AN13124

Fast Implementation of NXP Safety Library IEC60730/Class B

Rev. 0 — 03/2021 Application Note

by: NXP Semiconductors

Contents

1 Introduction

The goal of this document is to help with a very quick implementation of the basic Safety Class B tests and to help to understand their principles and purposes. The tests explained in this document are the CPU registers test, the CPU program counter test, the invariable memory test, and the variable memory test. Note that the described implementation is appreciably simplified and it does not fulfil the overall requirements. These requirements can be met after a consecutive study of the released NXP documentation about safety libraries and safety examples. In general, the additional requirements and implementation steps are the modification of the linker configuration file and the management of calling restrictions in a runtime mode of the application. When you go through the implementation steps described in this manual and think that our library would be suitable for your project and the device you use is not covered in any release of the NXP Class B libraries, contact your FAE or visit the NXP community on the web.

2 Getting the library that supports your core

- At the www.nxp.com/iec60730 webpage, scroll down to the product catalog.
- Find the row dedicated for the core of your device and download the library package from the “Library and documents” column.
- In the downloaded package, find the A or LIB files compiled for the IDE that you use (IAR, Keil, MCUXpresso) and dedicated to the “core” tests only (without “COM” in the name).
- Find the header files that are placed in a stand-alone folder.

3 Importing the library into your project

- Add the following items to your project:
  - Header files from the following sub-folders:
    - *flash*
    - *programCounter*
    - *ram*
    - *register*
    - *stack*
  - General header files:
    - *iec60730b.h*
    - *iec60730b_types.h*
    - *iec60730b_core.h*
    - *asm_mac_common.h*
  - The respective pre-compiled library (for example, *IEC60730B_M0_IAR_v4_1.a*)

  - The S file that appears in the *programCounter* sub-folder.
- The example is shown in Figure 1.
- Add the paths to the files in the compiler settings.
- Delete following includes in the *iec60730b.h* file:
  - **_aio.h*
  - **_clock.h*
  - **_dio.h*
  - **_dio_ext.h*
  - **_tsi.h*
  - **_wdog.h*
- Add the following includes to the source file where the library is used:
  - *iec60730b.h*
  - *iec60730b_core.h*

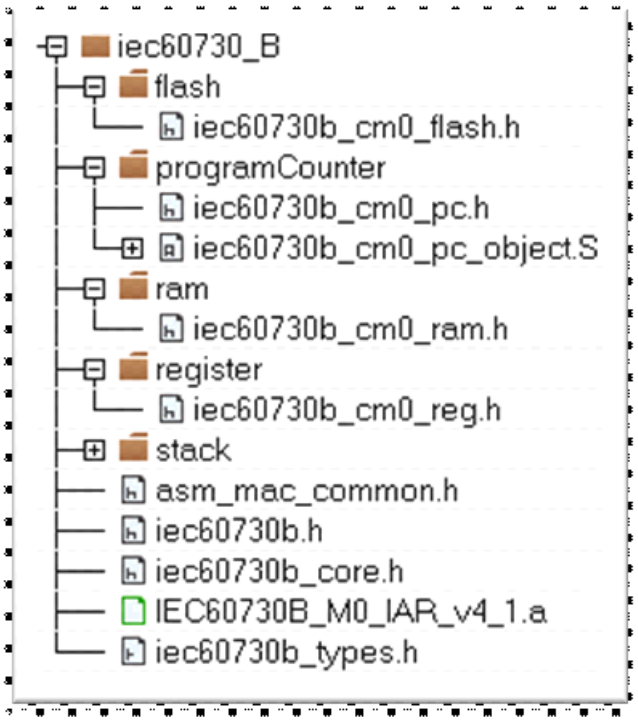

Figure 1. Files imported to the project

4 Final implementation

Now the library is ready for use. Proceed with the implementation.

4.1 CPU registers test

- These functions have no input parameters.

- For the first try, call the functions only when all interrupts are disabled.
- Define the variables where the results of the respective tests are to be stored.

```
uint32_t Result_CPU_Register;
uint32_t Result_CPU_NonStackedRegister;
uint32_t Result_CPU_Primask;
uint32_t Result_CPU_SPmain;
uint32_t Result_CPU_SPprocess;
uint32_t Result_CPU_Control;
```

- Call the functions, as shown below.

```
Result_CPU_Register = FS_CM0_CPU_Register();
…
```

- Check the results. If no CPU register is corrupt, the result should always be zero.
- See the library user's guide for the calling restrictions.

4.2 CPU program counter test

- Call the test when all interrupts are disabled.
- Define one variable to store the result to. Another zero-initialized variable is used as a flag. One variable is initialized with the value that represents a valid address in the RAM, which is aligned to 32 bits.

```
uint32_t Result_PC;
uint32_t pc_test_flag = 0;
uint32_t pc_test_address_in_ram = 0x4004000;
```

- Call the program counter test function with the following three input parameters:
  - A valid address in the RAM (in our case, the *pc_test_address_in_ram* variable)
  - The name of the auxiliary function from the **_pc_object.S* file (*FS_PC_Object*)
  - The pointer to the address of the "flag" variable: *(uint32_t*)&pc_test_flag*.
- It should look as follows:

```
Result_PC = FS_CM0_PC_Test(pc_test_address_in_ram, FS_PC_Object,(uint32_t*)&pc_test_flag);
```

- See the library user's guide for proper implementation.

4.3 Invariable memory test

- This implementation does not have any evaluation. It has just a simple CRC calculation without checking if the result is as expected.
- Define one variable to store the result to and define the start address (in the invariable memory) and size of the tested area. Do not choose a large part of the tested area. We use zero as the start condition seed parameter.

```
uint32_t Flash_CRC;
uint32_t flash_test_address = 0xF0;
uint32_t flash_test_size = 0x20;
uint32_t start_condition_seed = 0x0;
```

- To avoid problems with the CRC hardware module (which is not available on each device), use the software calculation.

```
Flash_CRC = FS_CM0_FLASH_SW16(flash_test_address, flash_test_size, 0x0,
start_condition_seed);
```

- To evaluate that the data in the tested area have not changed, call this function again after a while.
- See the respective user's guide to find out how to use the test in the runtime and how to calculate the CRC in the linking phase of an application (before writing to the memory).

4.4 Variable memory test

- Define one variable to store the test result to.

```
uint32_t Result_RAM;
```

- Define the backup area, the area to test, the start address, the end address, and the size of the test iterations.

```
uint32_t ram_test_backup[2]; // array of two 32-bit variables will be enough for backup
uint32_t ram_test_area[6] = {0x10101010, 0x20202020, 0x30303030, 0x40404040, \
 0x50505050, 0x60606060}; // an area to be tested
uint32_t ram_test_start_address;
uint32_t ram_test_end_address;
uint32_t ram_test_itteration_size = 8;
```

- The start address of the test is the address of the *ram_test_area* array. The end address is given by the start address and the size of the array.

```
ram_test_start_address = (uint32_t)&ram_test_area;
ram_test_end_address = ram_test_start_address + sizeof(ram_test_area);
```

- The last parameter of the RAM test is the name of the function that executes either the March C or March X algorithms.

```
Result_RAM = FS_CM0_RAM_AfterReset(ram_test_start_address, \
 ram_test_end_address, \
 ram_test_size, \
 (uint32_t)&ram_test_backup, \
 FS_CM0_RAM_SegmentMarchX);
```

5 Debugging of core tests

The core tests that are most often used in safety class B applications are already implemented. This chapter explains how to easily verify that the tests are working as expected and how to understand them better.

5.1 CPU registers test

In general, these tests consist of writing, reading, and comparing the 0x55 and 0xAA patterns. When stepping into the CPU registers test function with the debugger, patterns appear in the CPU registers under the test. After a successful execution, the function returns a zero value.

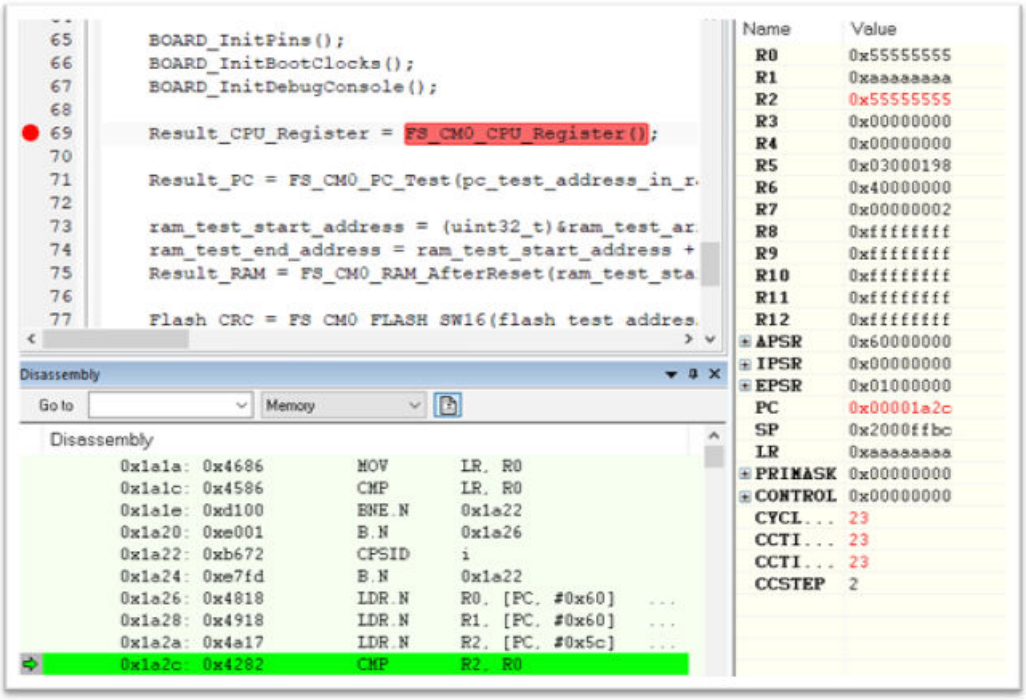

Figure 2. Debugging of the CPU registers test

5.2 CPU program counter test

In this example, we will focus on all three input parameters. Note the value of the first input parameter. It represents an address in the RAM memory. In this case, it is 0x4004000. The second parameter is a reference to the *FS_PC_Object* function. Find out the address where this object is linked in your application. It is in the MAP file that is created after a build. In this example, the address is 0x27f0. Finally, put the variable that represents the third parameter into a debug watch window to see its value.

Jump into the function and debug it step by step while watching the value of the flag variable and the value of the program counter register. During execution, the value of the flag variable should change from zero to one. Before the end of the function, it should be changed back to a zero value. In the program counter register, you shall see 0x4004000 and 0x27f0, along with other values. It means that the program counter went through the required addresses and we can consider it as tested. It is obvious that these two input parameters should be chosen with the aim to fill the program counter with the best possible patterns. An ideal case would be to use 0x55555555 and 0xAAAAAAAA. Due to memory mapping, it is not possible. The test function can be called multiple times with different first parameters. The data that occupies the address defined by the first parameter is backed up during the test.

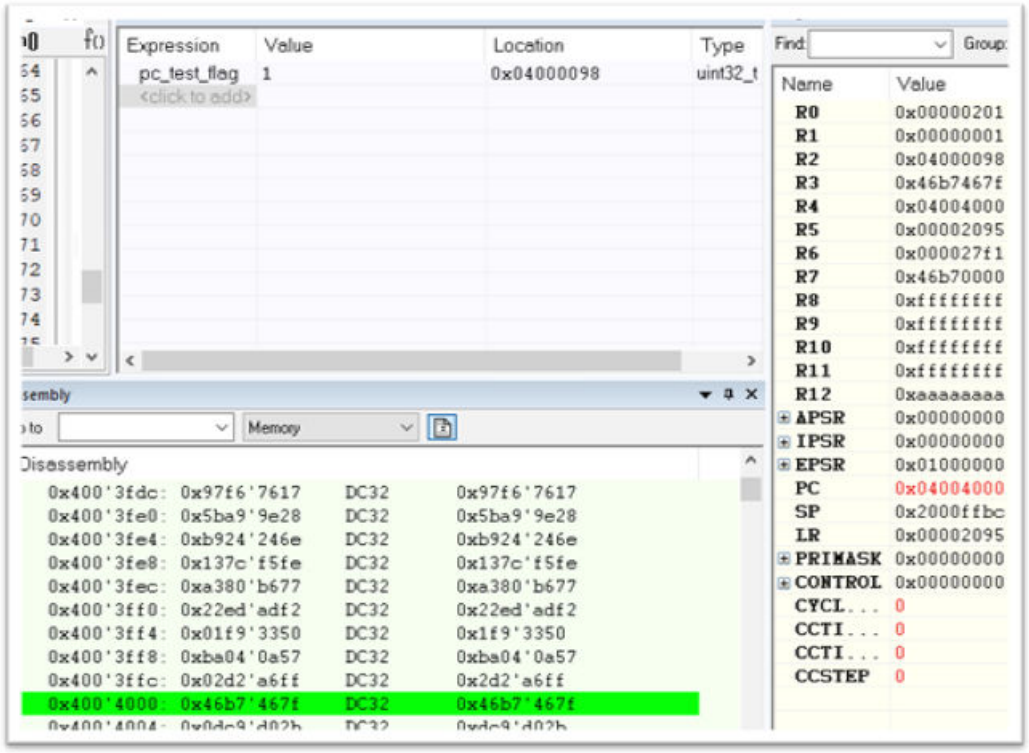

Figure 3. Debugging of the program counter test

5.3 Invariable memory test

The principle of this test is quite simple. The function takes the data from the defined memory range and applies a standard CRC calculation algorithm to it. The key is to compare this result with another relevant result. In a standard implementation, a mechanism in the post-build (linker) phase is used. The IAR IDE has its own integrated module that works very well. Other tools require the use of an external tool that can be embedded into the application (for example, SRecord). When comparing the results from the post-build calculation and the results from the library function, you can detect potential errors that happen when loading the application to the device. On the other hand, comparing the results from an after-reset test with the results from a runtime check can detect errors that happened during the life cycle of an application. The library flash test function simply takes the data from the defined addresses. Therefore, it can be also applied to the RAM memory, knowing that the unchanged result comes only with unchanged data.

5.4 Variable memory test

Probably the most straightforward way to debug this test is to put the backup array and the tested array to the watch window while going step by step through the disassembly code. While doing so, the function checks the backup area by writing and reading 0x55 and 0xAA in a predefined sequence. Then, the data from the tested area is moved to the backup, its location is tested, and the data is moved back. Figure 4 shows a situation where the memory is tested on addresses from 0x0400000C to 0x04000010, while the original data is stored in the *ram_test_backup* array.

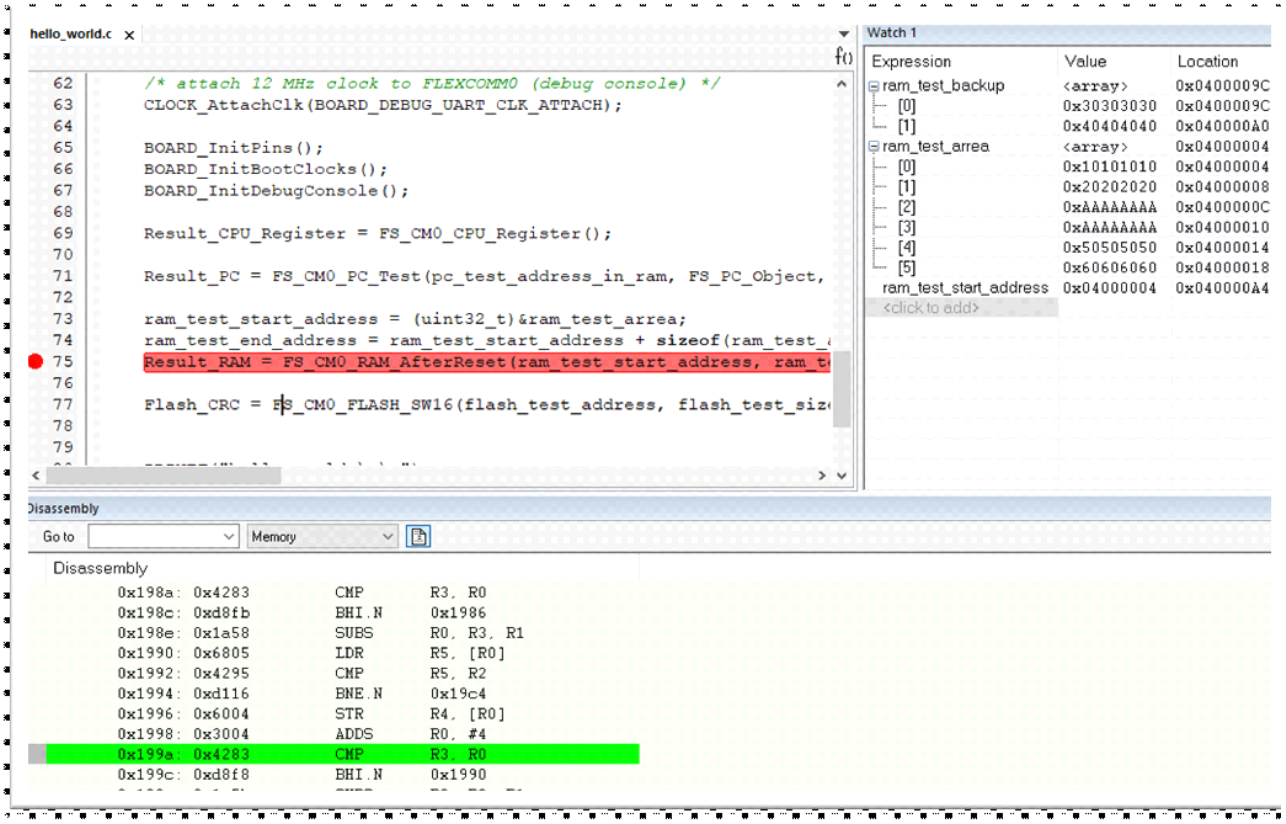

Figure 4. Debugging of the variable memory test

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 03/2021

Document identifier: AN13124

arm