AN78175

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

Author: Taras Kuzo, Andrij Bilynskyy, Dmytro Makara
Associated Project: Yes
Associated Part Family: CY8C38xx, CY8C55xx
Software Version: PSoC Creator™ 2.1 SP1
Related Application Notes: AN58771, AN17582, AN61998

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 1
Overview of Annex H 1
Class B Requirements 2
Class B Safety Software Library 3
API Functions for PSoC 3 and PSoC 5LP 4
- CPU Registers Test 4
- Program Counter 4
- Interrupt Handling and Execution 5
- Clock 6
- FLASH (Invariable Memory) 7
- EEPROM (Invariable Memory) 8
- SRAM (Variable Memory) 9
- March Test 10
- March C Test 10
- March C Minus Test 10
- March B Test 10
- Checkerboard Test 10
- Digital I/O 11
- A/D – Converter 11
- D/A – Converter 12
- Comparator 13
- Communications UART 14
- Communications SPI 14

Recommendations and Examples 15
- UDB Configuration Registers Test 15
- Start Up Configuration Registers Test 16
- Watchdog Test 17
- Example of Communications UART Data Transfer Protocol 17
- Data Delivery Controlling 18

Summary 20
References 20
Appendix A 21
- EEPROM CRC Calculates on Hardware Level using Hardware CRC Component 21

Document History 22
Worldwide Sales and Design Support 23
Products 23
PSoC® Solutions 23

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 the home appliance. MCU damage due to overheating, static discharge, very 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 the appliances software into the following categories (see Appendix B: "IEC 60730-1 Table H.11.12.7"):

- Class A Control functions, which are not intended to be relied upon for the safety of the equipment. Examples are humidity controls, lighting controls, timers, and switches.

- Class B Control functions, which are intended to prevent 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 are 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 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:

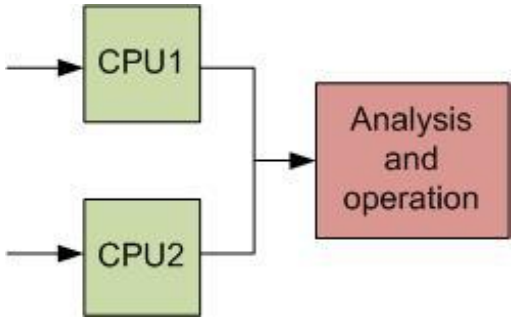
- Single channel with functional test
- Single channel with periodic self test
- Dual channel without comparison

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. Prior to 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 need 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

According to the IEC60730-1 Class B Annex H Table H.11.12.7, there is 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 provide 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 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

Class B IEC 60730 Components Required to be Tested on Electronic Control (see Table H.11.12.7 in Annex H)	Fault/Error
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 requirement is 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 only by adding an appropriated API function and *.c and *.h files from the Class B Safety Software Library. Other self-tests can be applied adding an appropriated 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 stuck bits.
   - A/D- and D/A- convertor. 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 interconnectivity. Often, the user application also needs these self-tests even though they are not provided in Appendix B IEC 60730-1.
   - Watchdog test. Test for chip reset possibility.
   - Comparator test.
   - 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 the 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 listed in this section. 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. See the Checkerboard Test section for more details.

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 256 bytes by looping through the 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. Bit 0 is always ‘0’, so 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 only 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 the Linker commands. The Linker command for placing the functions in the above-mentioned locations is as follows:
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 Project->Build Setting-> Linker-> Command line.
- For PSoC 5LP, this can be done by using the Linker script in **.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 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 triggers a reset because the program execution is out of range.

Function

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

Note 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.

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 executed after the completion of the interrupt service routine.

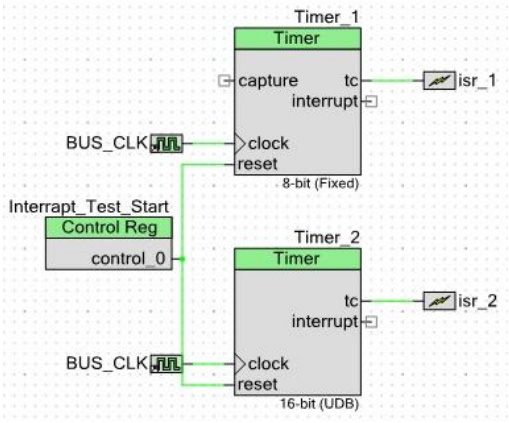
The Interrupt test implements independent time slot monitoring H.2.18.10.4 defined by the IEC 60730 standard. It checks whether the number of interrupts that occurred is within the predefined range.

The goal of the Interrupt test is to verify that interrupts occur regularly. The current test checks the Interrupt Controller by using two interrupt sources. Assume that other 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
          Schematic shown in Figure 3.
          User code from Interrupt Service
          Routine for isr_1 and isr_2 must
          be added.
```

Figure 3. PSoC Creator Schematic for Interrupt Test



Note The component's name should be the same as in Figure 3. 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.

Note For PSoC 5LP, Timer_1 must be moved from the Fixed Function block to a UDB block.

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 20, the test is passed.

Clock

According to the IEC 60730 standard, only harmonics and sub-harmonics of the clock need to be tested. 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).

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
         Schematic shown in Figure 4.
```

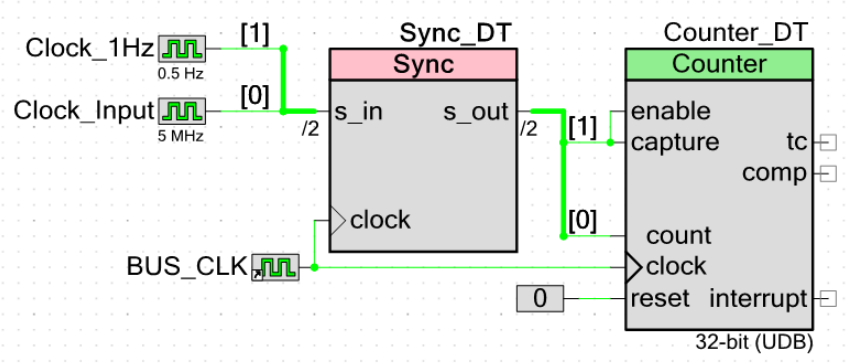
Note The tested clock sources and there accuracies are defined in the SelfTest_Clock.h. If the clock frequency is not defined, the test of the corresponding source will be skipped.

```
//      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 4 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-s 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's call is asynchronous, that is, the function returns the result immediately without a 1-s delay. If the result is not available, the function always returns 0 (success). However, when the result is in the timer capture register (every 1s), the function compares this value and returns the real result. Before returning a result, the function changes the clock source to the next one that must be measured.

Figure 4. Clock Test PSoC Implementation



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 bytes 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 for either error correcting codes (ECC) or additional data storage.

Error Correction Code (ECC) Method

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. The error status is logged into the firmware visible registers.

In PSoC 3 and PSoC 5LP if the error is correctable, the ECC block corrects it and updates the CACHE_INT_LOG3 register. If the error is not correctable, it updates the details of the flash location where an error occurred into the CACHE_INT_LOG4 register.

There are four interrupts related to the cache controller that are issued to the CPU but only two interrupts need to be described. 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 a fill 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 fill 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_Memory.c
            SelfTest_Memory.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

The checksum calculation method can be used as an alternative ECC and the ECC storage area used for data.

To complete a full diagnostic of the flash, a checksum of all used flash needs to be calculated. PSOC_FLASH_SIZE in *SelfTest_Memory.h* file defines the flash size that needs to be monitored. The first time a checksum is calculated, it is saved to EEPROM memory. A semaphore is used to check whether the checksum is already stored. Future calculated checksums are compared to the stored checksum to detect flash errors.

Function

```
uint8 SelfTest_FlashCheckSum_Init()
Returns:  0x00   No error
          0xA5   Flash checksum already
          stored in EEPROM
          0x01-0x10 error status of write
          operation to EEPROM
Located in: SelfTest_Memory.c
            SelfTest_Memory.h
```

This function checks the semaphore stored in EEPROM to detect if a checksum is already stored. If not stored, it calculates the checksum and stores it to EEPROM.

This function needs to be called one time before SelfTest_FlashCheckSum() function to initialize the checksum.

PSOC_FLASH_SIZE in the *SelfTest_Memory.h* file defines the size of flash that needs to be checked.

The flash checksum and semaphore need 4 bytes and are stored at the end of EEPROM memory.

Note If this function is used, the user first needs to clear the EEPROM used or change the semaphore (CHECKSUM_SEMAPHORE) defined in *SelfTest_Memory.h* file. If not, the EEPROM can contain the same value in the semaphore area and the SelfTest_FlashCheckSum() self-test will return error.

Function

```
uint8 SelfTest_FlashCheckSum()
Returns:  0   No error
          1   Error detected
Located in: SelfTest_Memory.c
            SelfTest_Memory.h
```

The function SelfTest_FlashCheckSum () is called to do the FLASH memory corruption test using the checksum method.

Note The SelfTest_FlashCheckSum_Init() function needs to be called one time before SelfTest_FlashCheckSum() function to initialize the checksum.

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.

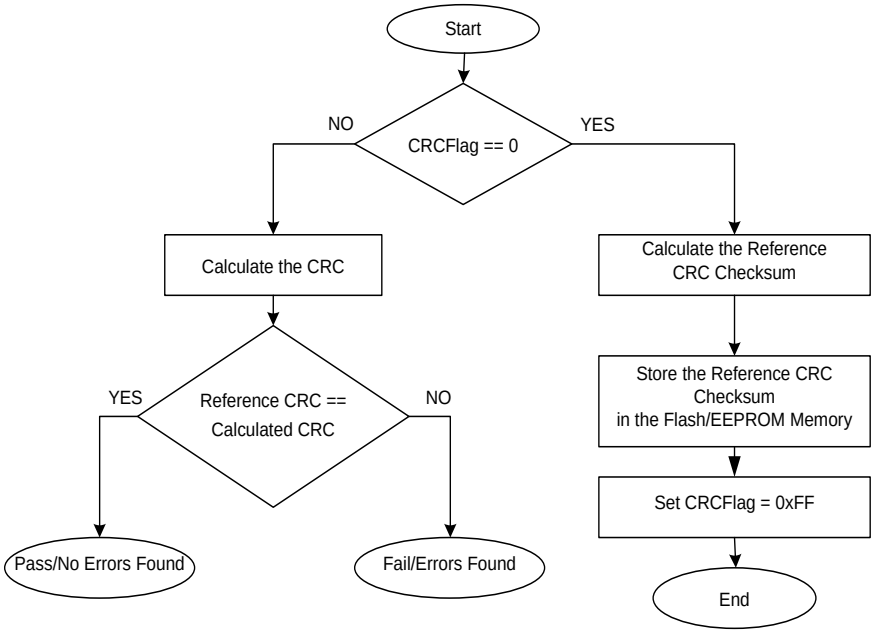
The PSoC 3 and PSoC 5LP EEPROM memories have the following organization:

- 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 the cyclic redundancy check (CRC). CRC-16 standard was used in example for the CRC calculation.

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

Figure 5. Flow Chart for Invariable Memory Test



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

1. At the 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 A.
```

The function SelfTest_EEPROM () is called to do the FLASH memory corruption test using hardware ECC.

Appendix A 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 synchronizations.

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

The example project uses the three last bytes in the EEPROM to store the First Calculation Status Byte and two CRC bytes. CRC_CALC_STATUS, CRC_LO and CRC_HI define this bytes.

These bytes are not take a part in the EEPROM CRC calculation.

ClearEEPROMUserArea() – this function clear 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 set CRC_CALC_STATUS.

These two functions must be called after the PSoC start and before the first test result from SelfTest_EEPROM() function. In this case:

If you need to periodically update EEPROM, follow these steps:

1. Change data in EEPROM
2. Call ClearEEPROMUserArea()function to clear CRC_CALC_STATUS status.
3. Call SelfTest_EEPROM()function once to calculate and set CRC value and set CRC_CALC_STATUS.
4. 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.

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 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 the memory and the runtime 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);

}

Checkerboard Test

The Checkerboard RAM test writes checkerboard patterns to a sequence of adjacent memory locations. This is a non-destructive memory test. This test performs the following major tasks:

1. Saves the contents of the memory locations to be tested in the CPU registers.
2. Writes the binary value (the length is dependent on the architecture) 10101010 to the memory location, ‘N’, and the inverted binary value, 01010101, to the memory location, ‘N+1’, and so on, until the whole memory chunk is filled.
3. Reads the contents of all the memory locations in the current chunk and verifies its contents. If the values match, the function continues; otherwise, it stops and returns an error.
4. Step 2 and 3 are repeated by writing the inverted pattern to the same locations.
5. Once a memory chunk is completed, the test of the next chunk starts until all the requested memory area is tested.

Function

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

The function SelfTest_RAM_All() is called to do the SRAM tests immediately after PSoC start.

The function SelfTest_RAM_All() contains the following functions:

- SelfTest_RAM_CHECKERBOARD
- SelfTests_RAM_March_C_Minus
- SelfTests_RAM_March_C
- SelfTests_RAM_March_B

The SelfTest_RAM_CHECKERBOARD can be called at any time. This function checks the RAM without data corruption.

Only one of SelfTests_RAM_March_C_Minus, SelfTests_RAM_March_C, or SelfTests_RAM_March_B functions can be chosen and called immediately after the PSoC start. These functions check the RAM with data corruption. After passing the test, these functions reset the PSoC. The third byte in the EEPROM is used in the example project to monitor the reset process caused by one of these functions

The user can select one of these functions to use for the RAM test.

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

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

Note This test is application dependent and can be run only if the hardware allows it! Test values may cause pins to be momentarily configured into an incorrect state for the end application.

All the PSoC I/O registers are located in the SRAM memory. If the user uses the SelfTest_RAM_CheckerBoard() function for the RAM test, this function also tests the PSoC I/Os registers for stuck bits.

Function

```
uint8 SelfTest_RAM_CHECKERBOARD()
Returns:  0   No error
          1   Error detected
Located in:   SelfTest_Memory.c
          SelfTest_Memory.h
```

The function `SelfTest_RAM_CheckerBoard()` is called to do the RAM memory and PSoC I/Os registers test on stuck bits.

A/D – Converter

This function implements the ADC value test. 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 1.

Function

```
uint8  SelfTest_ADC(void)
Returns:  0   No error
          1   Error detected
Located in:   SelfTest_Analog.h
              SelfTest_Analog.c
              Schematic shown in Figure 6.
```

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

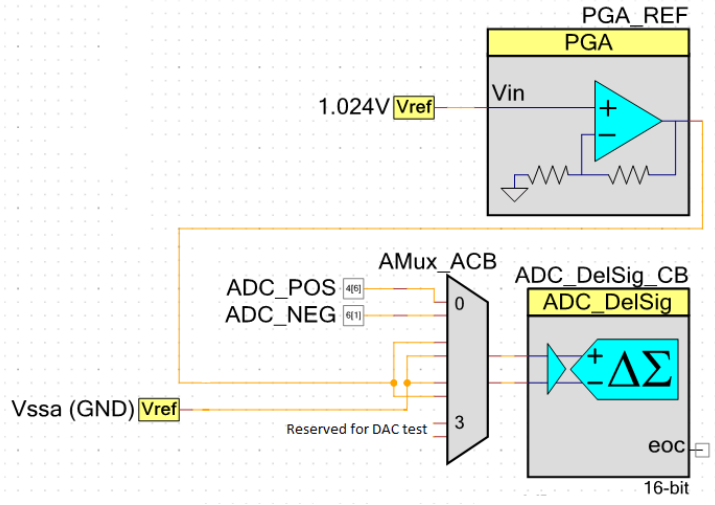
```
#define ADC_TEST_ACC 10
              // +/- ADC result value
```

Figure 6 shows this test PSoC schematic implementation. The ADC input terminals switch between the corresponding pins (or other user modules such as PGA, TIA, opamp, and so on) and 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. The Delta sigma ADC has four configurations and one of these configurations is used for the test, the other three are used for the user implementation.

The test function saves all the component configurations and non-retention registers before testing and restores them afterward. This function measures the input reference voltage value only one time during the call, during the next call the input reference voltage polarity changes and the test is provided with a new polarity. This function implementation decreases the test execution time.

Figure 6. ADC Test PSoC Implementation



D/A – Converter

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

Function

```
uint8  SelfTest_DAC(void)
Returns:  0   No error
          ▪   Error detected
Located in:   SelfTest_Analog.h
              SelfTest_Analog.c
              Schematic shown in Figure 7.
```

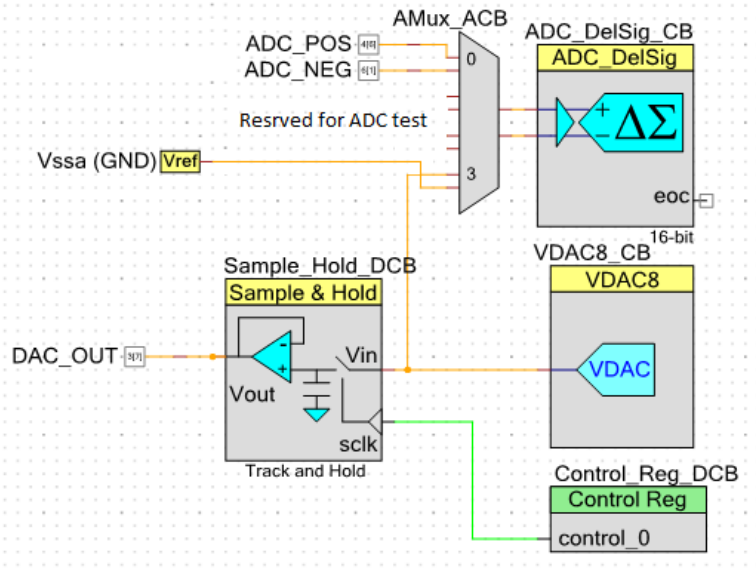
The ADC accuracy is defined in "*SelfTest_Analog.h*". A 10-bit ADC is used for testing.

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

Figure 7 shows this test PSoC schematic implementation. The ADC input terminals switch between the corresponding pins (or other user modules such as PGA, TIA, opamp, and so on) and the DAC output by using an analog multiplexer. The Track and Hold unit is used to maintain the DAC output value during the test time. As mentioned previously, the tested value is put into the DAC and this value is compared with the value read from the ADC component.

The test function saves all the component configurations and non-retention registers before testing and restores them afterwards. During the call, this function compares only one value, during the next call, this value is incremented, and the test is provided with a new value. This function implementation decreases the test execution time.

Figure 7. 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, so the output value of the comparator must change. When the test is a success, the function returns 0; otherwise, it returns 1.

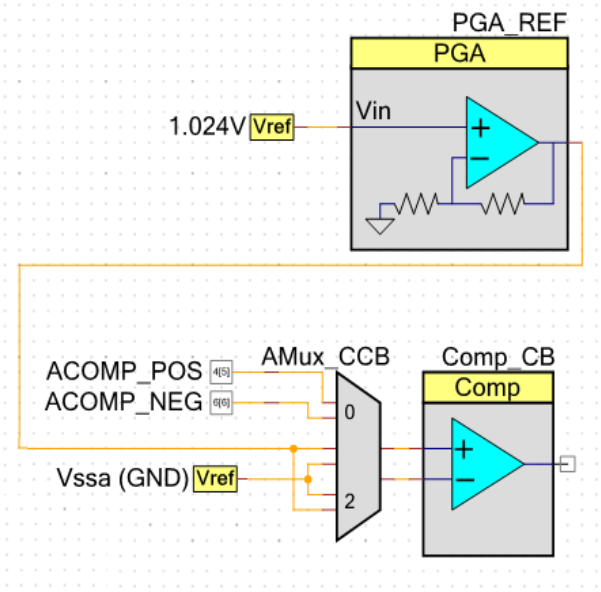
Function

```
uint8  SelfTest_Comparator(void)
Returns:  0   No error
          1   Error detected
Located in:   SelfTest_Analog.h
              SelfTest_Analog.c
              Schematic shown in Figure 8.
```

Figure 8 shows this test PSoC schematic implementation. The comparator input terminals switch between the corresponding pins (or other user modules such as PGA, TIA, opamp, and so on) and the reference voltages using an analog multiplexer. The output values of the comparator are analyzed on different polarities of the input signal. 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 restore them afterwards.

Figure 8. DAC 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. When the test is a success, the function returns 0; otherwise, it returns 1.

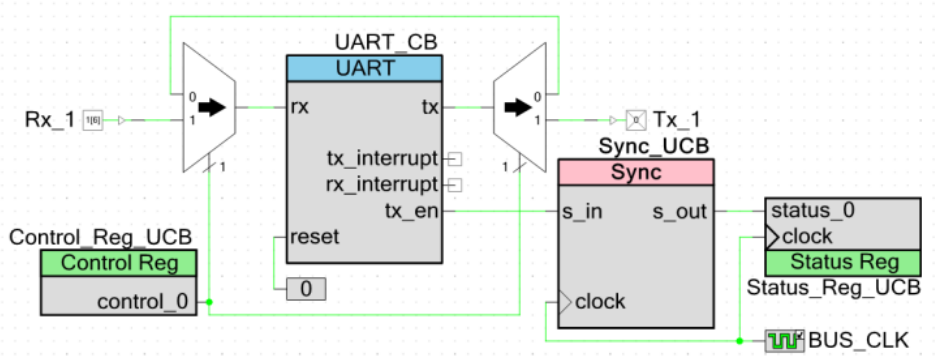
Function

```
uint8  SelfTest_UART_Data(void)
Returns:  0   No error
          1   Error detected
Located in:   SelfTest_UART.h
              SelfTest_UART.c
              Schematic shown in Figure 9.
```

Figure 9 shows this test 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 de-multiplexer. If there are any bytes in the receive buffer, the test is not executed and returns the 0 success. If there are any bytes in the transmit buffer, the function waits a determined period of time for the end of this transmitting and then the test is executed.

The test function saves all the component configurations and non retention registers before testing and restores them afterwards. During the call, this function transmits only 1 byte, during the next call the byte value is incremented and the test is provided with a new value. This function implementation decreases the test execution time.

Figure 9. 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. When the test is a success, the function returns 0; otherwise, it returns 1.

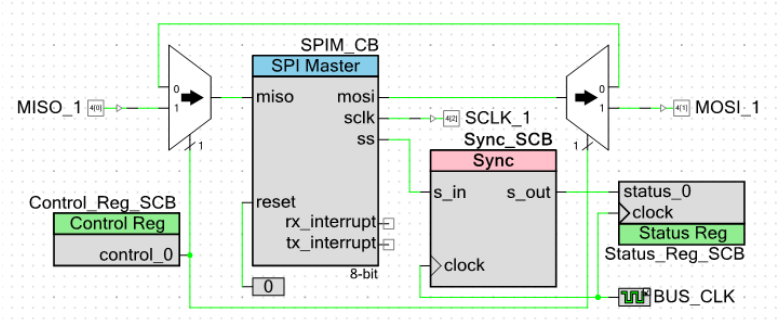
Function

```
uint8  SelfTest_SPI_Data(void)
Returns:  0   No error
          1   Error detected
Located in:   SelfTest_SPI.h
              SelfTest_SPI.c
              Schematic shown in Figure 10
```

Figure 10 shows this test 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 de-multiplexer. If there are any bytes in the receive buffer, the test is not executed and returns the 0 success. If there are any bytes in the transmit buffer, the function waits determined period of time for the end of this transmitting and then the test is executed.

The test function saves all the component configurations and non-retention registers before testing and restores them afterwards. During the call, this function transmits only 1 byte, during the next call, the byte value will be incremented and the test is provided with a new value. This function implementation decreases the test execution time.

Figure 10. 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

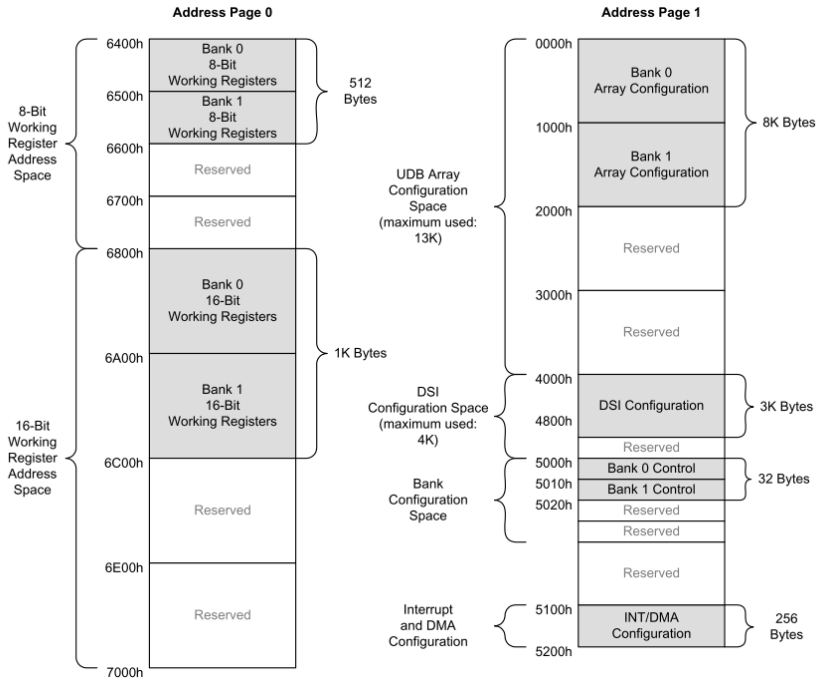
There are two main groups of UDB registers:

- 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 11.

UBD configuration registers are static and configured during design build. These registers should not be changed during device operational and can be checked against the initial configuration.

Figure 11. UDB Array Base Addresses



The following functions allow the user to implement UDB configuration register test in design.

Duplicates of UDB configuration registers are stored in Flash memory after device startup. Periodically, the configuration registers are compared with the stored duplicates. In addition, ECC can be enabled to monitor duplicate errors. Corrupted registers can be restored from flash after checking.

Function

```
cystatus SelfTests_Save_UBD_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 UDB configuration registers to flash memory. UDB configuration registers are located in memory address from 0x00010000u to 0x00012000u and occupy 8K Bytes. The 8K Bytes are 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 defines the location for UDB configuration registers in Flash memory.

Side Effects: Call this function once after PSoC initialization before entering the main program. It writes the correct UBD configuration register values to flash.

Function

```
uint8 SelfTests_UDB_ConfigReg (void)
Returns:  0   No error
          1   Error detected

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

Description: This function checks the UDB configuration registers with the saved values in flash by the SelfTests_Save_UBD_Cfg() function. UDB_REG_FIRST_BLOCK and DB_REG_LAST_BLOCK defines the location for UDB configuration registers in Flash memory.

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 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()

Duplicates of startup configuration registers are stored in Flash memory after device startup. Periodically the configuration registers are compared with stored duplicates. ECC can also be enabled to monitor duplicate errors. Corrupted registers can be restored from the flash after checking.

Note The following functions are examples and can only be applied to the example project. If the user makes changes in schematics or configuration, other configuration registers may be generated in *cyfitter_cfg.c* file. The user must 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 *SelfTest_ConfigRegisters.c* file. It excludes checking of CYREG_CLKDIST_DCFG1_CFG0 register generated in ClockSetup() function. This register is used for the Clock self test and may be changed 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 start up 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 all listed start up configuration registers. Functions compare values stored

in flash with the current configuration registers values. Returns fail if values are different.

Watchdog Test

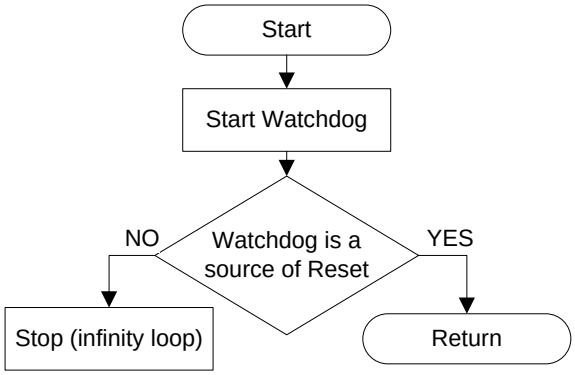
This function implements the watchdog functional test. The function starts the watchdog timer and runs an infinite loop. If the watchdog timer works, it generates a reset. After the reset, the function analyzes the reset source. If the watchdog is the source of the reset, the function returns; otherwise,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

Function

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

Figure 12 shows the test flow chart.

Figure 12. WDT Test PSoC Implementation



Example of Communications UART Data Transfer Protocol

Data is placed into the packets with a defined structure to be transferred. All packets have a CRC calculated with the packet data to ensure the packet's safe transfer.

Figure 13 shows the packet format.

Figure 13. 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 in 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. It is calculated for all packet bytes from ADDR to last data byte. CRC is calculated after 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. It is calculated for all packet's bytes from ADDR to last data byte including. CRC is calculated after 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 14 shows the Communication process time diagram. The communication procedure can be divided in 3 parts:

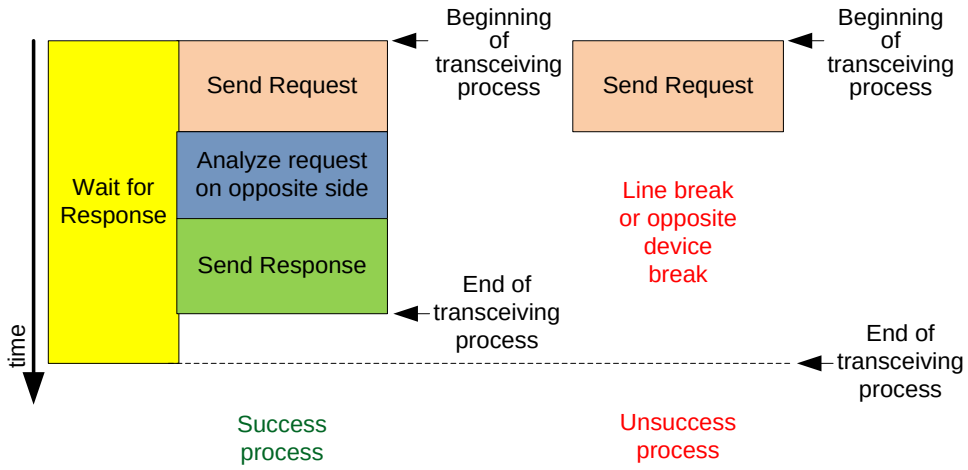
- Send Request (opposite side Receive Request)
- Wait for Respond (opposite side Analyze Request)
- Receive Respond (opposite side Send Respond)

'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 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 respond is not delivered.

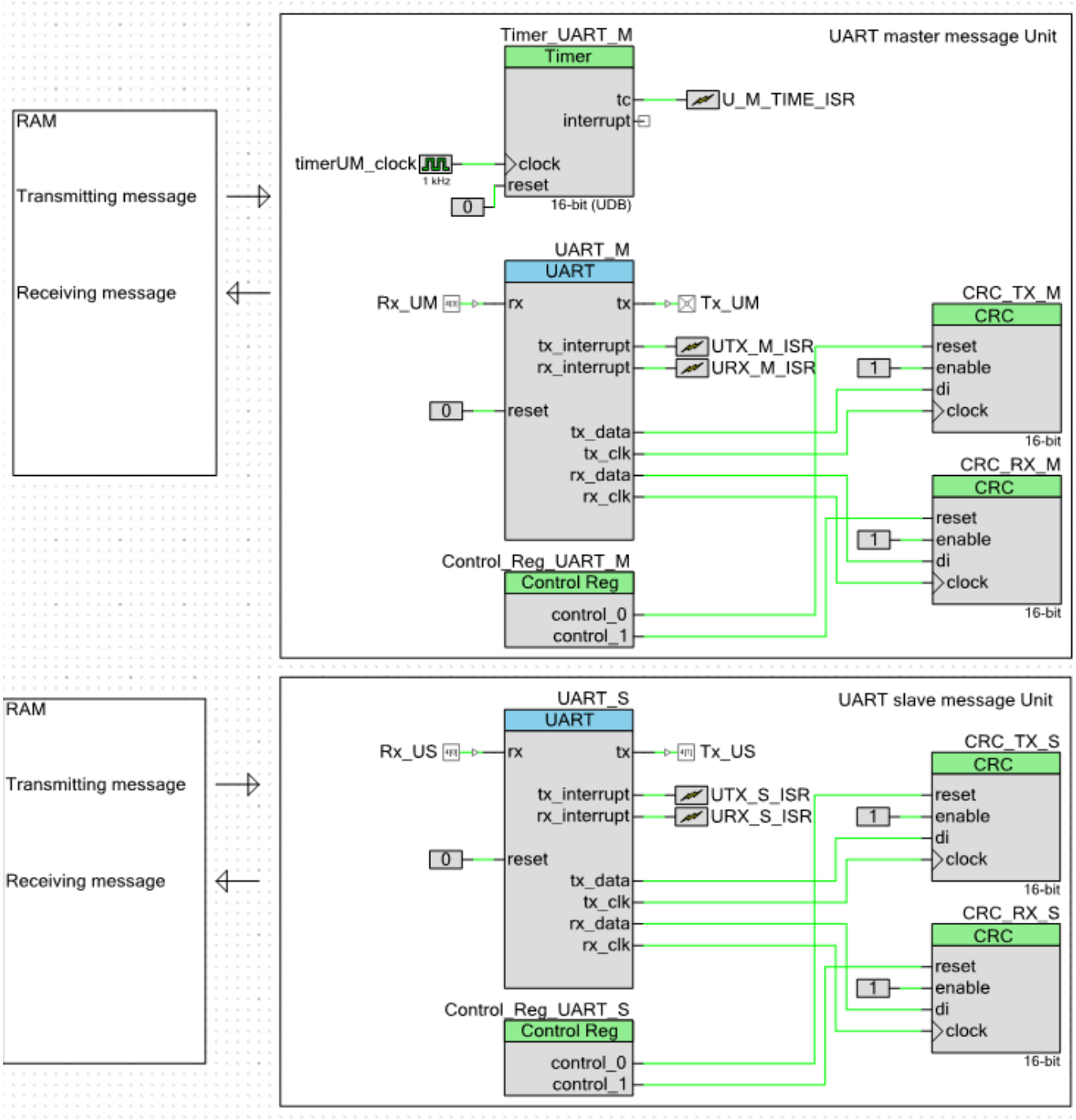
Figure 14. Timing Diagram of Communication Process



PSoC Implementation

Figure 15 represents the protocol PSoC 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 into 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 15. 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:

Initializing the UART message unit

Void UartMesMaster_Init(void)

Start the process of transmitting and receiving message

uint8UartMesMaster_DataProc(uint8 address, char * txd, uint8tlen, char * rxd, uint8rlen)

This function returns the result of the process start: 0 = success is 0 and 1 = error. An error can be that the unit is already busy sending a message or null transmitting length was detected.
The input parameters:
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

Return current state of the UART message unit

uint8UartMesMaster_State(void)

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, 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.

Obtain 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.

uint8UartMesMaster_GetDataSize(void)

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

Initializing the UART message unit

Void UartMesSlave_Init(uint8 address)

The input parameters:
address – the slave address.

Start respond

uint8 UartMesSlave_Respond(char * txd, uint8tlen)

This function returns the result of process start: success is 0 and 1 is error (the unit has not received a marker).
The input parameters:
txd – the pointer to the transmitted data (Request data)
tlen – the length of the request in bytes

Return current state of the UART message unit

uint8 UartMesSlave_State(void)

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.

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

uint8 UartMes_GetDataSize(void)

Obtain pointer to the received data

uint8 *UartMesSlave_GetDataPtr(void)

Summary

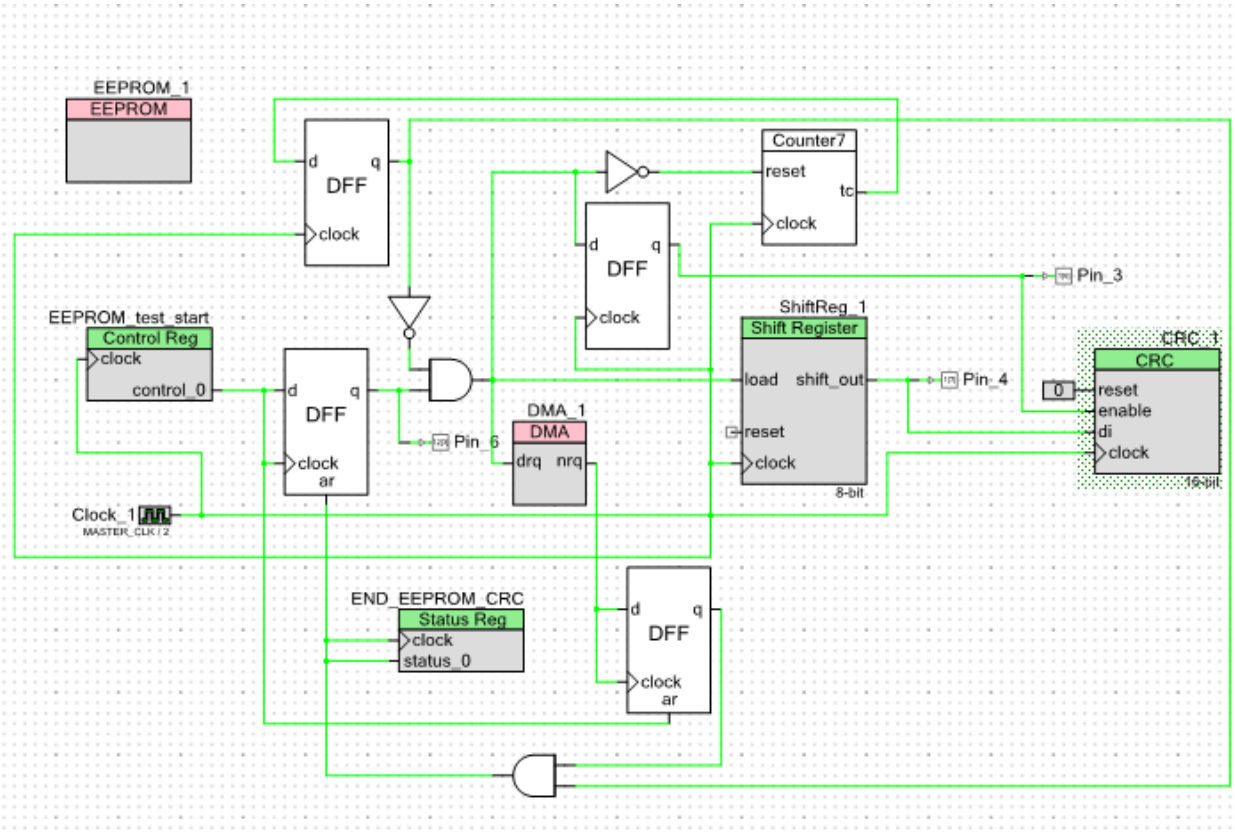
This application note describes how to implement various diagnostic tests defined by the IEC 60730 standard. The introduction of IEC 60730-1 into the design of white goods and other appliances will add a new level of safety for consumers. By taking advantage of PSoC 3 and PSoC 5LP, design teams can comply with regulations while maintaining or reducing electronic systems cost. The use of PSoC and the Class B Safety Software Library allow creating a strong system-level development platform and 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

EEPROM CRC Calculates on Hardware Level using Hardware CRC Component



The EEPROM CRC calculates at the hardware level using a hardware CRC component. 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 Status Register) were used for synchronization.

Document History

Document Title: 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/QUS	05/30/2012	Updated Recommendations and Examples (Removed CapSense CSD (Removed Test on Sensor Shorts, Test on Sensor Disconnect, and Modulator Component (C_{mod} and R_b) 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.

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

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5LP

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

PSoC is a registered trademark of Cypress Semiconductor Corp. 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

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

© Cypress Semiconductor Corporation, 2012. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.
This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.
Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.
Use may be limited by and subject to the applicable Cypress software license agreement.