NXP Semiconductors
Application Note

Document Number: AN5321
Rev. 0, 08/2016

Safety Class B with PMSM Sensorless Drive

Contents

1. Introduction 1
1.1. Audience 1
1.2. Scope of this application note 2
1.3. Safety application software 2
1.4. Abbreviations 4
2. Linking Phase 4
2.1. Memory partitioning and placement of code 4
2.2. Checksum 6
2.3. Stack and blocks reserved for safety tests 8
3. Initialization Phase 10
3.1. Watchdog 10
3.2. After-reset tests 12
4. Runtime Phase 12
5. Application Behavior in Case of Safety Error 13
6. Components 13
6.1. CPU registers 14
6.2. Program Counter (PC) 14
6.3. Interrupt handling and execution, program flow 17
6.4. Clock 19
6.5. Invariable memory 20
6.6. Variable memory 22
6.7. Stack 24
7. Conclusion 25
8. References 25
9. Revision History 25

1. Introduction

Safety is an obligatory part of many applications. It is controlled by international standards related to electrical devices. The affected devices are those that, in case of failure, can cause injury, death, pollution, or other dangerous situations. It is a difficult task to fulfill the safety requirements. This application note describes the software part of the application safety. It provides a complex overview of the appliance safety software and you can use it as a guide for creating such software.

1.1. Audience

If you plan to develop software for the Class B-related applications, this application note is a good starting point. If you are new to the field of safety applications, going through this application note can save you a lot of time and effort. The applications that are a part of the Safety Class B are indirectly specified by the IEC/UL 60335. The most common applications are: washing machines, dishwashers, dryers, induction cooking, circulating pumps, compressors, refrigerators, and other. This application note is also targeted for all engineers and designers who work on similar applications.

1.2. Scope of this application note

This application note focuses mainly on safety application software, which is only a part of the complex requirements for safety applications. To fulfill software requirements, tests of the respective components stated in the international standards (IEC/ UL 60730 and IEC/UL 60335) must be implemented. To get detailed information about most of the standards, look for the NXP Safety Class B library. The information provided in this document is essential to implement the components. This application note is provided together with the example application, where the safety library and safety software are integrated into a real motor control application. The parts of code from the example application are discussed in this document for a better explanation. The majority of the required Class B tests is implemented and described.

This is the list of the implemented tests:

- CPU registers—tested for the “stuck at” condition.
- Program counter—tested for the “stuck at” condition.
- Flash memory—tested for the single bit error and multiple bits error.
- RAM—tested for the “stuck at” condition and direct coupling faults.
- Clock source—tested for the “wrong frequency” condition.
- Watchdog—tested for wrong frequency and proper functionality.
- Application stack—the same test as for the RAM and the overflow and underflow conditions.
- Interrupt handling and execution—the interrupts are tested for too frequent or too infrequent executions. The program flow test is also integrated within this component.

NOTE

The Digital I/O and Analog I/O components’ tests are not covered by this application note.

The “PMSM sensorless on KV31” project is used as the base application. For more information about the “PMSM sensorless on KV31” application and the IEC60730 library, see Section 8, “References”. The difference in the hardware configuration is described in the respective release notes.

1.3. Safety application software

Most of the time, certain software tests must be performed periodically on the devices under control that are specified as Class B or Class C according to the IEC/UL 60730. The list of the software/hardware components that must be tested is found in the respective standards (for example IEC60730 and IEC60335). The tests required for a specific application are those that can affect its safe behavior. In most applications, the components must be tested during the runtime. This results from the rule that the failure must be detected (complete software tests must be performed) at least once during half of the “Mean Time Between Failure (MTBF)”. The exception can take place when the application is always switched off before half of the MTBF expires. In that case, the tests can be performed in the initialization phase. The tests that are described in this document are listed in the previous section. Be aware that the implementation of the safety tests requires the complete application to be adapted to it. All the details are stated in the following sections. The high-level diagram of such application is shown in the following figure.

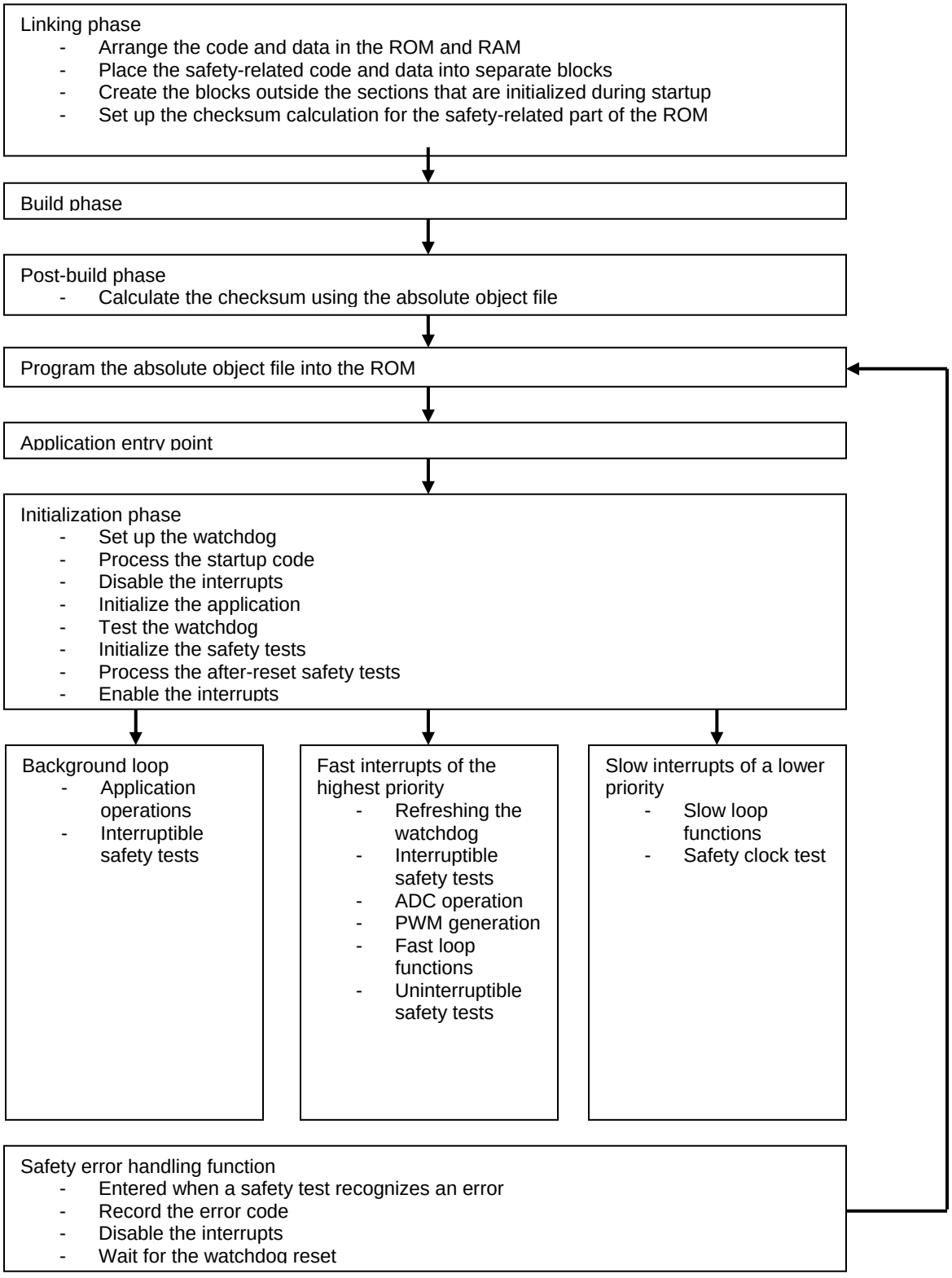

Figure 1. High-level diagram of application with safety integrated

1.4. Abbreviations

- POR—Power-On Reset
- RAM—Random Access Memory
- ROM—Read-Only Memory
- MTBF—Mean Time Between Failures
- CPU—Central Processing Unit
- PC—Program Counter
- CRC—Cyclic Redundancy Check
- LPTMR—Low-Power Timer

2. Linking Phase

A safety application can't be built without a customized linker setup. Most of the settings can/must be done in the linker configuration file (the extension for the IAR IDE is *.icf*). Some settings are done in the compiler options block.

In this phase, configure the memory partitioning. It is a good practice to separate the safety-related data and code from the safety-unrelated data and code. Some of the memory blocks must be allocated to the test routines. The checksum of the defined data must be calculated in the linker phase to be compared later in the application. This section provides valuable information and examples of adjusting the linker for safety applications. For more details about the linker directives and options, see the respective IAR IDE documentation.

2.1. Memory partitioning and placement of code

The first thing to start with is the memory partitioning and placement of code. If the data and code are well-arranged and structured from the beginning, it saves a lot of time and effort in the final phase. There are different approaches needed for the ROM and the RAM.

2.1.1. Code and data placement in ROM

On Kinetis devices, the ROM is always represented by the flash memory. In general, the flash memory is filled with the execution code and constants. The important thing is to have the safety-related code and data separated from other code and data. If these conditions are met, checking/testing of the flash memory during runtime is easier and faster. This is because only the safety-related part must be checked. The selected lines from the linker configuration file are shown below. Assume there is 512 KB of flash memory that begins at address 0x0. The safety-related code is placed in the area between 0x410 and 0x5fff. The checksum is placed in the area between 0x3fff0 and 0x3ffff.

1. The symbols that define the addresses important for the linker configuration are:

```
define symbol m_interrupts_start       = 0x00000000;
define symbol m_interrupts_end         = 0x000003FF;
define symbol m_text_start             = 0x00000410;
define symbol m_text_end               = 0x0007FFFF;
define symbol safety_region_end__      = 0x5fff;
```

```
define symbol __FlashCRC_start__       = 0x3fff0;
define symbol __FlashCRC_end__         = 0x3ffff;
```

2. The regions are defined by these symbols:

```
define region TEXT_region   = mem:[from m_interrupts_start to m_interrupts_end]
                            | mem:[from m_text_start to m_text_end];
define region SAFETY_region = mem:[from m_text_start to __safety_region_end__];
define region CRC_region    = mem:[from __FlashCRC_start__    to __FlashCRC_end__] ;
```

3. Specify the blocks in the flash memory. The first block contains a section with the code that is to be copied into the RAM in the application startup. The CHECKSUM block is filled with the newly-defined *.checksum* section. Put the start mark and the end mark and all the objects that are safety-related into the SAFETY FLASH BLOCK. This code is going to be tested within the flash test. The marks are referenced in Section 2.2, “Checksum” and Section 6.5, “Invariable memory”. The list of the linked objects is in the **.map* file.

```
define block CHECKSUM { section .checksum };
define block SAFETY_FLASH_BLOCK with fixed order
{
readonly section checksum_start_mark,
section .iar.init_table,
section .text object IEC60730_Kinetis_CM4_CM7_Class_B_v1_1.a,
section .text object IEC60730_Kinetis_CM4_CM7_Class_B_FPU_Extension_v1_0.a,
section .text object IEC60730_B_CM4_CM7_wdg.o,
section .text object IEC60730_Safety.o,
section .text object main.o,
section .rodata object IEC60730_Safety.o,
readonly section checksum_end_mark
};
```

4. Place the blocks into the defined regions:

```
place in TEXT_region { readonly };
place in CRC_region { block CHECKSUM };
place in SAFETY_region { block SAFETY_FLASH_BLOCK};
```

2.1.2. Code and data placement in RAM

The placement of code and data in the RAM has the same or at least similar reason as the code and data placement in the flash memory. The benefit is the necessity to check only a limited area of the RAM. Consider this approach for the time-consuming RAM test. Placing the data into the RAM can be performed in several ways. In this example, an extra section that is manually initialized during the application startup is created. The lines from the linker configuration file are shown below.

1. The following symbols define the addresses important for the linker configuration. Define them directly by the addresses or symbols defined before:

```
define symbol __size_cstack__           = 0x0400;
define symbol __ram_vector_table_size__ = 0x00000400;
define symbol m_interrupts_ram_start    = 0x1FFF8000;
define symbol m_data_start              = m_interrupts_ram_start + __ram_vector_table_size__;
define symbol m_data_end                = 0x1FFFFFF7;
define symbol m_data_2_start            = 0x20000018;
define symbol m_data_2_end              = m_stack_test_p_1 - 0x1; /* 0x2000FD9F */
```

2. The regions are defined by these symbols:

```
define region DATA_region = mem:[from m_data_start to m_data_end]
                          | mem:[from m_data_2_start to m_data_2_end-__size_cstack__];
```

3. Create the *.safety_ram* section that is going to reside in the SAFETY_RAM_BLOCK and place this block in the *DATA_region*. In this example, the block is not placed at a fixed address.

```
define block SAFETY_RAM_BLOCK {section .safety_ram};
place in DATA_region { block SAFETY_RAM_BLOCK};
```

In this case, the initialization code writes zeros to all variables. These lines are added into the application startup code:

```
#pragma section = ".safety_ram"

uint32 n;
uint8 *safety_ram = __section_begin(".safety_ram");
uint8 *safety_ram_end = __section_end(".safety_ram");

n = safety_ram_end - safety_ram;
 while (n--)
   *safety_ram++ = 0;
```

See Section 6.6, “Variable memory” as the reference to work with this section and the related data in the application.

2.2. Checksum

One of the basic safety procedures is checking the content of the read-only memory (flash). This is preferably made by calculating the checksum by a dedicated algorithm. In the library, the CRC-16 (0x1021) pattern is used. For more information about the checksum and ROM checking, see Section 6.5, “Invariable memory”.

In the linking phase of the application, the linker and the compiler must ensure the possibility of calculating the checksum of a given data (from the *.elf* file) before it is physically written into the device memory. If the compiler and the linker do not support this feature, it can be overridden by an external software. However, this is usually a quite complicated solution.

The things to do are:

- Specify the memory range.
- Specify the checksum algorithm.
- Specify the address to place the checksum to. The checksum must be placed in a different area than the area it is calculated for.

The two approaches of setting the parameters needed for the checksum calculation are mentioned here. The first one is easier to apply, but the addresses and the range are fixed. The second approach uses the start and end marks to adjust the parameters for the checksum automatically.

Let’s look at the easier approach first:

1. Specify the memory range and the algorithm parameters in the option block by navigating to *Options>Linker>Checksum*, as shown in the following figure. Set the same start and end addresses in the application.

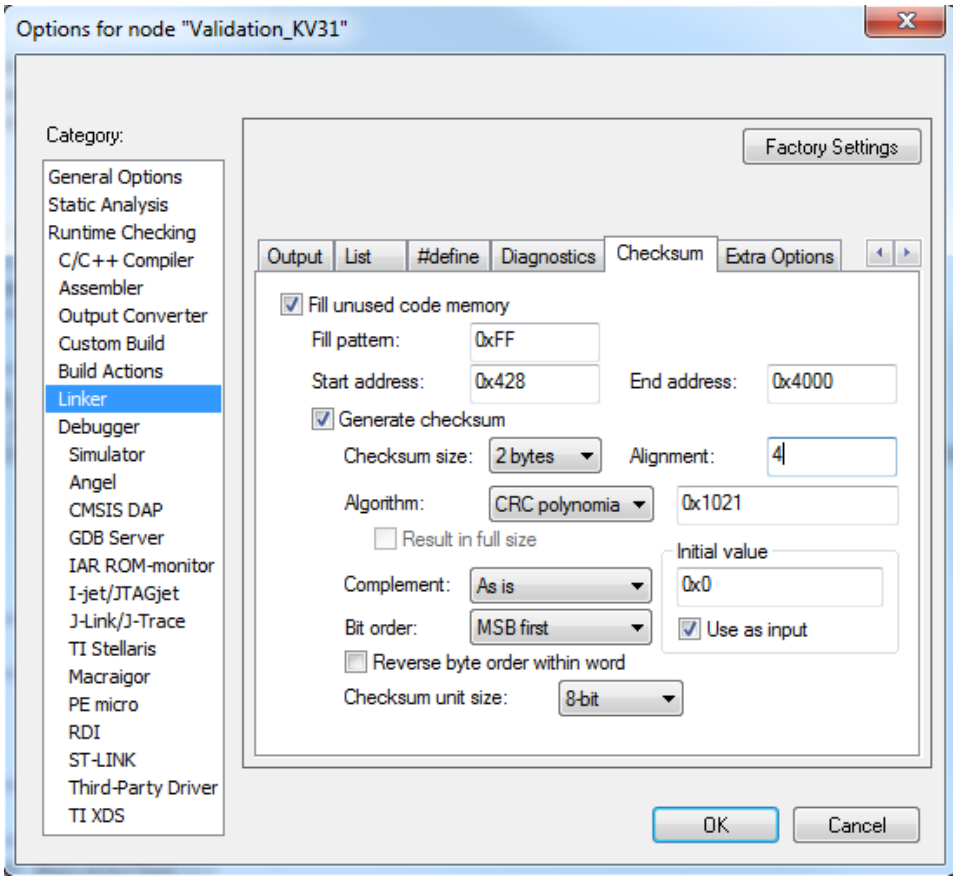

Figure 2. Basic linker setup for checksum calculation

2. Define a variable that contains the result of the checksum calculation in this block: *Options>Linker>Input>Keep symbols*. In this example, it is called *__checksum*.
3. Adjust the linker configuration file. The checksum must be stored in a different memory area than the area it is calculated for. The symbols define a small area to place the checksum result to.

```
define symbol __FlashCRC_start__                    = 0x3fff0;
define symbol __FlashCRC_end__                      = 0x3ffff;
```

4. Define a region terminated by the symbols. Define a block with its own section that is to be placed into the defined region.

```
define region CRC_region  = mem:[from FlashCRC_start  to  FlashCRC_end];
define block CHECKSUM {section checksum};
place in CRC_region   {block CHECKSUM};
```

5. Reference the section and the variable in the main program.

```
#pragma section = ".checksum"
#pragma location = ".checksum"
extern unsigned short const  checksum;  // calculated by linker
```

The second option provides a great advantage of adjusting the addresses of the checked region automatically for the linker and the application.

1. The *Fill unused code memory* option remains unchecked. Write this into *Options>Linker>Extra Options*:

```
--place_holder ielftool_checksum,2,.checksum,4
--keep=ielftool_checksum
```

- It reserves two bytes for the *ielftool_checksum* in the *.checksum* section. The alignment is set to 4.

2. The code in the linker configuration (related to the checksum region) is the same as in the previous approach. Section 2.1.1, “Code and data placement in ROM” takes its place here. For the linker configuration file code, see Section 2.1.1, “Code and data placement in ROM”.
3. Type this code into the *Options>Build Actions>Post Build* command line:

```
ielftool --fill 0xFF;c_ui32ChecksumStart-c_ui32ChecksumEnd+3
--checksum __checksum:2,crc16,0x0;c_ui32ChecksumStart-c_ui32ChecksumEnd+3
--verbose "$TARGET_PATH$" "$TARGET_PATH$"
```

- This fills the unused memory with 0xFF and calculates the checksum for the data area defined from the start address to the end address + 3. Note that the start addresses are 32 bits long and + 3 gives the address of the last byte within the 32-bit end address. This code also defines the checksum algorithm.

4. The reference in the main program can be as follows. If the start and end symbols are not referenced and used in the main program, a post-build error appears.

```
const uint32_t c_ui32ChecksumStart @ "checksum_start_mark";
const uint32_t c_ui32ChecksumEnd   @ "checksum_end_mark";
ui32Iec60730bStartAddress = (uint32_t)&c_ui32ChecksumStart;
ui32Iec60730bEndAddress = (uint32_t)&c_ui32ChecksumEnd;
ui32Iec60730bSize = ui32Iec60730bEndAddress - ui32Iec60730bStartAddress +4;
```

NOTE

- range = *checksum_start* - *checksum_end* + 3
- size = *checksum_start* - *checksum_end* +4

The disadvantages of this solution include a more difficult configuration and the fact that the start and end marks take two memory places. The advantage of this solution is that the range of the checked memory is adjusted automatically, and, together with the proper code and data placement in the invariable memory, the checking is as effective as possible.

2.3. Stack and blocks reserved for safety tests

When building a safety application, the settings for the stack must be customized as well. Controlling the stack for the underflow and overflow conditions during the application runtime is very useful. To do so, allocate two blocks of memory (above the stack and below the stack). These must have the same size and must be filled by the same pattern. The size of blocks can be adjusted (the minimum size is four bytes). A bigger size is safer, but too big a size is a wasteful use of the RAM. For more details, see the referenced stack test document. The example of a related code from the linker configuration file is shown below.

NOTE

The first lines illustrate the stack with the related addresses (variables) and other blocks that are used for safety purposes.

```
//    |             | --> data region
//    |             |
//    |_____________|
//    |_____________| --> STACK_TEST_P_1     ....ADR
//    |_____________|                        ....ADR + 0x4
//    |_____________|                        ....ADR + 0x8
//    |_____________| --> STACK_TEST_P_2     ....ADR + 0xC
//    |             |
//    |             |
//    |             |
//    |   STACK     |
//    |             |
//    |             |
//    |             |
//    |             |
//    |_____________| --> __BOOT_STACK_ADDRESS
//    |_____________| --> STACK_TEST_P_3
//    |_____________|
//    |_____________|
//    |_____________| --> STACK_TEST_P_4
//    |_____________| --> SAFETY_ERROR_CODE
//    |_____________| --> PC_test_flag
//    |             | --> WD_TEST_BACKUP
//    |_____________|
//    |             |
//    |             | --> RAM_TEST_BACKUP
//    |_____________|
```

1. The following symbols define the stack size, the stack test block size, and the RAM test backup size. The symbols are exported, so they can be recognized and used in the main program as well.

```
define symbol __size_cstack__                  = 0x0400;
define exported symbol stack_test_block_size   = 0x10;
define exported symbol ram_test_backup_size    = 0x20;
```

2. The following symbols are defined by a calculation according to the defined sizes and the physical memory size.

```
define exported symbol m_ram_test_backup       = 0x2000FFFF - ram_test_backup_size + 0x1;
define exported symbol m_wd_test_backup        = m_ram_test_backup - 0x10;
define exported symbol m_pc_test_flag          = m_wd_test_backup - 0x4;
define exported symbol m_safety_error_code     = m_pc_test_flag - 0x4;
define exported symbol m_stack_test_p_4        = m_safety_error_code - 0x4;
define exported symbol m_stack_test_p_3        = m_stack_test_p_4 - stack_test_block_size
+0x4;
define exported symbol __BOOT_STACK_ADDRESS    = m_stack_test_p_3 - 0x4;
define exported symbol m_stack_test_p_2        = BOOT_STACK_ADDRESS - __size_cstack__ -0x4;
define exported symbol m_stack_test_p_1        = m_stack_test_p_2 - stack_test_block_size;
define symbol m_data_2_end                     = m_stack_test_p_1 - 0x1;
```

Notice the blocks behind the stack and its test region. In this example, there are three restricted areas:

- The area for the program counter test flag
- The area for the watchdog test variables
- The backup area for the RAM test

These areas are placed at the end of the RAM, even behind the stack. They are not covered by the defined RAM region. It is so because these areas are not to be touched by the startup code. They must keep their content after any non-power-on resets. Their usage is described in the respective sections.

3. Initialization Phase

This is another essential part of safety applications. The application behavior during the initialization phase must be exactly defined and clearly identifiable. It must be different after the power-on reset and after the resets caused by other sources. This section covers also the startup code and the safety tests before the application runtime.

3.1. Watchdog

The watchdog is an important safety mechanism that must be implemented in every safety application. By default, the watchdog is enabled after each reset. It must not be disabled at any state of the application. If the timeout is not long enough, the watchdog must be refreshed also during the startup and initialization phases.

As a part of the safety mechanism, the watchdog itself must be tested as well. The common concept of the watchdog test and its usage is as follows:

1. The macros from the configuration header file are:

```
#define WATCHDOG_ON 1
#define Watchdog_refresh WDOG->REFRESH = 0xA602;WDOG->REFRESH = 0xB480

#define WATCHDOG_BUS_CLOCK        0
#define WATCHDOG_WINDOW
#define WATCHDOG_PRESCALE         1
#define ENDLESS_LOOP_ENABLE       0
#define WATCHDOG_RESETS_LIMIT     1000
#define WATCHDOG_TIMEOUT_VALUE    200
#define WATCHDOG_TEST_TOLERANCE   30
#define WATCHDOG_REFRESH_RATIO    20
```

2. The structure of the watchdog test variables is defined in the *IEC60730_B_CM4_CM7_wdg.h* file:

```
typedef struct {
  unsigned long counter;
  unsigned long resets;
  unsigned long wd_test_uncomplete_flag;
}volatile WD_Test_Str;
```

3. The definition of the variable of the *WD_Test_Str* type must be placed in a region that is not touched by the startup code. See Section 2.3, “Stack and blocks reserved for safety tests”.

```
extern uint32_t m_wd_test_backup;   /* from Linker configuration file */
const uint32_t c_ui32WdBackupAddress = (uint32_t)&m_wd_test_backup;
#define WATCHDOG_TEST_VARIABLES ((WD_Test_Str  *)  c_ui32WdBackupAddress)
```

4. At the very beginning of the application, configure the watchdog. This configuration must be final. Here is an example of the watchdog configuration function:

```
void ClassBWatchdogUpdate(unsigned long u32Timeout,
                          unsigned long u32Prescaler,
```

```
                        Watchdog_clock_t eClockSource)
{
 unsigned short u16TimeoutLow;
 unsigned short u16TimeoutHigh;
 unsigned short u16Prescaler;

 /* split timeout value into two 16-bit variables */
 u16TimeoutLow = (unsigned short)u32Timeout;
 u16TimeoutHigh = (unsigned short)(u32Timeout >> 16);

 if(u32Prescaler >= 1)  /* arrange the prescaler to fit the potential prescale definition */
   u32Prescaler = u32Prescaler - 1;

 u16Prescaler = (unsigned short)u32Prescaler;
 WDOG->UNLOCK = 0xC520;         /* unlock sequence */
 WDOG->UNLOCK = 0xC520;         /* unlock sequence */
 asm("nop");                    /* must wait at least one cycle after unlocking */
 WDOG->TOVALL = u16TimeoutLow;  /* timeout low value */
 WDOG->TOVALH = u16TimeoutHigh; /* timeout high value */
 WDOG->PRESC = WDOG_PRESC_PRESCVAL(u16Prescaler); /* divides the clock for watchdog timer */

 /* WD enabled. WD enabled in CPU Wait mode, CPU Stop mode, updates allowed. */
 WDOG->STCTRLH = ( WDOG_STCTRLH_WAITEN_MASK
                 | WDOG_STCTRLH_STOPEN_MASK
                 | WDOG_STCTRLH_WDOGEN_MASK
                 | WDOG_STCTRLH_ALLOWUPDATE_MASK
                 | WDOG_STCTRLH_CLKSRC(eClockSource));
}
```

5. Refresh the watchdog after calling this function and during the startup and initialization phases of the application.
6. Configure the independent timer clocked by an independent clock source before the watchdog test occurs. The LPTMR is used in this example.

```
MCG->C1 |= MCG_C1_IRCLKEN_MASK;   /* MCGIRCLK active */
MCG->C2 &= (~MCG_C2_IRCS_MASK);   /* slow reference clock selected */

SIM->SCGC5 |= SIM_SCGC5_LPTMR_MASK;                 /* enable clock gate to LPTMR */
LPTMR0->CSR = 0;                                    /* time counter mode */
LPTMR0->CSR = LPTMR_CSR_TCF_MASK|LPTMR_CSR_TFC_MASK; /* CNR reset on overflow */
LPTMR0->PSR |= LPTMR_PSR_PBYP_MASK;    /* prescaler bypassed, clock directly feeds counter */
LPTMR0->PSR &= (~LPTMR_PSR_PCS_MASK);               /* clear prescaler clock */
LPTMR0->PSR |= LPTMR_PSR_PCS(0);                    /* select the clock input */
LPTMR0->CMR = 0;                                    /* clear the compare register */
LPTMR0->CSR |= LPTMR_CSR_TEN_MASK;                  /* enable timer */
```

7. The watchdog test must be the first safety test within the application. It is called only after the POR (after the start of the application).

```
if(RCM_SRS0_WDOG_MASK ==(RCM->SRS0 & RCM_SRS0_POR_MASK)) {
  IEC60730B_CM4_CM7_watchdog_test_setup(WATCHDOG_TEST_VARIABLES); }
```

8. The purpose of this function is to initialize the watchdog reset using the wait loop and measure the time needed to reset all at once. The time to reset is measured using the LPTMR. The test setup function must periodically (in a wait loop) store the value of the independent timer into a dedicated memory place (*WATCHDOG_TEST_VARIABLES->counter*).

9. When the expected watchdog reset occurs, this test setup function is not called again. The watchdog test check function must be called after the watchdog reset and after each non-POR reset. This function uses three variables stored in the specified RAM area. The first variable contains the independent counter value. It is the last value stored within the wait loop. The check function compares this value to the expected limit values (calculated in the pre-process). The second variable is the watchdog reset counter which is incremented within this function. The number limit of the watchdog resets must be defined by the application developer. The third variable is the watchdog test uncomplete-flag. It is set to 1 if the check function is called after any reset other than the watchdog reset. Its purpose is to recognize whether the watchdog test was performed correctly or not.

```
if(RCM_SRS0_WDOG_MASK !=(RCM->SRS0 & RCM_SRS0_POR_MASK))
{
  IEC60730B_CM4_CM7_watchdog_test_check(u32WdTestLimitHigh,
                                        u32WdTestLimitLow,
                                        WATCHDOG_RESETS_LIMIT,
                                        ENDLESS_LOOP_ENABLE,
                                        WATCHDOG_TEST_VARIABLES);
}
```

NOTE

The conditions from the previous code sample must be changed when debugging. The source code for the watchdog test setup function and the watchdog test check function is located in the NXP safety library.

If the endless loop within the check function is enabled, it waits for a reset in case of a watchdog test fault.

3.2. After-reset tests

In the standard safety applications, all the possible tests are performed after any reset or before the runtime, respectively. The watchdog test is a special case and is described in the previous section. The clock test and the program flow check are usually not performed before the runtime. The stack overflow/underflow test can be also omitted. The first phase related to the after-reset safety tests is the initialization. The initialization is common for the after-reset and runtime tests. However, it is only needed for some of the tests.

According to the Class B requirements, only a single failure case is considered. When performing a test, it is assumed that everything else is functional and running properly. This approach requires a certain order of the executed tests. The first test to perform is usually the CPU register test, with the general-purpose registers being tested first. When the registers are tested, the PC register test must take place, followed by the ROM and RAM tests (the order of the tests that follow after them is independent).

4. Runtime Phase

There are many things to consider when creating the runtime phase of a safety application. The timing is very important. The possibilities depend on the device or hardware available. A good practice is to drive the whole application (CPU, interrupts, exceptions, tasks, and so on) by one common clock source.

You must have an independent clock source available to perform the clock test. The watchdog timer must be driven by an independent clock source as well.

Check whether the safety application goes through all vital parts of the code or not. There are several program flow check techniques developed for this purpose. One of them is described later on in a standalone section.

Most safety applications are based on periodical control. The periodical control is achieved using interrupts. Interrupts are the safety-critical parts of the application. Their handling and occurrence must be checked as well. This issue is described in detail in Section 6.3, “Interrupt handling and execution, program flow”. It is also associated with the clock test and watchdog operations.

5. Application Behavior in Case of Safety Error

The definitions state that when a safety error is identified, the application must be put into a safe state. It is up to you what you do with this. The most important thing is that the solution must be accepted by the certification body.

One acceptable approach is to jump into the wait loop in case of a safety error, wait for the watchdog reset, and then perform the test again (in the after-reset phase). If the error still appears, the application can’t continue because a safe behavior is not guaranteed. This mechanism must be customized in some cases; for example, in case of some CPU registers, PC, and other. The customization is described in the respective sections. Using this approach, the solution may look like this:

```
extern uint32_t m_safety_error_code; /* from Linker configuration file */
const uint32_t c_ui32SafetyErrorCodeAddress = (uint32_t)&m_safety_error_code;
#define SAFETY_ERROR_CODE ((uint32_t *) c_ui32SafetyErrorCodeAddress)

void SafetyErrorHandling(safety_common_t *psSafetyCommon)
{
  *SAFETY_ERROR_CODE = psSafetyCommon->ui32SafetyErrors;
#ifdef SAFETY_ERROR_ACTION
  __disable_irq();
   while(1);
#endif
}
```

If possible, you may implement storing of error codes into the memory (RAM or ROM). It is useful when diagnosing or servicing the application. In the above example, the error code is stored into the part of RAM that is not rewritten after the startup code.

6. Components

This section contains information about the tests of components that result from the Class B requirements. For more details about implementing these tests using the core self-test library, see the library documentation.

6.1. CPU registers

The CPU registers test is usually the second test in the queue after the watchdog test. This test does not need an initialization and is the same in the after-reset and runtime phases. However, there are some issues that must be taken into consideration when performing the runtime testing. The most important thing is that some register tests can't be interrupted. If they are interrupted, the application can either end up in an undefined state or incorrectly signalize a safety error. To avoid this, the testing must be done with the interrupts disabled or within an interrupt of the highest priority. For better and easier implementation, the CPU registers test is split into more functions in the core self-test library. The respective functions can be called within a high-priority interrupt and do not cause a long delay.

However, there are two exceptions. The CONTROL register can't change the values of all its active bits when the application is in an interrupt. Because the test of this register can't be interrupted, the safest and probably the best practice is to disable the interrupts when this register is tested. Choose the same approach also for the SP_PROCESS register test because this register is dependent on the CONTROL register.

All registers are tested for the "stuck at" condition, which fulfills the requirements of the IEC60730 standard for Class B components. The philosophy of register testing is that every register is filled with a specific pattern and then compared either to a constant or to another register. The register is then filled with the inverted pattern and compared again. If the compared values do not match, the function returns a code that specifies the safety error. A developer-defined action is then performed to put the application into a safe state. For example, it can be waiting for the watchdog reset and let the test to be performed again. If the error still appears, the application can't continue to perform its tasks.

NOTE

Because the CPU registers are the base components of the CPU the mentioned action can't be performed when some of the registers that participate in that action are corrupt. Some of the functions contain wait loops inside them and wait for the watchdog reset without performing the standard operations before it. For more information, see *IEC60730B_CM4_CM7_2.0 Release Notes* (document IEC6073B20RN).

The example of using one of the CPU register test functions is as follows:

```
psSafetyCommon->IEC60730B_cpu_reg_test_result = IEC60730B_CM4_CM7_CPU_RegisterTest();
if(psSafetyCommon->IEC60730B_cpu_reg_test_result == IEC60730B_ST_CPU_REGISTER_FAIL)
  {
   psSafetyCommon->ui32SafetyErrors |= CPU_REGISTERS_ERROR;
   SafetyErrorHandling(psSafetyCommon);
  }
```

6.2. Program Counter (PC)

The PC is technically a CPU register, but, in terms of functional safety, it is a standalone component. The PC is tested for the "stuck at" condition as well. The PC itself has a specific functionality. If a value is written into it, it tries to reach the address given by this value. The testing of the PC is often limited. There are two test approaches. The first approach is certified for use with the CM0+ and CM4 core-based devices. The second approach is certified for the CM7 core-based devices. The first approach can't be implemented on the CM7 devices because they have a different memory mapping.

NOTE

The PC is a very specific CPU component. Most of the time, its failure causes an immediate crash of the application. The two approaches introduce probably the best way of testing the PC within the scope of the Class B requirements and possibilities.

The first approach (for the CM0+ and CM4 cores) is described as follows:

Use two addresses within the RAM. These addresses must be close to the address number 0x2000 0000. That enables you to test 29 of 31 active bits of the PC register for the “stuck at” faults.

1. The area between 0x1FFF FFF8 and 0x2000 0017 must be reserved for the test. In the linker configuration file, it is omitted from the RAM region.

```
define symbol m_data_end                        = 0x1ffffff7;
define symbol __region_RAM_PCTEST_start__       = 0x1ffffff8;
define symbol __region_RAM_PCTEST_end__         = 0x20000017;
define symbol m_data_2_start                    = 0x20000018;
```

2. Define a specific section to reside in the area dedicated for the PC test.

```
define region PCTEST_region = mem:[from __ICFEDIT_region_RAM_PCTEST_start__ to
__ICFEDIT_region_RAM_PCTEST_end__];

define block PCTEST_block { section .pctest };
place in PCTEST_region   { block PCTEST_block };
```

3. In the main program, define the structure of variables which are going to reside in the PC test dedicated area. This is not mandatory, but it helps to ensure that the PC test is always performed under correct conditions. The dedicated memory is occupied by the specific variables so the addresses can’t be allocated for another data or code.

```
#pragma section = ".pctest"

typedef struct _pc_test_memory
{
 uint32_t ui32Address0; /* 0x1FFFFFF8*/
 uint32_t ui32Address1; /* 0x1FFFFFFC*/
 uint32_t ui32Address2; /* 0x20000000*/
 uint32_t ui32Address3; /* 0x20000004*/
 uint32_t ui32Address4; /* 0x20000008*/
 uint32_t ui32Address5; /* 0x2000000C*/
 uint32_t ui32Address6; /* 0x20000010*/
 uint32_t ui32Address7; /* 0x20000014*/
} pc_test_memory_t;

pc_test_memory_t sSafetyPcMemory @ ".pctest";
```

4. Define the independent variables due to data checking in the dedicated memory space.

```
typedef struct _pc_test
{
 uint32_t ui32AddrVal0;
 uint32_t ui32AddrVal1;
 uint32_t ui32AddrVal2;
 uint32_t ui32AddrVal3;
 uint32_t ui32AddrVal4;
 uint32_t ui32AddrVal5;
```

```
 uint32_t ui32AddrVal6;
 uint32_t ui32AddrVal7;
} pc_test_t;

pc_test_t g_sSafetyPcTest @ ".safety_ram";
```

5. Call the initialization function from the safety library. This function fills the dedicated memory with the code needed for the PC testing. The addresses from the dedicated memory space must be defined within the application.

```
#define IEC60730B_CFG_PC_ADDR0           0x1FFFFFF8
#define IEC60730B_CFG_PC_ADDR1           0x20000006

  IEC60730B_CM4_PC_Init(IEC60730B_CFG_PC_ADDR0, IEC60730B_CFG_PC_ADDR1);
```

6. Fill the independent variables with the variables from the dedicated memory. This step is not mandatory.

```
psSafetyPcTest->ui32AddrVal0 = psSafetyPcMemory->ui32Address0;
psSafetyPcTest->ui32AddrVal1 = psSafetyPcMemory->ui32Address1;
psSafetyPcTest->ui32AddrVal2 = psSafetyPcMemory->ui32Address2;
psSafetyPcTest->ui32AddrVal3 = psSafetyPcMemory->ui32Address3;
psSafetyPcTest->ui32AddrVal4 = psSafetyPcMemory->ui32Address4;
psSafetyPcTest->ui32AddrVal5 = psSafetyPcMemory->ui32Address5;
psSafetyPcTest->ui32AddrVal6 = psSafetyPcMemory->ui32Address6;
psSafetyPcTest->ui32AddrVal7 = psSafetyPcMemory->ui32Address7;
```

7. The test function is identical in the after-reset and in runtime phases. This test can’t be interrupted, so it must be called in the runtime within the interrupt of the highest priority or with interrupts disabled. A good practice is to check the memory content before performing the test.

```
uint32_t ui32MemoryFault;
/* test if the memory dedicated for PC test still has the same content */
ui32MemoryFault  = psSafetyPcTest->ui32AddrVal0 ^ psSafetyPcMemory->ui32Address0;
ui32MemoryFault |= psSafetyPcTest->ui32AddrVal1 ^ psSafetyPcMemory->ui32Address1;
ui32MemoryFault |= psSafetyPcTest->ui32AddrVal2 ^ psSafetyPcMemory->ui32Address2;
ui32MemoryFault |= psSafetyPcTest->ui32AddrVal3 ^ psSafetyPcMemory->ui32Address3;
ui32MemoryFault |= psSafetyPcTest->ui32AddrVal4 ^ psSafetyPcMemory->ui32Address4;
ui32MemoryFault |= psSafetyPcTest->ui32AddrVal5 ^ psSafetyPcMemory->ui32Address5;
ui32MemoryFault |= psSafetyPcTest->ui32AddrVal6 ^ psSafetyPcMemory->ui32Address6;
ui32MemoryFault |= psSafetyPcTest->ui32AddrVal7 ^ psSafetyPcMemory->ui32Address7;
```

8. The test function can be called only if the memory content is okay.

```
if(ui32MemoryFault)
  psSafetyCommon->IEC60730B_pc_test_result = IEC60730B_CM4_PC_Test(IEC60730B_CFG_PC_ADDR0,
IEC60730B_CFG_PC_ADDR1, PC_TEST_FLAG);
```

9. The function returns either the pass or fail codes, and it also works with the error flag. The flag is set to 1 at the beginning of the function and cleared to 0 at the end of the function. This is useful for the diagnostics after a PC fault when the test function is not even able to return the fail code.
10. Define the address for the PC test flag in the linker configuration file. It must be placed in the RAM, outside the block allocated for the data.

```
define exported symbol m_pc_test_flag = m_wd_test_backup - 0x4;
```

11. Reference the symbol in the main program.

```
extern uint32_t m_pc_test_flag;   /* from Linker configuration file */
const uint32_t u_ui32ProgramCounterTestFlag = (uint32_t)&m_pc_test_flag;
#define PC_TEST_FLAG ((uint32_t  *)  u_ui32ProgramCounterTestFlag
```

The implementation for the CM0+ devices is very similar. The only difference is that the PC test flag is not used here. You can use the flag, but you must set and clear it manually right before and right after the function call.

A brief description of the second approach (for the CM7 core-based devices) is as follows:

The CM7 devices have a different memory mapping. The RAM is not placed around the address 0x2000 0000. It is not even placed at the addresses that can be used for a reliable testing of the PC, as it is done in the first approach.

The principle of the test is to use one address in the ROM and one address in the RAM. Both addresses must be defined by the developer and suitable to test all of the possible PC bits. The address (place) in ROM is defined in the linker configuration file. The *IEC60730_B_CM4_CM7_pc_object.o* object must be placed to this address. This object is a part of the library. The address in RAM is passed to the test function as one of the input parameters. It can be any of the valid memory addresses. You can use different RAM addresses with multiple calls of the test function. The data from the chosen address is automatically saved and restored within the function. The function also integrates a PC test flag (as in the first approach for the CM4 devices). This test can't be interrupted. The example of the test implementation is shown below.

These are the definitions from the linker configuration file:

```
define symbol m_pc_test_start = 0x100fffe0;
define symbol m_pc_test_end = 0x100fffff;
define exported symbol m_pc_test_flag = __ICFEDIT_region_OCRAM_end__ + 0x1;
define region PC_region = mem:[from m_pc_test_start__ to m_pc_test_end__];
define block PC_TEST { section .text object IEC60730_B_CM4_CM7_pc_object.o};
place in PC_region { block PC_TEST};
```

This is the configuration for the test:

```
extern uint32_t m_pc_test_flag;   /* from Linker configuration file */
const uint32_t u_ui32ProgramCounterTestFlag = (uint32_t)&m_pc_test_flag;
#define PC_TEST_FLAG ((uint32_t  *)  u_ui32ProgramCounterTestFlag)
```

This is the testing itself:

```
IEC60730B_pc_test_result = IEC60730B_CM7_PC_Test(0x5554, IEC60730B_PC_object, PC_TEST_FLAG);
```

6.3. Interrupt handling and execution, program flow

Some tests partially participate in the interrupt handling and execution check. First of all, there must be a simple test based on the variables that are incremented in each periodic safety-related interrupt, and, in the appropriate moment, their values are compared to the predefined values. It indicates whether the interrupts occur and whether the time ratio of their occurrence is adherent or not. The example below is valid for applications with two interrupts (one faster with the highest priority) and a background loop. Let's assume that the fast interrupt is ten times faster than the slow one.

1. One global variable is incremented within the fast interrupt and cleared in the slow interrupt.

```
u32FastCtrlIsrCnt++;       // in fast interrupt
u32FastCtrlIsrCnt = 0;     // in slow interrupt
```

2. In the fast interrupt, the variable is compared to a predefined high limit. In the slow interrupt, it is compared to the predefined high limit and also to the predefined low limit. The comparisons in the slow interrupt must be done before clearing the variable.

```
if(ui32FastCtrlIsrCnt > FAST_CTRL_EXEC_PER_SLOW_CTRL)     // in fast interrupt
ui32IsrCheckRuntimeStatus = INCORRECT_EXECUTION_FAST_ISR; // in fast interrupt

if(ui32FastCtrlIsrCnt > FAST_CTRL_EXEC_PER_SLOW)          // in slow interrupt
ui32IsrCheckRuntimeStatus = INCORRECT_EXECUTION_FAST_ISR; // in slow interrupt

if(ui32FastCtrlIsrCnt < FAST_CTRL_EXEC_PER_SLOW_CTRL_MIN) // in slow interrupt
ui32IsrCheckRuntimeStatus = INCORRECT_EXECUTION_FAST_ISR; // in slow interrupt
```

3. Another global variable is incremented in the slow interrupt and cleared in the background. Because the execution of code in the background is not synchronized with the execution in the interrupts, the variable must be cleared in a suitable part of code. Depending on how often the whole code from the background is processed when compared to the execution of code in the slow interrupt, the clearing can be placed multiple times to the background or done only once after multiple background code executions.

```
ui32SlowCtrlIsrCnt++;   // in slow interrupt
ui32SlowCtrlIsrCnt = 0; // in background loop
```

4. Before clearing, the variable is compared to the predefined value. Due to asynchronous timing, the predefined value must be chosen with a large margin.

```
if(ui32SlowCtrlIsrCnt > SLOW_CTRL_EXEC_PER_SAFETY_WINDOW)
  ui32IsrCheckRuntimeStatus = INCORRECT_EXECUTION_SLOW_ISR;
```

NOTE

The failure state (when no interrupts are executed) can’t be detected by these checks. In the application, this part of the test is covered by the clock test and the watchdog operation.

The program flow check is another important test. It is not classified as a component in the IEC60730/Annex H.1 table, but it partially covers the requirements of more components from the table. Thematically, it belongs under the interrupt handling and execution. The purpose of the program flow check is to test whether the program goes through all important parts (nodes) of the software. The method used is called Control Flow Checking by Software Signatures (CFCSS).

The program flow check is processed independently in the respective threads. In this case, there are three threads: fast interrupt, slow interrupt, and background loop. Each thread contains several nodes. The number of nodes in a thread is not accurately defined. It depends on how long the code in the thread is. Another consideration is that each node takes some computing time. A node represents the short code that is placed in the appropriate place in the original application code. The signatures on the respective nodes are predefined. The differences between the neighbor node signatures (with the use of the XOR function) are calculated, or directly defined in the preprocessing phase.

Here are the examples of some signatures and differences:

```
#define SIGN_NODE_1  0x1 // 0b001
```

```
                                #define DIFF_1_2  0x3 // 0b011
#define SIGN_NODE_2  0x2 // 0b010
                                #define DIFF_2_3  0x1 // 0b001
#define SIGN_NODE_3  0x3 // 0b011
                                #define DIFF_3_4  0x7 // 0b111
#define SIGN_NODE_4  0x4 // 0b100
```

The principle of the test is to calculate the difference between the actual $node_x$ signature and the precalculated difference between $node_x$ and $node_{x+1}$. The result must be equal to the signature of $node_{x+1}$. If not, an error status must be indicated.

Here is an example of the node processing:

```
ui32ProgFlowSgn = SIGN_NODE_1; // initialization step

ui32ProgFlowSgn ^= DIFF_1_2;                          // node 1
ui32Correctness = ui32ProgFlowSgn ^ SIGN_NODE_2;      // node 1
if(ui32Correctness)                                   // node 1
  ui32ProgFlowStatus |= INCORRECT_PROG_FLOW_OPERATION; // node 1

ui32ProgFlowSgn ^= DIFF_2_3;                          // node 2
ui32Correctness = ui32ProgFlowSgn ^ SIGN_NODE_3;      // node 2
if(ui32Correctness)                                   // node 2
  ui32ProgFlowStatus |= INCORRECT_PROG_FLOW_OPERATION; // node 2
```

Keep in mind that all threads have their own nodes, node signatures, and differences. The threads are independent from other program flow check threads.

When all nodes are processed correctly, it also proves that the program goes through all the application code that is near to the nodes.

6.4. Clock

The clock testing is an important part of the Class B products. The application must periodically compare its clock source to an independent clock source, making sure that their relative rates fit into the defined limit values. The limit values dispersal can be adjusted according to the specific application. The standards state that the harmonic and subharmonic frequencies must be detected for the quartz-synchronized clock. The hardware support is really reputable for the clock test. The ideal case is to use the resources of the MCU and the application with minimum impact on the application performance. The implementation of the test for the Kinetis devices varies according to the implemented timers (RTC, LPTMR) and the available clock sources (RC oscillator, LPO, external crystal). Two clock test variants are created. Both of them use the Systick timer and the LPTMR/RTC timer. The timers must have independent clock sources. One timer generates the interrupts and the counter value from the other timer is captured within the interrupt function.

In the first test variant, the Systick timer generates periodical interrupts. The interrupt is defined by the developer and can be also used for other purposes if needed. The Systick and LPTMR/RTC timers must be initialized by the developer as well. An example of the implementation is shown below.

1. Calculate or define the limit values for the test. When the Systick interrupt occurs, the value of the LPTMR counter must fit between these limit values.
2. Call the clock test initialization function from the IEC60730 library.

```
IEC60730B_CM4_CM7_CLK_SYNC_Init(&psSafetyClockTest->ui32ClockTestContext);
```

3. Initialize the Systick timer.

```
void InitSystick(uint32_t reload_value)
{
NVIC_SetPriority(SysTick_IRQn, 3);  /* Set priority to interrupt */
SysTick->VAL = 0;
SysTick->LOAD = reload_value;
SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_ENABLE_Msk |
                SysTick_CTRL_TICKINT_Msk;
}
```

4. Initialize the LPTMR. Both the Systick and the LPTMR initialization must be processed in a very short time before the interrupts are enabled and the background loop starts.

```
void InitLPTMR(void)
{
SIM->SCGC5 |= SIM_SCGC5_LPTMR_MASK;                    /* enable clock gate to LPTMR */
LPTMR0->CSR = 0;                                       /* time counter mode */
LPTMR0->CSR = LPTMR_CSR_TCF_MASK | LPTMR_CSR_TFC_MASK;  /* CNR reset on overflow */
LPTMR0->PSR |= LPTMR_PSR_PBYP_MASK;                    /* prescaler bypassed, selected clock
                                                          directly clocks the CNR */
LPTMR0->PSR &= ~LPTMR_PSR_PCS_MASK;          /* clear prescaler clock 0 selected MCGIRCLK */
LPTMR0->CMR = 0;                                       /* clear the compare register */
LPTMR0->CSR |= LPTMR_CSR_TEN_MASK;                     /* enable timer */
}
```

5. In this application, the Systick interrupt is used for the clock check and the slow loop control. In the interrupt function, call the Safety clock test function from the IEC 60730 library.

```
IEC60730B_CM4_CM7_CLK_SYNC_LPTMR_Isr((uint32_t*)LPTMR0, &psSafetyClockTest-
>ui32ClockTestContext);
```

6. The function that evaluates the test can be called from the background loop. Make sure not to call it before the first Systick interrupt occurs.

```
psSafetyCommon->IEC60730B_clock_test_result =
IEC60730B_CM4_CM7_CLK_Check(psSafetyClockTest->ui32ClockTestContext,
                            psSafetyClockTest->ui32ClockTestLimitLow,
                            psSafetyClockTest->ui32ClockTestLimitHigh);
```

The second variant of the clock test uses the LPTMR/RTC to generate the interrupt. The timer must be triggered manually each time. It causes a delay, but can be useful when the asynchronous time events are used.

6.5. Invariable memory

The invariable memory is implemented in the Kinetis devices as the flash memory. It is often called ROM. The purpose of the invariable memory test is to check whether the data written into the flash memory are the same during the whole application lifetime. The principle of the test is to calculate a data signature (checksum) using known techniques. Firstly, the checksum is calculated in the linking phase of the application. The result is written into a dedicated place which is not in the checked area. In the after-reset and runtime phases, the checksum is calculated again and compared to the original one. The information related to the linking phase, such as the checksum calculation and data placement in the ROM (including the example code) is given in Section 2, “Linking Phase”. In the NXP safety library, the flash test is performed by one function that is written in two ways. The first way uses the CRC

peripheral module to speed up the calculation. The second way uses only the software calculation, which makes the test slower, but independent of hardware. For more information, see the respective documentation from the safety library. The implementation of the test in the application can be as follows:

1. Define the variables and parameters needed for the flash test block. The block size defines the amount of memory to be tested within one function call in the runtime.

```
#define FLASH_TEST_BLOCK_SIZE 0x100
#define IEC60730B_ST_FLASH_PASS (0x0)
#define IEC60730B_ST_FLASH_FAIL (0x00000301)

typedef struct _flash_runtime_test_parameters
{
 uint32_t ui32BlockSize;
 uint32_t ui32ActualAddress;
 uint32_t ui32PartCrc;
} flash_runtime_test_parameters_t;

typedef struct _flash_configuration_parameters
{
 uint32_t ui32Iec60730bStartConditionSeed;
 uint32_t ui32Iec60730bStartAddress;
 uint32_t ui32Iec60730bEndAddress;
 uint32_t ui32Iec60730bSize;
 uint32_t ui32Iec60730bBlockSize;
} flash_configuration_parameters_t;
```

2. Reference the checksum value that is calculated by the linker. Reference also the marks that determine the safety-related addresses in the flash memory.

```
#pragma section = ".checksum"
#pragma location = ".checksum"
extern uint16_t const __checksum; /* calculated by Linker */
const uint32_t c_ui32ChecksumStart @ "checksum_start_mark";
const uint32_t c_ui32ChecksumEnd @ "checksum_end_mark";
```

3. Initialization of the test:

```
SIM->SCGC6 |= SIM_SCGC6_CRC_MASK; /* enable clock to CRC module */

ui32Iec60730bStartConditionSeed = 0x0000;
ui32Iec60730bStartAddress = (uint32_t)&c_ui32ChecksumStart;
ui32Iec60730bEndAddress = (uint32_t)&c_ui32ChecksumEnd;
ui32Iec60730bSize = ui32Iec60730bEndAddress - ui32Iec60730bStartAddress +4;
ui32Iec60730bBlockSize = FLASH_TEST_BLOCK_SIZE;

ui32ActualAddress  = ui32Iec60730bStartAddress;
ui32PartCrc = ui32Iec60730bStartConditionSeed;
ui32BlockSize = (ui32Iec60730bSize == ui32Iec60730bBlockSize + 1) ? ui32Iec60730bSize :
ui32Iec60730bBlockSize;
```

4. After the reset, the memory test can be performed by one call of the test function. The result must have the same value as the checksum calculated by the linker.

```
IEC60730B_flash_test_result = IEC60730B_CM4_CM7_Flash_HWTest(ui32Iec60730bStartAddress,
ui32Iec60730bSize, ui32Iec60730bStartConditionSeed);
if((uint16_t)IEC60730B_flash_test_result != (uint16_t)__checksum)
```

```
  {
   ui32SafetyErrors |= FLASH_TEST_ERROR;
   SafetyErrorHandling(psSafetyCommon);
  }
```

5. The implementation in the runtime phase is up to the developer. Here is an example of a handling function that calculates the checksum in successive steps:

```
uint32_t SafetyFlashTestHandling(uint16_t const __checksum, flash_runtime_test_parameters_t
*psFlashCrc, flash_configuration_parameters_t *psFlashConfig)
{
  psFlashCrc->ui32ActualAddress += psFlashCrc->ui32BlockSize;
  if(psFlashCrc->ui32ActualAddress == psFlashConfig->ui32Iec60730bEndAddress + 4)
   {
     if((uint16_t)psFlashCrc->ui32PartCrc == (uint16_t)__checksum)
      {
        psFlashCrc->ui32PartCrc = psFlashConfig->ui32Iec60730bStartConditionSeed;
        psFlashCrc->ui32ActualAddress = psFlashConfig->ui32Iec60730bStartAddress;
        psFlashCrc->ui32BlockSize = psFlashConfig->ui32Iec60730bBlockSize;
        return (IEC60730B_ST_FLASH_PASS);
      }
     else
      {
        return (IEC60730B_ST_FLASH_FAIL);
      }
   }
  else
   {
     if (psFlashConfig->ui32Iec60730bEndAddress +0x4 - psFlashCrc->ui32ActualAddress <
psFlashConfig->ui32Iec60730bBlockSize)
      {
        psFlashCrc->ui32BlockSize = psFlashConfig->ui32Iec60730bEndAddress + 0x4 -
psFlashCrc->ui32ActualAddress;
      }
     return (IEC60730B_ST_FLASH_PASS);
   }
}
```

6. Function calling—the advantage of this test is the fact that it can be interrupted:

```
psFlashCrc->ui32PartCrc = IEC60730B_CM4_CM7_Flash_HWTest(psFlashCrc->ui32ActualAddress,
psFlashCrc->ui32BlockSize, psFlashCrc->ui32PartCrc);
if(IEC60730B_ST_FLASH_FAIL == SafetyFlashTestHandling(__checksum, psFlashCrc, psFlashConfig))
 {
  psSafetyCommon->ui32SafetyErrors |= FLASH_TEST_ERROR;
  SafetyErrorHandling(psSafetyCommon);
 }
```

6.6. Variable memory

In the Kinetis MCUs, the variable memory is represented by the RAM (more accurately SRAM). The March C and March X test techniques that are reliable for direct coupling faults detection are provided. The RAM checking is one of the most difficult Class B tests. Its execution can’t be interrupted because it would lead to the crash of the application. The test is also relatively arduous from the time consumption point of view. When implementing the RAM test, consider the implementation steps from

the beginning (the linker phase). The needed information are provided in Section 2, “Linking Phase”. Allocate the backup area in the memory and also separate the safety-related data.

The two functions that provide the RAM testing are a part of the NXP safety library. The first function is designed to test the RAM after the reset. The second function is designed for runtime testing. Define the size of the tested block. A smaller block size means a lower execution time. The RAM test can’t be interrupted. The best practice is to call the RAM test function within the interrupt of the highest priority (if possible). The RAM test function is also able to check the memory occupied by the application stack during runtime. As the stack area and the area of the safety-related data in the RAM are usually not close to each other, the test function can be called independently for two or more regions. Make sure that it uses independent variables and parameters. The implementation example is shown below.

1. Reference the section and the symbols from the linker. Refer to the related sections above.

```
#pragma section = ".safety_ram"

uint32_t *ui32SafetyRamStart = __section_begin(".safety_ram");
uint32_t *ui32SafetyRamEnd = __section_end(".safety_ram");

extern uint32_t m_ram_test_backup; /* symbol from Linker configuration file */
const uint32_t c_ui32BackupAddress = (uint32_t)&m_ram_test_backup;
```

2. Define the RAM test block size and the backup area size. The RAM test block size gives the size of a block that is tested in the runtime within one function call. The backup area size defines the size of a block in the RAM that is used to back up the data during the test. Its size must correspond to the linker configuration.

```
#define RAM_TEST_BACKUP_SIZE     0x20
#define RAM_TEST_BLOCK_SIZE      0x4
```

3. Define the variables that are used by the RAM test functions. Adding *@ “.safety_ram”* after the definition instructs the compiler to place the object into the desired section. Use it for all variables and other objects that you want to place into their own sections which are to be subjected to the RAM test.

```
typedef struct _ram_test
{
 uint32_t ui32RamTestStartAddress;
 uint32_t ui32RamTestEndAddress;
 uint32_t ui32BlockSize;
 uint32_t ui32ActualAddress;
 uint32_t ui32DefaultBlockSize;
 uint32_t ui32BackupAddress;
} ram_test_t;

ram_test_t g_sSafetyRamTest @ ".safety_ram";
```

4. Initialize the variables for the RAM test.

```
SafetyRamTestInit(&sSafetyRamTest, ui32SafetyRamStart, ui32SafetyRamEnd);

void SafetyRamTestInit(RAM_TEST_STRUCT *psSafetyRamTest,
                       uint32_t *pui32SafetyRamStart,
                       uint32_t *pui32SafetyRamEnd)
{
  psSafetyRamTest->ui32RamTestStartAddress = (uint32_t)pui32SafetyRamStart;
  psSafetyRamTest->ui32RamTestEndAddress =   (uint32_t)pui32SafetyRamEnd;
```

```
  psSafetyRamTest->ui32DefaultBlockSize =  RAM_TEST_BACKUP_SIZE;
  psSafetyRamTest->ui32BlockSize = RAM_TEST_BLOCK_SIZE;
  psSafetyRamTest->ui32ActualAddress = psSafetyRamTest->ui32RamTestStartAddress;
  psSafetyRamTest->ui32BackupAddress = (uint32_t)&RAM_TEST_BACKUP;
}
```

5. After-reset testing.

```
psSafetyCommon->IEC60730B_ram_test_result =
IEC60730B_CM4_CM7_RAM_AfterResetTest(psSafetyRamTest->ui32RamTestStartAddress,
                                     psSafetyRamTest->ui32RamTestEndAddress,
                                     psSafetyRamTest->ui32DefaultBlockSize,
                                     psSafetyRamTest->ui32BackupAddress,
                                     IEC60730B_CM4_CM7_RAM_SegmentMarchC);
```

6. Runtime testing. Call the test function in the interrupt of the highest priority, or when all interrupts are disabled.

```
psSafetyCommon->IEC60730B_ram_test_result =
IEC60730B_CM4_CM7_RAM_RuntimeTest(psSafetyRamTest->ui32RamTestStartAddress,
                                  psSafetyRamTest->ui32RamTestEndAddress,
                                  &psSafetyRamTest->ui32ActualAddress,
                                  psSafetyRamTest->ui32BlockSize,
                                  psSafetyRamTest->ui32BackupAddress,
                                  IEC60730B_CM4_CM7_RAM_SegmentMarchX);
```

NOTE

The last parameter of the test functions designates which March pattern is used for testing (March C or March X). Those are standalone functions from the IEC60730 library.

6.7. Stack

The stack test is not directly mentioned in the Class B components table in IEC 60730, but it is strongly recommended to ensure the safety. A wrong handling of the stack can lead to the crash of the application. The principle of this test is to have the exactly-defined data stored behind and in front of the area that is reserved for the stack. This data is periodically checked during the runtime. When the data changes in time, it means that the stack overflowed or underflowed. The definitions of the addresses and the memory are given in Section 2, “Linking Phase”. The remaining implementation steps are described below.

1. Reference the linker configuration file.

```
extern uint32_t m_stack_test_p_2;   /* symbol from Linker configuration file */
extern uint32_t m_stack_test_p_3;   /* symbol from Linker configuration file */
const uint32_t c_ui32StackTestFirstAddress = (uint32_t)&m_stack_test_p_2;
const uint32_t c_ui32StackTestSecondAddress = (uint32_t)&m_stack_test_p_3;
```

2. Define the pattern that is to be written into the stack test blocks. This pattern must be unique (something that usually does not appear in the application). Define the size of the blocks. This definition must correspond to the definition from the linker configuration file.

```
#define STACK_TEST_PATTERN       0x77777777
#define STACK_TEST_BLOCK_SIZE    0x10
```

3. Call the stack test initialization function.

```
IEC60730B_CM4_CM7_Stack_Init(STACK_TEST_PATTERN,
                             c_ui32StackTestFirstAddress,
                             c_ui32StackTestSecondAddress,
                             STACK_TEST_BLOCK_SIZE);
```

4. The test function can be called in the background loop.

```
psSafetyCommon->IEC60730B_stack_test_result =
IEC60730B_CM4_CM7_Stack_Test(STACK_TEST_PATTERN,
                             c_ui32StackTestFirstAddress,
                             c_ui32StackTestSecondAddress,
                             STACK_TEST_BLOCK_SIZE);
```

Perform the testing mainly during the application runtime. In the after-reset phase, the stack pointer is usually not outside the stack area. The size of the test blocks is defined by the developer. The minimum size is 4 B. The larger the blocks are, the longer the execution is, and the more memory is occupied. On the other hand, a larger block can act as a protecting case that absorbs the crash caused by the overflow and underflow. The test function can be interrupted without any risk.

7. Conclusion

NXP provides the safety Class B libraries that are backed by years of experience. The implementation and usage of such libraries is still a task that requires a certain level of knowledge. This application note provides help and guidance to developers that are new to the Class B safety.

8. References

- *KV31 Sub-Family Reference Manual* (document KV31P100M120SF7RM)
- *PMSM Control Reference Solution Package* (document PMSMCRSPUG)
- *IEC60730B_CM4_CM7_2.0 Release Notes* (document IEC6073B20RN) together with the respective library documentation
- *IAR C/C++ Development Guide*
- *ARM v7-M Architecture Reference Manual*

9. Revision History

This table summarizes the changes done to this document since the initial release:

Table 1. Revision history

Revision number	Date	Substantive changes
0	08/2016	Initial release

How to Reach Us:

Home Page:
www.nxp.com

Web Support:
www.nxp.com/support

Document Number: AN5321
Rev. 0
08/2016