a system fault is detected. If the threshold value is too small, you can get false fault triggering. That occurs when there is a small change in baseline due to environmental changes. In addition, this value must not be too large or you will not have a reliable disconnect detection. This implementation provides two predefined threshold values: 25% and 12.5% of the stored baseline value. These values are sufficient for most applications. Choose the actual value in your system by running tests on real boards in real systems.

Function

```
BYTE   SelfTest_CSD_CheckBaselines(void)
```

Returns:

'0'	No error
'N'	Error detected, N - number of sensor with error

Include:

Source File: `SelfTest_CSD.asm`

Header File: `SelfTest_CSD.h`

Important Note
The CSD User Module must be named the CSD to use the functions as such.

The function `SelfTest_CSD_CheckBaselines()` is called to detect the sensor disconnection.

Modulator Component (C_{mod} and R_b) Testing

The sigma-delta modulator in the CSD User Module in bleed resistor (Rb) mode uses two external components: a modulation capacitor (C_{mod}) and a discharge resistor (R_b). These components can also cause errors in the capacitance measurement. For example, these components could be shorted or opened.

The simplest method to test both C_{mod} and R_b simultaneously is to estimate the RC time constant by charging C_{mod} through R_b. This measurement requires a minimal hardware reconfiguration and is easy to implement in software.

If C_{mod} is completely discharged and then charged to V_{cc} through R_b, the voltage on the capacitor changes according to Equation 1.

$$V_c(t) = V_{cc}(1 - e^{-t/\tau}) \qquad \text{Equation 1}$$

In this equation, the time constant (τ) = R_bC_{mod}.

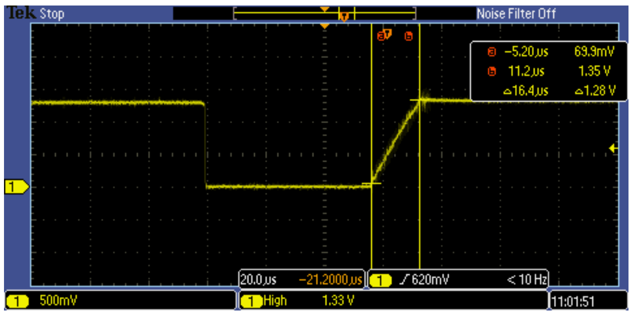
The capacitor charges until its voltage reaches the comparator reference voltage. This reference voltage depends on the CSD Component settings. In the example case, the reference source is bandgap (VBG) and the reference voltage is 1.3V. The time needed to charge the capacitor from 0 to 1.3V with V_{cc} = 5V is shown in Equation 2 and 3.

$$t = \ln(4)\cdot\tau \qquad \text{Equation 2}$$

$$t \approx 1.39\cdot\tau \qquad \text{Equation 3}$$

The oscilloscope image in Figure 12 shows the proposed RC test method. In the first stage, C_{mod} is completely discharged and in the second stage it is charged to V_{ref}. The measured charge time is about 16 µs. This completely agrees with the previous equations for C_{mod} = 10 nF and R_b = 5.1 k, as shown in Equation 4.

Figure 12. Voltage on C_{mod} When Charged Through Rb



$$t = 0.3\ .\ 5.1k\ .\ 10nF = 15.3\ \mu s \qquad \text{Equation 4}$$

In software, it is more convenient to measure the charge time in the CPU cycles, as shown in Equation 5

$$N_{CLK} = 0.3\ .\ R_bC_{mod}\ .\ CPU\_Clock \qquad \text{Equation 5}$$

For example, if C_{mod} = 10 nF, R_b = 5.1 k, and CPU_Clock = 12 MHz, the measured charge time is N_{CLK} = 183 CPU cycles.

Important Note
C_{mod} and R_b must be located on the correct pins in the wizard settings in PSoC Designer. If they are not, an error is reported.

Modulator Component (C_{mod} and IDAC) Testing

This test is similar to the C_{mod} and R_b test, except a predetermined current is used to charge C_{mod} to a predetermined voltage, and the charge time is calculated based on CPU_Clock cycles.

When the current is constant (I_d), the rate at which C_{mod} charges is linear and is proportional to the current and value of C_{mod}. If the final voltage on C_{mod} is fixed to V_{ref}, then the relationship between current (I_d), voltage (V_{ref}), capacitor (C_{mod}), and time (t) is shown in Equation 6.

$$V_{ref} = I_d * t / C_{mod} \qquad \text{Equation 6}$$

Based on Equation 6, "t" can be calculated, and similar to Equation 5, the number of CPU clocks can be derived.

Function

```
BYTE   SelfTest_CSD_CheckMod (void)
Returns:  0 - No error
          1 - Error detected
Include:
```

Source File: `SelfTest_CSD.asm`

Header File: `SelfTest_CSD.h`

Important Note
The CSD User Module must be named the CSD to use the functions as such.

The function `SelfTest_CheckMod()` is called to detect an error in C_{mod} and R_b/IDAC working. The function will automatically compile for R_b/IDAC based on the modulator combination selected in the project.

Important Note
The CSD User Module in PSoC Designer 5.3 supports built-in FMEA. The code provided with this application note is similar to the UM FMEA/Self-Test code. This code can be used with previous Designer versions.

Checkerboard RAM Test

This is an additional RAM test that can help to detect stuck-at fault in RAM areas without destroying them. But this test does not cover coupling faults. The checkerboard RAM test writes the checkerboard patterns (0x55 and 0xAA) to a sequence of memory locations. This is a nondestructive memory test. This test performs the following major tasks:

1. Saves the contents of the memory locations to be tested in temporary registers (TMP_DRx).
2. Writes the binary value 10101010 to the memory location 'N', and the inverted binary value, 01010101, to the memory location '~N'.
3. Reads the content of the memory location 'N' and checks if it is 10101010; it then checks 01010101 in memory location '~N'.
4. Restores the data from the temporary registers
5. Steps 2 and 3 are repeated for the entire content in the memory page.
6. After the test is completed, it is repeated starting from the last address and in descending order.
7. After a memory page is completed, the test of the next page starts until all the memory pages defined are tested.

If an error is detected in the temporary registers used in the checkerboard RAM test, avoid those corresponding tests to prevent loss of RAM data.

Function

```
BYTE SelfTest_RAM_CheckerBoard(void)
Returns:
    0 (SelfTest_RAM_OK)           No error
    2 (SelfTest_ERR_RAM_FAILURE)  Error
```

```
Include:
Source File: selfTest.asm,
                selftest_ram_rom.c
Header File: selfTest.h, selftest.inc,
                selftest_ram_rom.h
```

The `SelfTest_RAM_Checkerboard` can be called at any time. This function checks the RAM without destroying the data. This function should be called only if the `SelfTest_CPU()` returns 0.

Redundant Inverse Storage of Safety Variable

The RAM tests provided and the ones mandated for Class B are good at detecting bad RAM cells, whether they are stuck to a value or coupled to nearby cells. But when it comes to program execution, the danger with RAM is variable corruption because of unexpected ISRs, poor coding, or even poor optimization by the compiler. There is no standard or test to check variable corruption. This section provides a brief description of how it can be done.

To do this test, each global variable (or at least the key variables) will require a shadow variable associated with it. So it practically doubles the RAM used for global variables. But, at times, this can be very handy and a much needed RAM test compared to the March algorithms. This test can be carried out in two ways:

Check Before Access

This method provides full protection against corruption but it increases flash usage and execution time

1. The shadow variable is defined at a remote memory location from the global variable being protected, maybe in a different page for better protection against corruption.
2. Whenever the global variable is written, the shadow variable is updated with a bit-inverted value of the global variable.
3. Before the global variable is accessed for read, the values of the global variable and shadow variable are bitwise XORed. If the result is 0xFF (all 1's) the variable is not corrupt. Otherwise, declare error and process the necessary error handling, which can be a simple program reset..

Periodic Variable Check

This method provides partial protection against corruption, but with less impact on the flash size and execution time.

1. The shadow variable is defined at a far-away memory location from the global variable being protected, maybe in a different page for better protection against corruption.
2. Whenever the global variable is written, the shadow variable is updated with a bit-inverted value of the global variable.
3. A periodic check function can be implemented, which runs once in the while(1) loop and keeps a check on the variable by bit-wise XORing with its shadow and checking the result for 0xFF. In case of error, simple error handling, such as resetting the system, can be triggered.

Example code accompanies this application note in which a simple function is used to check variable corruption of a BYTE variable using its inverse storage.

Summary

This application note described how to implement diagnostic measures proposed by the IEC 60730 standard. The introduction of IEC 60730-1 into the design of white goods and other appliances adds a new level of safety for consumers. By taking advantage of PSoC 1, design teams can comply with regulations while maintaining or reducing the cost of electronic systems. The use of PSoC and the Class B Safety Software Library allows you to create a strong system-level development platform and achieve superior performance, fast time-to-market, and energy efficiency.

References

1. IEC 60730 Standard, “Automatic Electrical Controls for Household and Similar Use,” IEC 60730-1 Edition 4.0, 2010-03.
2. IEC 60335 Standard, “Household and Similar Electrical Appliances – Safety,” IEC 60335-1 Edition 5.0, 2010-05.
3. Koopman, P. & Chakravarty, T., “Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks,” DSN04, June 2004.

About the Author

Name:	Meenakshi Sundaram Ravindran
Title:	Senior Applications Engineer
Background:	The author holds a Bachelor’s of Engineering degree in Electronics and Communication from College of Engineering, Guindy, Chennai (India). He works on PSoC 1, CapSense, and motor control solutions at Cypress.
Contact:	msur@cypress.com

Appendix A: How to Obtain the Checksum of Flash (Invariable Memory)

Step 1:
Browse to the root folder of the PSoC Designer project and locate the *.hex* file.

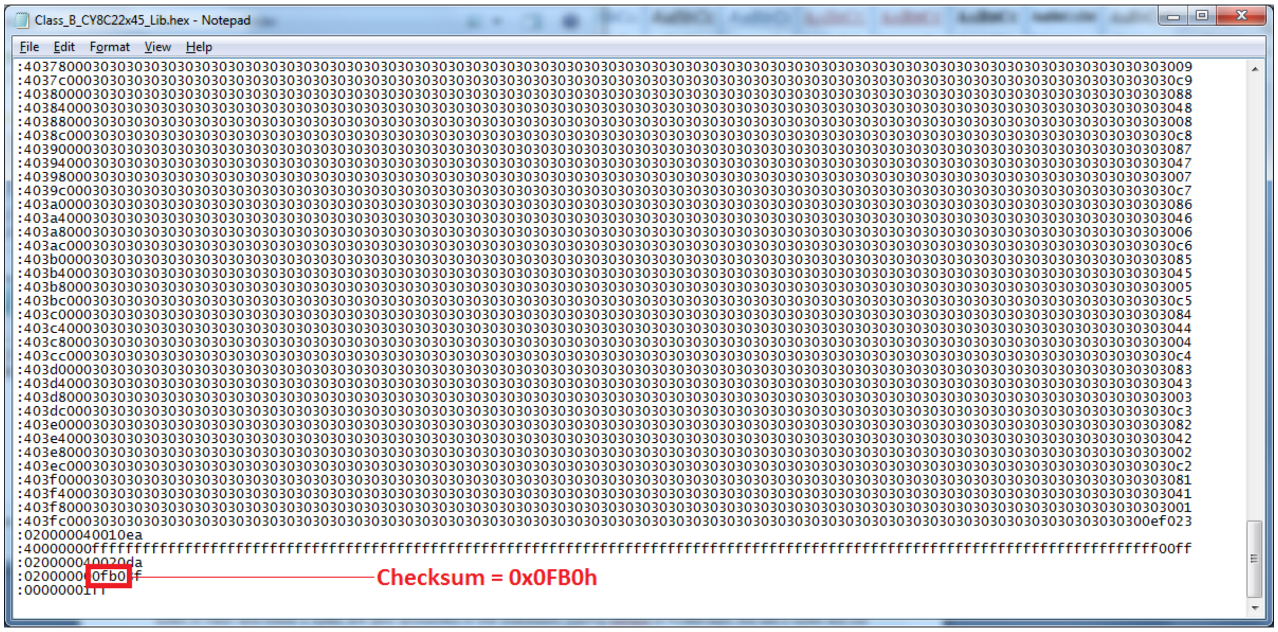
Step 2:
Open the *.hex* file in any text editor.

Step 3:
Place the following code in any *.c* file in the project to place the checksum in the last two bytes of flash; the address of the second-to-last byte for a 16-KB flash device is 0x3FFE.

```
#pragma lit_abs_address:<address of second last BYTE in flash>
__flash WORD checksum = 0x0000h;
#pragma end_abs_address
```

Step 4:
Go to the end of the file and locate the checksum, as shown in Figure 13.

Figure 13. CheckSum location in the *.hex* file


Class_B_CY8C22x45_Lib.hex - Notepad
File Edit Format View Help
Checksum = 0x0FB0h

```
:403780003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303009
:4037c0003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303c9
:403800003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303088
:403840003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303048
:403880003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303008
:4038c0003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303c8
:403900003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303087
:403940003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303047
:403980003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303007
:4039c0003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303c7
:403a00003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303086
:403a40003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303046
:403a80003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303006
:403ac0003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303c6
:403b00003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303085
:403b40003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303045
:403b80003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303005
:403bc0003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303c5
:403c00003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303084
:403c40003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303044
:403c80003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303004
:403cc0003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303c4
:403d00003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303083
:403d40003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303043
:403d80003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303003
:403dc0003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303c3
:403e00003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303082
:403e40003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303042
:403e80003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303002
:403ec0003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303c2
:403f00003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303081
:403f40003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303041
:403f80003030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303001
:403fc000303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030303030300ef023
:020000040010ea
:40000000ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff00ff
:020000040020da
:020000000fb03f
:00000001ff
```

Step 5:
Replace the checksum value “0x0000h” entered in step 3 with the found checksum in step 4 as follows.

```
#pragma lit_abs_address:<address of second last BYTE in flash>
__flash WORD checksum = <checksum_value>;
#pragma end_abs_address
```

For asm:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Org <address of second last BYTE in flash>
DW <checksum_value>
Area <flash area to be switched to for user code>
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Note After placing the code, the checksum observed in step 3 will change because the code in step 4 places two different bytes at the end of the flash area, and these two bytes are also accounted in that checksum. During a self-test of ROM/flash memory, the last two bytes are not included in the checksum calculation. Therefore, the calculated checksum and this stored checksum need to match for proper ROM.

When using EEPROM UM or CSD self-test routines, care should be taken while calculating the checksum in this way. In such scenarios, after the address of last byte until which checksum needs to be calculated is set, the rest of the flash area must be written with ‘0’ to obtain the proper checksum through the method described. By default, ‘0x30h’ (the halt instruction) is written in unused flash bytes. Or checksum can be manually calculated by subtracting (0x30h * number of bytes not used for checksum calculation) from the found checksum.

Appendix B: IEC 60730-1 Table H.11.12.7

API	Source File(s)	Header File(s)	Arg (s)	Return	Fault Detect Time (for CPUCLK = 12 MHz)	Class B Component Tested

SelfTest_CPU_Registers	Selftest.asm	Selftest.h; Selftest.inc;	-	BYTE:
				0 – No Err;
				1 – TMP_DR Err;
				Halt – CPU register Err;
					45 µs	CPU Registers except PC

SelfTest_PC	Selftest_pc.asm	Selftest_pc.inc; Selftest_pc.h;	-	BYTE:
				0 – PC proper;
				1 – Special Calculation error – PC might be corrupt;
				Does not return/halt – PC Err
					~12.5 µs	CPU Register – Program Counter (Island test)

SelfTest_IMO_Check	Selftest_clock.asm	Selftest_clock.h;
		Selftest_clock.inc
			-	BYTE:
				0 – No Err;
				1 – ILO/IMO Err;
					~2 ms	Clock

SelfTest_Clock_HW	Selftest_clock_hw.c	Selftest_clock_hw.h	-	BYTE:
				0 – No Err;
				1 – ILO/IMO Err;
					~2 ms	Clock (using HW timer)

SelfTest_ROM	Selftest.asm;
	Selftest_ram_rom.c
		Selftest.h; Selftest.inc ;
		Selftest_ram_rom.h;
			WORD noOfBlocks; Number of blocks to be appended to checksum;	BYTE:
				0 – No Err;
				1 – Not complete;
				2 – CheckSum mismatch;
					~110 ms for 16k	Flash (invariable memory)

API	Source File(s)	Header File(s)	Arg (s)	Return	Fault Detect Time (for CPUCLK = 12 MHz)	Class B Component Tested
SelfTest_RAM_CheckerBoard	Selftest.asm;	Selftest_ram_rom.c; Selftest.h; Selftest.inc ;	-	BYTE: 0 – No Err;	~50 ms for 1024 bytes	
SelfTest_RAMMarchC_Periodic	Selftest_ram_march_c_periodic.c; Selftest_ram_march_low_lvl.asm;	Selftest_ram_march_c_periodic.h; Selftest_ram_march_low_lvl.h;	BYTE *startAdr; BYTE *endAddr; Start and end address of RAM area to be tested	2 – RAM Err; 0xFF – Ram address Out of bounds	~76 ms for 768 bytes	
SelfTest_bSTKPG_Test_March_C	Selftest_ram_march_c.asm;	Selftest_ram_march_c.h; Selftest_ram_march_c.inc;	-		~3.5 ms for 256 bytes	
SelfTest_bRAM_Test_March_B	Selftest_ram_march_b.asm;	Selftest_ram_march_b.h;	-		~15 ms for 768 bytes	
SelfTest_bSTKPG_Test_March_B					~5 ms for 256 bytes	RAM (variable memory)
SelfTest_SAR	Selftest_SAR_ADC.asm	Selftest_SAR_ADC.inc; Selftest_SAR_ADC.h;	-	BYTE: 0 – No Err; 0xFF- SAR Err;	~70 µs	A/D Converter
SelfTest_Comparator	SelfTest_Comparator.c	SelfTest_Comparator.h	-	0 – No Err; 1 – Err;	~42 µs	Comparator
SelfTest_UART_LoopBack	SelfTest_UART.c	SelfTest_UART.h	BYTE: Parity; 0 – No Parity 2 – Even; 6 – Odd Parity; 0xFF –	BYTE: 0 – No Err; 1 – Err; 2 – Test Not done;	~2.3 ms @ 93.75 ksps	External Communication; Protocol Test

API	Source File(s)	Header File(s)	Arg (s)	Return	Fault Detect Time (for CPUCLK = 12 MHz)	Class B Component Tested
			Disable UART			
SelfTest_SPIS_LoopBack	SelfTest_SPI.c	SelfTest_SPI.h	BYTE: SPI Mode; 0xFF – Disable SPIS after test		~360 µs @ 93.75k SCLK	
CRC_Calculate	CRC.asm	CRC.h; CRC.inc;	BYTE: Next Byte into CRC-8 engine	-	~25 µs for one BYTE	External Communication; CRC-Single Word
SelfTest_WDT	Selftest_WDT.c	Selftest_WDT.h	-	BYTE: 0 – No Err; 1 – WDT disabled/Err;	4 * Sleep timer	Others
SelfTest_CSD_CheckShorts	Selftest_CSD.asm	Selftest_CSD.h	-	BYTE: 0 – No Err;	~380 µs for 2 sensors	Others
SelfTest_CSD_CheckBaselines				N – Err Sensor number	~60 µs for 2 sensors	
SelfTest_CSD_CheckMod				WORD: CMOD charge/discharge CPU counts	~1.5 ms	Others
SelfTest_StackOverflowChk_Init	SelfTest_StackOverflow.asm	SelfTest_StackOverflow.inc; SelfTest_StackOverflow.h;	-	-	7 µs	Stack Overflow check
SelfTest_StackOverflowChk				BYTE: 0 – No Err; 1 – Err;	9 µs	

Document History

Document Title: PSoC® 1 – IEC 60730 Class B Safety Software Library – AN81828

Document Number: 001-81828

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	4222277	MSUR	12/17/2013	New specification.
*A	4637314	VAIR	01/27/2015	Updated correct hyperlink to IEC 60730 General Requirements in Introduction. Updated the clock test description in IntroductionClock (Table 1 – 3). Updated the argument description SelfTest_ROM() API in Flash (Invariable Memory, Table 1 – 4.1). Updated the section Modulator Component (Cmod and Rb) Testing. Updated the note under SelfTest_CSD_CheckShorts() API in Test on Sensor Shorts. Added a note indicating VC2 is used as SCLK in Communications SPI.
*B	5711125	AESATP12	04/26/2017	Updated logo and copyright.

Worldwide Sales and Design Support

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

Products

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

PSoC® Solutions

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

Cypress Developer Community

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

Technical Support

cypress.com/support

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

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