Document identifier: IEC60730BHCS08LUG40
Rev. 0, 01/2020

IEC60730_B_HCS08_Library_UG_v4_0

IEC60730B Library User's Guide

Contents

Chapter 1
Core self-test library

The task of the core self-test library is to provide the functions performing the MCU core self-test. The library consists of independent functions performing tests compliant with international standards (IEC 60730, IEC 60335 UL 60730, UL 1998). The library supports the CodeWarrior IDE. The NXP core self-test library performs these tests:

- CPU registers test
- CPU program counter test
- Variable memory test
- Invariable memory test
- Stack test
- Clock test
- Digital input/output test
- Analog input/output test
- Watchdog test

The test architecture and implementation are comprehensively described in the independent sections for each test.

The library supports the S08PBxx family based on the HCS08 core.

The core self-test library has two versions: the source code version and the object code version. The object code is compiled from the common source code version. The header files are the same for both versions.

1.1 Core self-test library – object code version

The object code version of the library consists of the precompiled binary file and the same list of header files as for the source code version of the library.

Object files:

- CodeWarrior IDE: *IEC60730B_HCS08_Class_B_CW_v4_0.a*.

1.2 Core self-test library – source code version

The library name is IEC60730B_HCS08_Class_B_CW_v4_0. The main header files are *iec60730b.h* and *iec60730b_core.h*.

Each source file (**.c* or **.S*) has a corresponding header file (**.h*).

Table 1. List of library items

File name	Test type	Function name	Functions size [bytes]	Functions duration [µs]
iec60730b.h	Library header file	-	-	-
iec60730b_core.h	Core-dependent library header file	-	-	-
iec60730b_s08_aio.c	Analog I/O test	FS_AIO_InputInit()	39[1]	13,6[1]
	Analog I/O test	FS_AIO_InputTrigger()	9[1]	6,1[1]
	Analog I/O test	FS_AIO_InputSet()	63[1]	17,3[1]

Table continues on the next page...

Table 1. List of library items (continued)

File name	Test type	Function name	Functions size [bytes]	Functions duration [µs]
	Analog I/O test	FS_AIO_InputCheck()	205[1]	44,1[1]
iec60730b_s08_clock.c	Clock test	FS_CLK_Check()	50[1]	18[1]
	Clock test	FS_CLK_Init()	6[1]	3,4[1]
	Clock test	FS_CLK_RTC()	33[1]	3,9[1]
iec60730b_s08_dio_ext.c	Extended Digital I/O test	FS_DIO_InputExt()	127[1]	38,4[1]
	Extended Digital I/O test	FS_DIO_ShortToGNDSet()	120[1]	37,3[1]
	Extended Digital I/O test	FS_DIO_ShortToAdjSet()	173[1]	48,5[1]
iec60730b_s08_flash.c	Invariable Memory test (Flash)	FS_FLASH_HW16()	85[1]	See the function dedicated chapter
iec60730b_s08_pc.S	Program Counter test	FS_PC_Test()	21[1]	16.3[1]
	Program Counter test	FS_PC_Subroutine()	13[1]	-
iec60730b_s08_pc_object1.S	Program Counter test	FS_PC_Object_1()	5[1]	-
iec60730b_s08_pc_object2.S	Program Counter test	FS_PC_Object_2()	5[1]	-
iec60730b_s08_ram.S	Variable Memory test (RAM)	FS_RAM_AfterReset()	183[1]	See the function dedicated chapter
	Variable Memory test (RAM)	FS_RAM_Runtime()	152[1]	See the function dedicated chapter
	Variable Memory test (RAM)	FS_RAM_CopyMemory()	23[1]	-
	Variable Memory test (RAM)	FS_RAM_SegmentMarchC()	151[1]	-
	Variable Memory test (RAM)	FS_RAM_SegmentMarchX()	101[1]	-
iec60730b_s08_reg.S	Register test	FS_CPU_Register()	91[1]	18[1]
	Register test	FS_SP()	42[1]	12,1[1]
iec60730b_s08_Stack.S	Stack test	FS_STACK_Init()	52[1]	18,6[1]
	Stack test	FS_STACK_Test()	61[1]	23,6[1]

Table continues on the next page...

Table 1. List of library items (continued)

File name	Test type	Function name	Functions size [bytes]	Functions duration [µs]
iec60730b_s08_wdg.c	Watchdog test	FS_watchdog_setup()	56[1]	See the function dedicated chapter
	Watchdog test	FS_watchdog_check()	170[1]	46,8[1]

1.2.1 S08PBxx dedicated functions

This library is written for the S08PBxx family. Therefore, all functions mentioned in Core self-test library – source code version are written for all S08PBxx MCUs.

1.3 Functions performance measurement

This section contains remarks about the informative size and approximate time of execution for the functions. The numbers in this list are used as remark links from the corresponding chapters.

1. The function parameter was measured in the CodeWarrior 11.1. IDE on S08PB16 with an 8-MHz clock.

Chapter 2
Analog Input/Output (IO) test

The analog IO test procedure performs a plausibility check of the digital IO interface of the processor. The analog IO test can be performed once after the MCU reset and also during the runtime.

The identification of a safety error is ensured by the specific FAIL return in the case of an analog IO error. Compare the return value of the test function with the expected value. If this is equal to the FAIL return, the safety error-handling function must occur. The safety error-handling function may be specific to the application and it is not a part of the library. The main purpose of this function is to put the application into a safety state.

The principle of the analog IO test is based on sequence execution, where a certain analog level is connected to a defined analog input. The test function checks whether the converted value is within the tolerance. The test covers a check of the analog input interface and checks three reference values: reference high, reference low, and bandgap voltage. See the device specification document to set up correct values. The block diagram for the analog IO test is shown in this figure:

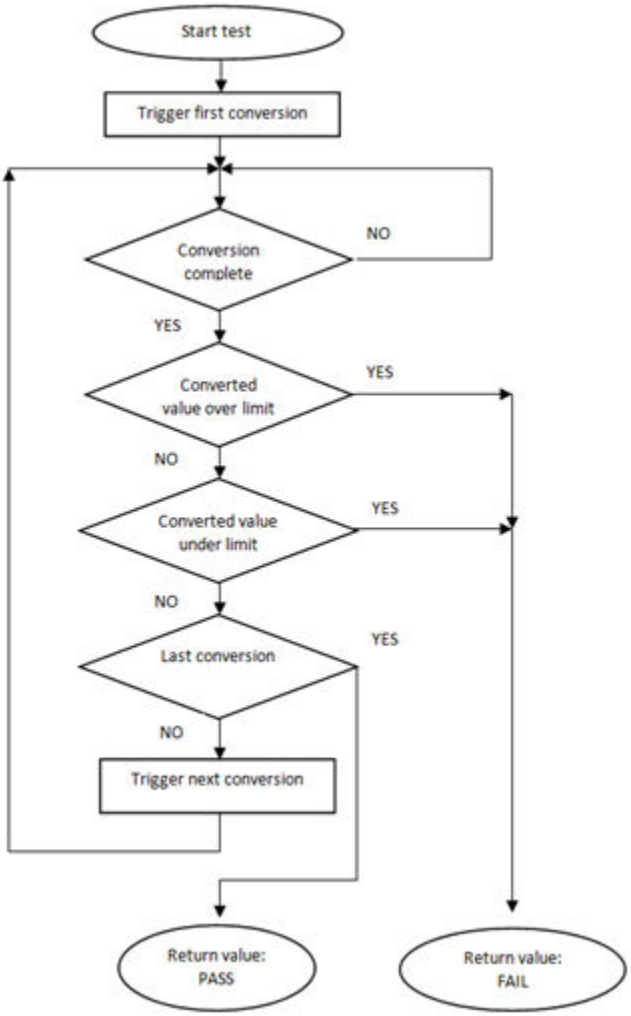

Figure 1. Block diagram for analog input test

2.1 Analog IO test in compliance with IEC/UL standards

The performed overload test fulfills the safety requirements according to the IEC 60730-1, IEC 60335, UL 60730, and UL 1998 standards, as described in this table:

Table 2. Analog IO test in compliance with IEC and UL standards

Test	Component	Fault / Error	Software / Hardware Class	Acceptable Measures
Input/Output periphery	7. Input/Output periphery (7.2 – A/D conversion)	Abnormal operation	B/R.1	Plausibility check

2.2 Analog IO test implementation

The test functions for the analog IO test are placed in the *iec60730b_s08_aio.c* file and written as "C" functions. The header file with the function prototypes is *iec60730b_s08_aio.h*. The *iec60730b.h* and *iec60730b_core.h* files are the common header files for the safety library.

These functions are called to test the analog input:

- *FS_AIO_InputInit ()*
- *FS_AIO_InputTrigger()*
- *FS_AIO_InputSet()*
- *FS_AIO_InputCheck()*

The principle of the analog input test is based on a conversion of three analog inputs with known voltage values and the test checks if the converted values fit into the limits defined by the user. The limit should be roughly 10 % around the desired reference values. The test is triggered by the FS_AIO_InputTrigger() function. The test is divided into three parts: initialization, test execution, and the end of the test.

Here is an example of the function call:

Configuration of parameters

```
#define TESTED_ADC ADC0
#define ADC_RESOLUTION 12
#define ADC_MAX ((1<<(ADC_RESOLUTION))-1)
#define ADC_REFERENCE 5
#define ADC_BANDGAP_LEVEL 1.2
#define ADC_BANDGAP_LEVEL_RAW (((ADC_BANDGAP_LEVEL)*(ADC_MAX))/(ADC_REFERENCE))
#define ADC_DEVIATION_PERCENT 10
#define ADC_MIN_LIMIT(val) (((val) * (100 - ADC_DEVIATION_PERCENT)) / 100)
#define ADC_MAX_LIMIT(val) (((val) * (100 + ADC_DEVIATION_PERCENT)) / 100)
#define FS_CFG_AIO_CHANNELS_CNT 3
#define FS_CFG_AIO_CHANNELS_LIMITS_INIT\
{\
{ADC_MIN_LIMIT(0), ADC_MAX_LIMIT(10)}, \
{ADC_MIN_LIMIT(ADC_MAX), ADC_MAX_LIMIT(ADC_MAX)},\
```

```
{ADC_MIN_LIMIT(ADC_BANDGAP_LEVEL_RAW), ADC_MAX_LIMIT(ADC_BANDGAP_LEVEL_RAW)}|
}
#define FS_CFG_AIO_CHANNELS_INIT {30, 29, 23}
```

Variables definition

```
fs_aio_test_t aio_Str;
fs_aio_limits_t FS_ADC_Limits[FS_CFG_AIO_CHANNELS_CNT] = FS_CFG_AIO_CHANNELS_LIMITS_INIT;
unsigned char FS_ADC_inputs[FS_CFG_AIO_CHANNELS_CNT] = FS_CFG_AIO_CHANNELS_INIT;
```

Initialization of the test

```
FS_AIO_InputInit(&aio_Str, (fs_aio_limits_t*)FS_ADC_Limits, (unsigned char*)FS_ADC_inputs,
FS_CFG_AIO_CHANNELS_CNT);
FS_AIO_InputTrigger(&aio_Str);
```

The test

```
for(uint8_t i=0;i<4;i++)
{
psSafetyCommon->AIO_test_result = FS_AIO_InputCheck(&aio_Str, (uint16_t *)TESTED_ADC);
switch(psSafetyCommon->AIO_test_result)
{
case FS_AIO_START:
FS_AIO_InputSet(&aio_Str, (uint16_t *)TESTED_ADC);
break;
case FS_AIO_FAIL:
psSafetyCommon->ui32SafetyErrors |= AIO_TEST_ERROR;
SafetyErrorHandling(psSafetyCommon);
break;
case FS_AIO_INIT:
FS_AIO_InputTrigger(&aio_Str);
break;
case FS_AIO_PASS:
FS_AIO_InputTrigger(&aio_Str);
break;
default:
__asm("NOP");
break;
}
}
```

2.2.1 FS_AIO_InputTrigger()

This function sets up the the analog input test to start the execution of the test.

Function prototype:

*void FS_AIO_InputTrigger(fs_aio_test_t *pObj);*

Function inputs:

pObj – The pointer to the analog test instance.

Function output:

Void. The function does not return a value.

Function performance:

The information about the function performance is in Core self-test library – source code version.

2.2.2 FS_AIO_InputInit()

This function initializes an instance of the analog input test.

Function prototype:

*void FS_AIO_InputInit(fs_aio_test_t *pObj, fs_aio_limits_t *pLimits, uint8_t *pInputs, uint8_t cntMax);*

Function inputs:

pObj – The pointer to the analog test instance.

pLimits – The pointer to the array of limits used in the test.

pInputs – The pointer to the array of input numbers used in the test.

cntMax – The size of the input and limit arrays.

Function output:

Void. The function does not return a value.

Function performance:

The information about the function performance is in Core self-test library – source code version.

2.2.3 FS_AIO_InputSet()

This function executes the first part of the AIO test sequence. This part sets up the ADC input channel. When the ADC converter is configured for a software trigger, this function also triggers the conversion. This function can be called when the ADC module is idle and ready for the next conversion.

Function prototype:

*FS_RESULT FS_AIO_InputSet(fs_aio_test_t *pObj, uint16_t *pAdc);*

Function inputs:

pObj - The pointer to the analog test instance.

pAdc - The pointer to the base address of the ADC module.

Function output:

This function returns a value of the unsigned long data type. It has these values:

- FS_AIO_PASS (0x00)
- FS_AIO_FAIL (0x71)

- FS_AIO_PROGRESS (0x72)
- FS_AIO_INIT (0x73)
- FS_AIO_START (0x74)

Function performance:

The information about the function performance is in Core self-test library – source code version.

2.2.4 FS_AIO_InputCheck()

This function executes the second part of the AIO test sequence. This part reads the converted analog value and check if the value fits into the pre-defined limits. The test is finished when this function reports FS_AIO_PASS or FS_AIO_FAIL.

Function prototype:

*FS_RESULT FS_AIO_InputCheck(fs_aio_test_t *pObj, uint16_t *pAdc);*

Function inputs:

pObj - The pointer to the analog test instance.

pAdc - The pointer to the base address of the ADC module.

Function output:

The function returns a value of the unsigned long data type. It can have these values:

- FS_AIO_PASS (0x00)
- FS_AIO_FAIL (0x71)
- FS_AIO_PROGRESS (0x72)
- FS_AIO_INIT (0x73)
- FS_AIO_START (0x74)

Function performance:

The information about the function performance is in Core self-test library – source code version.

Chapter 3
Clock test

The clock test tests the clock frequency of the processor for a wrong oscilator frequency. The clock test can be performed once after the MCU reset and also during the runtime.

The identification of a safety error is ensured by the specific FAIL return in the case of a clock fault. Assess the return value of the test function. If it is equal to the FAIL return, the safety error-handling function should occur. The safety error-handling function may be specific to the application and it is not a part of the library. The main purpose of this function is to put the application into a safety state.

The clock test principle is based on the comparison of two independent clock sources. If the test routine detects a change in the frequency ratio between the clock sources, a fail error code is returned. The test routine uses one timer and one periodical event in the application. The periodical event could be also an interrupt from a different timer than that already involved.

The block diagram for the clock test is shown in this figure:

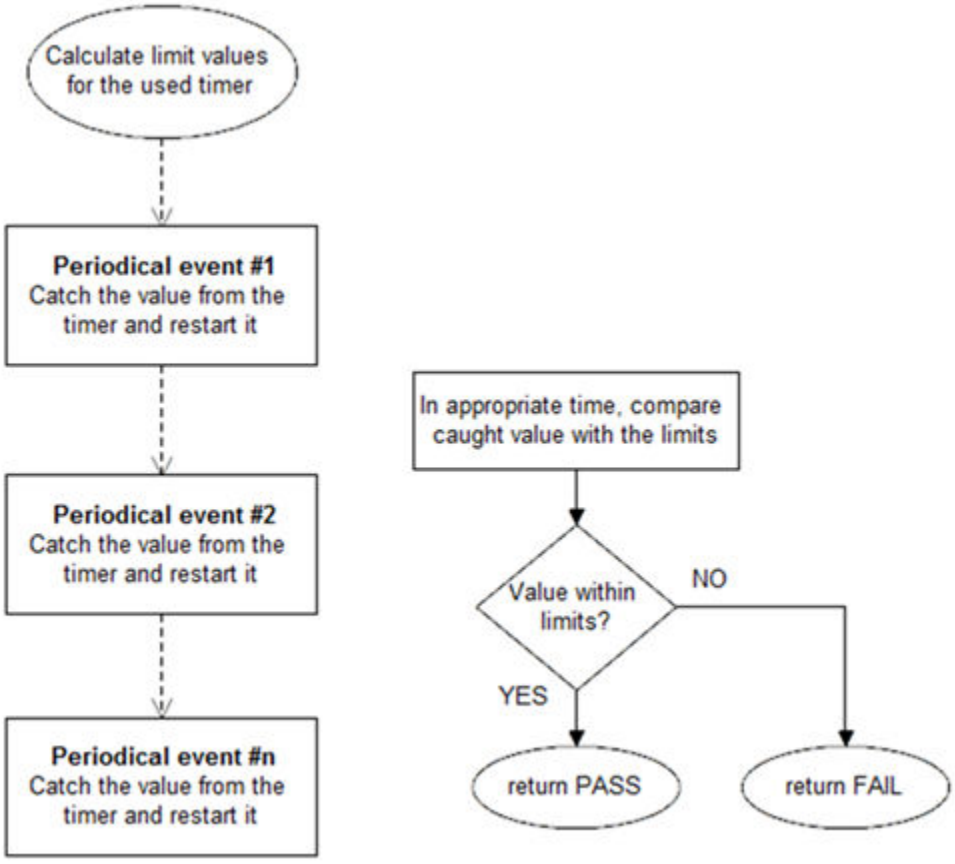

Figure 2. Block diagram for clock test

3.1 Clock test in compliance with IEC/UL standards

The performed overload test fulfils the safety requirements according to the EC 60730-1, IEC 60335, UL 60730, and UL 1998 standards, as described in this table:

Table 3. Clock test in compliance with IEC and UL standards

Test	Component	Fault / Error	Software / Hardware Class	Acceptable Measures
Clock test	3.Clock	Wrong frequency	B / R.1	Frequency monitoring

3.2 Clock test implementation

Test functions for the clock test are placed in the *iec60730b_s08_clock.c* file and written as "C" functions. The header file with function prototypes is *iec60730b_s08_clock.h*. The common library header files are *iec60730b.h* and *iec60730b_core.h*.

The following functions are called to test the clock frequency:

- FS_CLK_Init()
- FS_CLK_RTC()
- FS_CLK_Check()

Configure the reference timer, choose an appropriate periodical event, and calculate the limit values. Then declare a 16-bit global variable to store the content of the timer counter register to. The clock source of the chosen timer must differ from the clock source of the periodical event. The FS_CLK_Init() function is called once, usually before the while() loop. The FS_CLK_RTC() function is then called within a periodic event. The function to evaluate FS_CLK_Check() can be called anytime. When the test is in the initialization phase, the check function returns the “in progress” value. If the captured value from reference counter is within the preset limits, the check function returns a pass value. If not, a defined fail value is returned.

The example of a test implementation is:

```
#include “iec60730b.h”
FS_RESULT st;
unsigned long clockTestContext;
#define REF_TIMER_CLOCK_FREQUENCY 32768
#define ISR_FREQUENCY 1000 /* Hz */
#define CLOCK_TEST_TOLERANCE 20 /* % */
#define RTC_BASE 0x306A
/* There is necessary initializace RTC timer and also "timer_isr" counter */
FS_CLK_Init((uint16_t *)&clockTestContext);
while(1) { st = FS_CLK_Check(clockTestContext,FS_CLK_FREQ_LIMIT_LO, FS_CLK_FREQ_LIMIT_HI);
if(FS_CLK_FAIL == st) SafetyError();
}
void timer_isr(void)
{
FS_CLK_RTC((uint16_t*)RTC_BASE, &clockTestContext);
}
```

3.2.1 FS_CLK_Check()

This function handles the clock test. It evaluates the caught value stored in the TestContext variable with pre-defined limits. Until the first execution of the respective Isr function, the check function returns FS_CLK_PROGRESS.

Function prototype:

FS_RESULT FS_CLK_Check(uint16_t testContext, uint16_t limitLow, uint16_t limitHigh);

Function inputs:

testContext – The context value of the clock test. The type of the context variable is unsigned long.

limitLow – The lower limit for the value captured at the timer counter.

limitHigh – The higher limit for the value captured at the timer counter.

Function output:

This function returns the value of the unsigned long data type. It can have these values:

- FS_CLK_PASS (0x00)
- FS_CLK_FAIL (0x61)
- FS_CLK_PROGRESS (0x62)

Function performance:

The information about the function performance is in Core self-test library – source code version.

3.2.2 FS_CLK_Init()

This function initializes an instance of the clock sync test. It sets the "TestContext" value to the “in progress” state.

Function prototype:

*void FS_CLK_Init(uint16_t *pTestContext);*

Function inputs:

pTestContext – The pointer to the context value of the clock test.

Function output:

Void. The function does not return a value.

Function performance:

The information about the function performance is in Core self-test library – source code version.

3.2.3 FS_CLK_RTC()

This function is used only with the RTC module. This function reads the counter value from the timer and saves it into the TestContext variable. After that, it starts the RTC again.

Function prototype:

*void FS_CLK_RTC(uint16_t *pSafetyTmr, uint16_t *pTestContext);*

Function inputs:

pSafetyTmr – The pointer to the base address of the RTC module.

pTestContext – The pointer to the context value of the clock test.

Function output:

Void. The function does not return a value.

Function performance:

The information about the function performance is in Core self-test library – source code version.

Chapter 4
Digital Input/Output (DIO) test

The DIO test procedure performs a plausibility check of the digital IO interface of the processor.

The identification of the safety error is ensured by the specific FAIL return in case of the digital IO error. Assess the return value of the test function. If it is equal to the FAIL return, the safety error-handling function must occur. The safety error-handling function is specific to the application and it is not a part of the library. The main purpose of this function is to put the application into a safe state.

The DIO test functions are designed to check the digital input and digital output functionality and the short circuit conditions between the tested pin and the GND or optional adjacent pins. The execution of the DIO test must be adapted to the final application. Pay attention to the hardware connections and design. Be sure which functions can be applied to a respective pin. In most cases, the tested (and sometimes also auxiliary) pin must be reconfigured during the application run. When performing the digital output test, ensure that there is a sufficient time gap between the test arrangement and the reading of a result.

4.1 Digital IO test in compliance with IEC/UL standards

The performed overload test fulfils the safety requirements according to the IEC 60730-1, IEC 60335, UL 60730, and UL 1998 standards, as described in this table:

Table 4. Digital input/output test in compliance with IEC and UL standards

Test	Component	Fault / Error	Software / Hardware Class	Acceptable Measures
Input/Output periphery	7. Input/Output periphery (7.1 – Digital I/O)	Abnormal operation	B/R.1	Plausibility check

4.2 Digital IO test implementation

The test functions for the digital IO test are placed in the *iec60730b_s08_dio_ext.c* file. The header file with the function prototypes is *iec60730b_s08_dio_ext.h*. The *iec60730b.h* and *iec60730b_core.h* files are the common files for all components of the library.

The digital IO test can be executed using these functions:

- FS_DIO_InputExt()
- FS_DIO_ShortToGNDSet()
- FS_DIO_ShortToAdjSet()

The pointer to the fs_dio_test_t structure type is the parameter of each function. The structure is defined in the *iec60730b_s08_dio_ext.h* file.

```
typedef struct {
uint8_t PORT_PTxD; /* GPIO Pull Resistor Enable Register */
uint8_t PORT_PTxOE; /* GPIO Pull Resistor Type Select */
uint8_t PORT_PTxIE; /* GPIO Data Direction Register */
uint8_t PORT_PTxPE; /* GPIO Data Register */
} fs_dio_backup_s08_t;

typedef struct {
```

```
uint16_t p_port; /* Pointer to port module */
uint8_t port_index;
uint8_t pinNum;
uint8_t pinDir;
fs_dio_backup_s08_t sTestedPinBackup;
} fs_dio_test_s08_t;
```

This variable/variables must be initialized before calling a test function. Here is an example of initialization:

```
fs_dio_test_s08_t dio_safety_test_item_0 = /* PTC_0 */
{
/*.p_port =*/ 0, /* Pointer to port module */
/*.port_index =*/ 2, /*A = 0, B = 1, C = 2 */
/*.pinNum =*/ 0,
/*.pinDir =*/ 0
};
fs_dio_test_s08_t dio_safety_test_item_1 = /* PTB_4 */
{
/*.p_port =*/ 0, /* Pointer to port module */
/*.port_index =*/ 1, /*A = 0, B = 1, C = 2 */
/*.pinNum =*/ 4, /*.pinDir =*/ 0
};
fs_dio_test_t *dio_safety_test_items[] = { &dio_safety_test_item_0, &dio_safety_test_item_1, 0 };
```

4.2.1 FS_DIO_InputExt()

This test is used as a get function for the short-to tests. The function is applied to the pin that is already configured as a GPIO input and you know what logical level is expected during the test. The logical level can either result from the actual configuration in the application or it can be initialized (if possible) for the test. The block diagram of the FS_DIO_InputExt() function is shown in this figure:

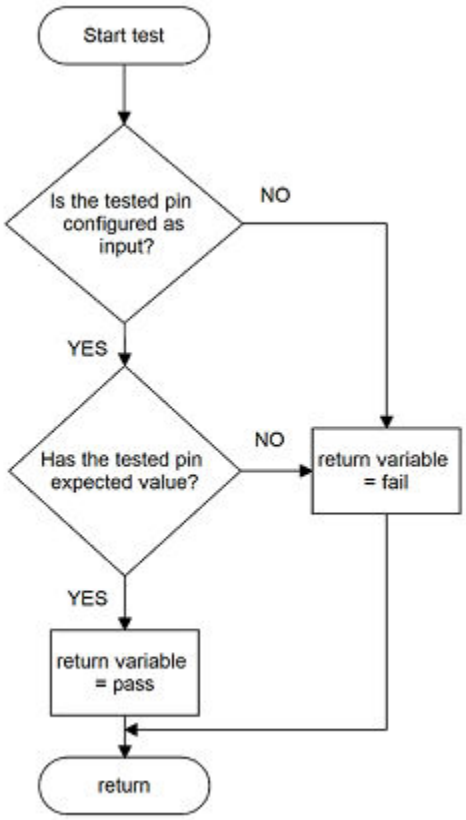

Figure 3. Extended digital input test

Function prototype:

*FS_RESULT FS_DIO_InputExt(fs_dio_test_s08_t *pTestedPin, uint8_t compareValue);*

Function inputs:

pTestedPin – The pointer to a variable of the *fs_dio_test_t* type. Specifies the tested pin.

compareValue – The value compared to the current pin value.

Function output:

The function return signalizes the result of the test. It can have these values:

- FS_DIO_FAIL (0x81)
- FS_DIO_PASS (0)

Example of function call:

fs_dio_input_test_result = FS_DIO_InputExt(&dio_safety_test_item_0, LOGICAL_ONE);

Function performance:

The information about the function performance is in Core self-test library – source code version.

Calling restrictions:

The tested pin must be configured as the GPIO input before the function call.

4.2.2 FS_DIO_ShortToAdjSet()

This function ensures the required conditions for the short to adjacent pin test. The purpose of this function is to configure the tested pin and the adjacent pin properly. The adjacent pin is an optional pin that can be theoretically shorted with the tested pin. The function block diagram is shown in the following figure. Similarly to the short to GND test, this test requires the use of two functions. The second (get) function evaluates the test result. The FS_DIO_InputExt() function is described in the respective chapter. Specify the tested pin and the adjacent pin also for the input test function.

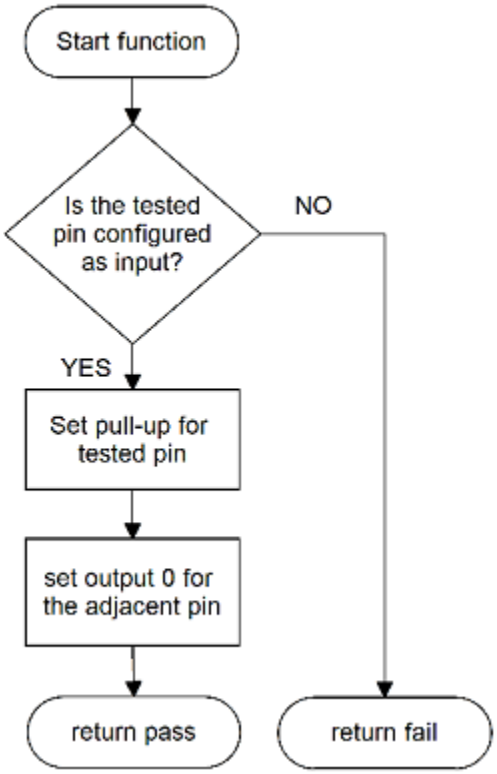

Figure 4. Block diagram of FS_DIO_ShortToAdjSet() function

Function prototype:

*FS_RESULT FS_DIO_ShortToAdjSet(fs_dio_test_s08_t *pTestedPin, fs_dio_test_s08_t *pAdjPin);*

Function inputs:

pTestedPin – The pointer to a variable of the *fs_dio_test_t* type. Specifies the tested pin.

pAdjPin – The pointer to a variable of the *fs_dio_test_t* type. Specifies the adjacent pin.

Function output:

The function returns a value of the unsigned long data type and has either of these values:

• FS_DIO_FAIL (0x81)

• FS_DIO_PASS (0)

Example of function call:

The following is an example of the short to adjacent pin test.

```
dio_short_to_adj_test_result = FS_DIO_ShortToAdjSet(&dio_safety_test_items[0], &dio_safety_test_items[1]);

dio_short_to_adj_test_result =FS_DIO_InputExt(&dio_safety_test_items[0], 1);
```

Function performance:

The information about the function performance is in Core self-test library – source code version.

Calling restrictions:

The tested pin must be configured as the GPIO input and the adjacent pin must be configured as the GPIO output before calling the function. The directions (tested pin as input, adjacent pin as output) must be configured by the user. After the end of the function, the application cannot manipulate neither the tested nor the adjacent pins, until the FS_DIO_InputExt function is called for these pins.

4.2.3 FS_DIO_ShortToGNDSet()

This function creates the first part of the short to supply test. It can be used to test the short circuit between the tested pin and the hardware ground (GND). Its block diagram is shown in the following figure. The second part of the test (result evaluation) is ensured by the FS_DIO_InputExt() function described in the respective chapter. The main purpose of the FS_DIO_InputExt() function is to set the pull-up resistor connection on the tested pin. It also ensures whether the pin is correctly configured.

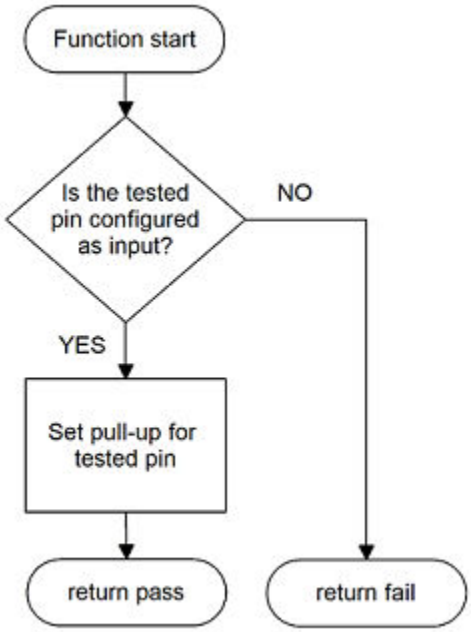

Figure 5. Block diagram of FS_DIO_ShortToGNDSet function

Function prototype:

*FS_RESULT FS_DIO_ShortToGNDSet(fs_dio_test_s08_t *pTestedPin);*

Function inputs:

pTestedPin – The pointer to a variable of the *fs_dio_test_t* type. Specifies the tested pin.

Function output:

The function returns a value of the uint16_t data type and has either of these values:

- FS_DIO_FAIL (0x81)
- FS_DIO_PASS (0)

Example of function call:

The following is an example of the test for the short-to-GND.

```
dio_short_to_test_result = FS_DIO_ShortToGNDSet(&dio_safety_test_items[0]);

dio_short_to_test_result = FS_DIO_InputExt(&dio_safety_test_items[0]), 1);
```

Function performance:

The information about the function performance is in Core self-test library – source code version.

Calling restrictions:

The tested pin must be configured as the GPIO input before calling the function. The input direction must be configured by the user. After the end of the function, the application cannot manipulate the tested pin, until the FS_DIO_InputExt function for the tested pin is called.

Chapter 5
Invariable memory test

The invariable memory on the supported HCS08 MCUs is the on-chip flash. The principle of the invariable memory test is to check whether there is a change in the memory content during the application run. Several checksum methods can be used for this purpose. The checksum is an algorithm that calculates a signature of the data placed in the tested memory. The signature of this memory block is then periodically calculated and compared with the original signature.

The signature for the assigned memory is calculated in the linking phase of an application. The signature must be saved in the invariable memory, but in a different area than the one for which the checksum is calculated. In runtime and after a reset, the same algorithm must be implemented in the application to calculate the checksum. The results are compared and, if not equal, a safety error state occurs.

The algorithm that calculates a checksum parameter (signature) in the post build phase must be set to use the 16-bit CRC polynomial (0x1021) to generate a CRC code for error detection. The same algorithm is set in the HW–CRC module. In CodeWarrior, it is possible to calculate the CRC by a linker or an external tool.

The HCS08 MCUs have a hardware CRC engine which provides an easy method of calculating the CRC of multiple bytes/words written to it. Using hardware for the invariable memory test offers better performance levels.

5.1 Invariable memory test in compliance with IEC/UL standards

The performed overload test fulfils the safety requirements according to the IEC 60730-1, IEC 60335, UL 60730, and UL 1998 standards, as described in this table:

Table 5. Invariable memory test in compliance with IEC and UL standards

Test	Component	Fault / Error	Software / Hardware Class	Acceptable Measures
Invariable memory	4.1 – Invariable memory	All single bit faults	B/R.1	Periodic modified checksum

5.2 Invariable memory test implementation

The test functions for the flash memory are placed in the *iec60730b_s08_flash.c* file and written as "C" functions. The header file with definitions and function prototypes is *iec60730b_s08_flash.h*. The *iec60730b.h* and *iec60730b_core.h* files must be placed in the application as well. These functions are located in the *iec60730b_s08_flash.c* file:

- FS_FLASH_HW16()

The FS_FLASH_HW16 function uses the hardware CRC module that is included in the supported MCUs.

5.2.1 Computing of CRC value in linking phase of application

The checksum of a memory block must be calculated before it is written into the flash memory. The checksum calculation is best done using a linker. The following example is valid only for the CodeWarrior IDE. For further details, see the CodeWarrior help.

The result of the CRC calculation must be stored in the flash memory. It must not be stored in the area where the checksum occurs. A good method is to define a small block in the flash (ROM) memory where the result of the checksum will be stored. To do this, modify the linker configuration file. The path to the linker configuration file can be found at: Project > Options > C/C++ Build >Settings >S08 Linker. The file name extension is **.prm*. In this example, an ROM_CRC segment is defined and the gap in the calculated area must be filled with "0xFF" ("FILL 0xFF").

```
SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in PLACEMENT below. */
ROM = READ_ONLY 0xC000 TO 0xE554 FILL 0xFF;
```

```
...
ROM_CRC = READ_ONLY 0xFFA0 TO 0xFFAF;
...
/* INTVECTS = READ_ONLY 0xFFB0 TO 0xFFFF; Reserved for Interrupt Vectors */
END
```

To calculate the CRC in the linker, put these settings to the **.prm* file:

```
CHECKSUM
  CHECKSUM_ENTRY
   METHOD_CRC_CCITT
   OF READ_ONLY 0xD000 TO 0xDFFF
   INTO READ_ONLY 0xFFA0 SIZE 2
   UNDEFINED 0xff
   END
END
```

This example shows that the CRC is calculated by the CCITT algorithm over the area from 0xD000 to 0xDFFF and the result is placed to adress 0xFFA0.

To access the checksum in the application, create a pointer which allows reading the checksum.

```
uint16_t * checksum = 0xFFA0; /* Address of CRC from linker *.prm file */
```

In the example above, a pointer variable checksum is filled with the address of the linker-calculated checksum in the memory.

5.2.2 Test performed once after MCU reset

When implemented after the reset or when there is no restriction to the execution time, the function call is as follows:

```
#include “iec60730b.h”
#define CRC_BASE 0x3060
#define FLASH_TEST_CONDITION_SEED 0xFFFF
uint16_t * checksum = 0xFFA0; /* Address of CRC from linker *.prm file */
if( *checksum != FS_FLASH_HW16(start_address, size, CRC_BASE, FLASH_TEST_CONDITION_SEED ))
SafetyError();
```

Where:

- *checksum* - The pointer to the CRC value computed in the linking phase of an application.
- *start_address* - The initial address of the memory block to be tested.
- *size* - The size of the memory block to be tested (first address – end address + 1).
- *CRC_BASE* - The base address of the CRC module.
- *FLASH_TEST_CONDITION_SEED* - The start condition seed (usually 0xFFFF).

5.2.3 Runtime test

In the application runtime with limited time for execution, the CRC is computed in a sequence. It means that the input parameters have different meanings in comparison with the calling after reset. The implementation example is as follows:

```
#include “iec60730b.h”
#define CRC_BASE 0x3060
#define FLASH_TEST_CONDITION_SEED 0xFFFF
uint16_t * checksum = 0xFFA0; /* Address of CRC from linker *.prm file */
flash_crc.part_crc = FS_FLASH_HW16(flash_crc.actual_address, flash_crc.block_size, CRC_BASE,
flash_crc.part_crc);
if(FS_FLASH_FAIL == SafetyFlashTestHandling(*checksum, &flash_crc))
SafetyError();
```

Where:

- *checksum* - The pointer to the CRC value computed in the post-build phase of an application.
- *flash_crc.part_crc* - The particular CRC result and the seed parameter for the next iteration.
- *flash_crc.actual_address* - The actual address of the memory block to be tested.
- *CRC_BASE* - The base address of the CRC module.
- *flash_crc.block_size* - The size of the memory block to be tested.

The handling of the function must be carried out by the application developer. When the checksum of a block is calculated in more iterations, the result from the first iteration (function call) is the seed value for the next function call. After the last part of the memory is processed with the test function, the result is the final checksum of the whole tested memory block.

5.2.4 FS_FLASH_HW16()

This function calculates the 16-bit CRC polynomial (0x1021) using the CRC module.

Function prototype:

uint16_t FS_FLASH_HW16(uint16_t startAddress, uint16_t size, uint16_t moduleAddress, uint16_t crcVal);

Function inputs:

startAddress – The first address of the memory block to be tested.

size – The size of the block to be tested.

moduleAddress – The base address of the CRC module.

crcVal – The input seed value for the calculation. It is 0xFFFF for the first iteration and the result from the previous function call for the next iterations.

Function output:

uint16_t – The checksum of the block of memory, defined with the input parameters.

Function performance:

The function size is 85 B.[1]

The function duration depends on the defined block size. Several examples are shown in this table:[1]

Table 6. Duration of FS_FLASH_HW16() in dependence of tested block size

Block size (Bytes)	Clock cycles	Execution time (approximately)
0x10	836	104.5 µs
0x20	1473	184.2 µs
0x50	3394	424.3 µs

Calling restrictions:

The function cannot be interrupted with the function that changes the content or setup of the hardware CRC module.

Chapter 6
CPU program counter test

The CPU program counter register test procedure tests the CPU program counter register for the stuck-at condition. The program counter register test can be performed once after the MCU reset and also during runtime.

Compared to the other CPU registers, the program counter cannot be simply filled with a test pattern. It is necessary to force the CPU (program flow) to access the corresponding address which is testing the pattern to verify the program counter functionality.

The program counter test works without an initialization function. Another object (the short function) is written in a separate file. This object must be placed to an appropriate address in the flash memory by declaring it in the linker configuration file. The test function uses the address of this routine and also the appropriate address in the RAM memory to test the program counter.

At the start of the test function, a flag addressed as an input parameter is set. After a successful execution of the test, this flag is cleared.

The object function should be placed to the appropriate and valid address in the memory. Usually, this setup is done in the linker. The address itself is a test pattern for the program counter register and should test as many bits of the program counter as possible.

The purpose of the previously mentioned flag is that it can be useful in the case of a program counter test failure and the subsequent diagnostics. If the failure of the program counter register is a stuck-at of one bit, everything depends on the position of that bit in the register. If the bit is a part of the address that is used before the test itself, the application is damaged.

Otherwise, if the stuck bit is caught in the test, the test itself is unable to proceed all the jumps and returns. The application will be probably damaged as well, but the flag will be set and it will prove the PC register failure in the post-process diagnostics.

The block diagrams of the program counter register test are shown in this figure:

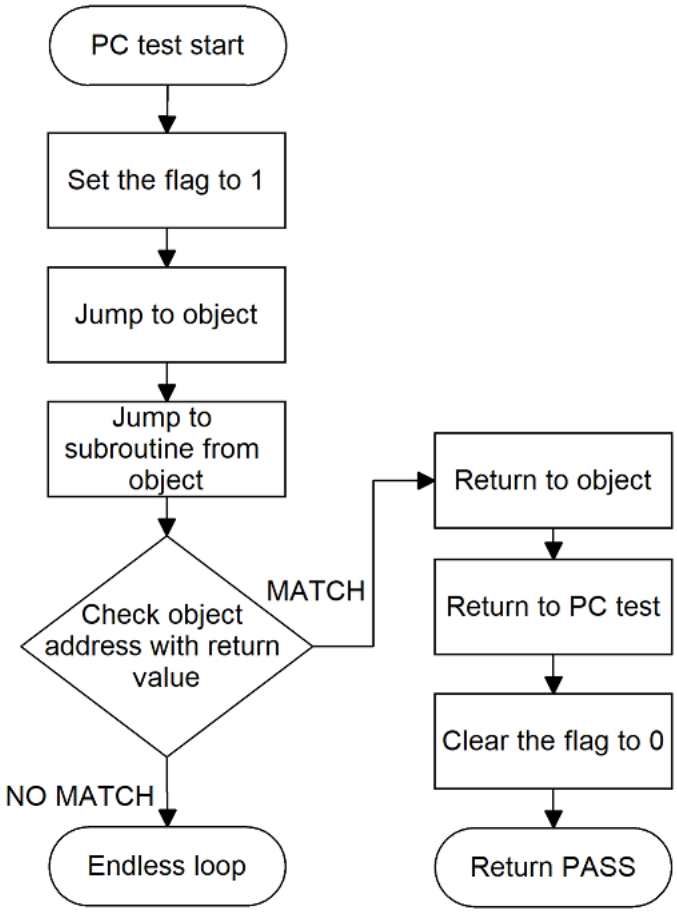

Figure 6. Block diagram of program counter test

6.1 CPU program counter test in compliance with IEC/UL standards

The performed overload test fulfils the safety requirements according to the IEC 60730-1, IEC 60335, UL 60730, and UL 1998 standards, as described in this table:

Table 7. CPU program counter test in compliance with IEC/UL standards

Test	Component	Fault / Error	Software / Hardware Class	Acceptable Measures
CPU	CPU (1.3 – Programme Counter)	Stuck at	B/R.1	Periodic self test

6.2 CPU program counter test implementation

The test functions for the CPU program counter are placed in the *iec60730b_s08_pc.S* file and written as assembler functions. The header file with the test patterns and function prototypes is *iec60730b_s08_pc.h*. The *iec60730b.h* and *iec60730b_core.h* files are included in the *iec60730b_s08_pc.S* file and must be placed in the application also. The *iec60730b_s08_pc_object.S* file must be placed to the appropriate address in the flash memory.

Implementation example of program counter test:

The only function that is handled in the application is FS_PC_Test().

Provide the appropriate parameters to the function (see FS_PC_Test()). If needed, the function can be called more times in a sequence with different object files. The *iec60730b_s08_pc_object.S* file must be placed to an appropriate address in the flash memory.

This is an example of the function call:

```
#include “iec60730b.h”

fs_pc_test_result = FS_PC_Test(FS_PC_Object, FS_PC_Subroutine, &ui8PcFlag);
if(FS_PC_FAIL == fs_pc_test_result)
SafetyError();
```

6.2.1 FS_PC_Test()

The program counter register is tested according to the block diagram in CPU program counter test.

Function prototype:

*FS_RESULT FS_PC_Test(tFcn_pc pObject_function, tFcn_pc pSubroutine, uint8_t * PC_flag);*

Function inputs:

pObjectFunction – The pointer to the function. It is always FS_PC_Object (an auxiliary function from the *iec60730b_s08_pc_object.S* file).

pSubroutine – The pointer to the FS_PC_Subroutine auxiliary function from the *iec60730b_s08_pc_object.S* file.

*PC_flag – The pointer to an address in the memory (not accessible by the startup code) used as a flag.

Function output:

typedef uint16_t FS_RESULT;

It can have two values:

FS_PC_PASS (0)

In case of incorrect test execution, PC_flag has a value of "1" and the function stays in an endless loop.

Function performance:

The information about the function performance is in Core self-test library – source code version.

Calling restrictions:

This function cannot be interrupted.

6.2.2 FS_PC_Object()

This function performs the program counter test. It should be called only by the FS_PC_Test() function. Place this function to a reliable address in the flash memory.

This example shows how to place the function to the desired address in the linker configuration file for the CodeWarrior IDE:

```
SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in PLACEMENT below. */
Z_RAM = READ_WRITE 0x0040 TO 0x00FF;
...
PC_OBJECT_1 = READ_ONLY 0xE555 TO 0xE55A;
....
```

```
/* INTVECTS = READ_ONLY 0xFFB0 TO 0xFFFF; Reserved for Interrupt Vectors */
END
PLACEMENT /* Here all predefined and user segments are placed into the SEGMENTS defined above. */
....
PC_TEST_SECTION_1 INTO PC_OBJECT_1;
...
END
```

The information for the linker that this file must be linked to a specific section must be put to the *iec60730b_s08_pc_object.S* source file.

```
PC_TEST_SECTION_1: SECTION
```

The sentence in the above example must be put to the file before the FS_PC_Object function.

Function prototype:

void FS_PC_Object(void);

Function inputs:

None.

Function output:

Void.

Function performance:

The function duration is included in the duration of the FS_PC_Test() function. Its size is 5 B.[1]

Calling restrictions:

This function performs the program counter test and should be called only by the FS_PC_Test() function.

6.2.3 FS_PC_Subroutine()

This function is used to perform the program counter test. It should be called only by the FS_PC_Object() function.

Function prototype:

void FS_PC_Subroutine(void);

Function inputs:

None.

Function output:

Void.

Function performance:

The function duration is included in the duration of the FS_PC_Test() function. Its size is 13 B.[1]

Calling restrictions:

This function is used to perform the program counter test. It should be called only by the FS_PC_Object() function.

Chapter 7
Variable memory test

The Variable memory test for the supported devices checks the on-chip RAM for DC faults. The March C and March X schemes are used as control mechanisms. Choose which scheme to use (March C or March X). The handling functions for the after-reset test and the runtime test are different. Both functions must have a backup area defined in the RAM and reserved by the application developer. The size of this area must be at least the same as the size of the tested block. A RAM test is considered destructive. This is because the data from the memory area with the variables and functions placed in the RAM is moved away, rewritten multiple times (for example, with test patterns 0x55 and 0xAA), and then moved back to the original memory area. The test procedure is very sensitive and cannot be interrupted. The block diagrams for the RAM tests are shown in these figures:

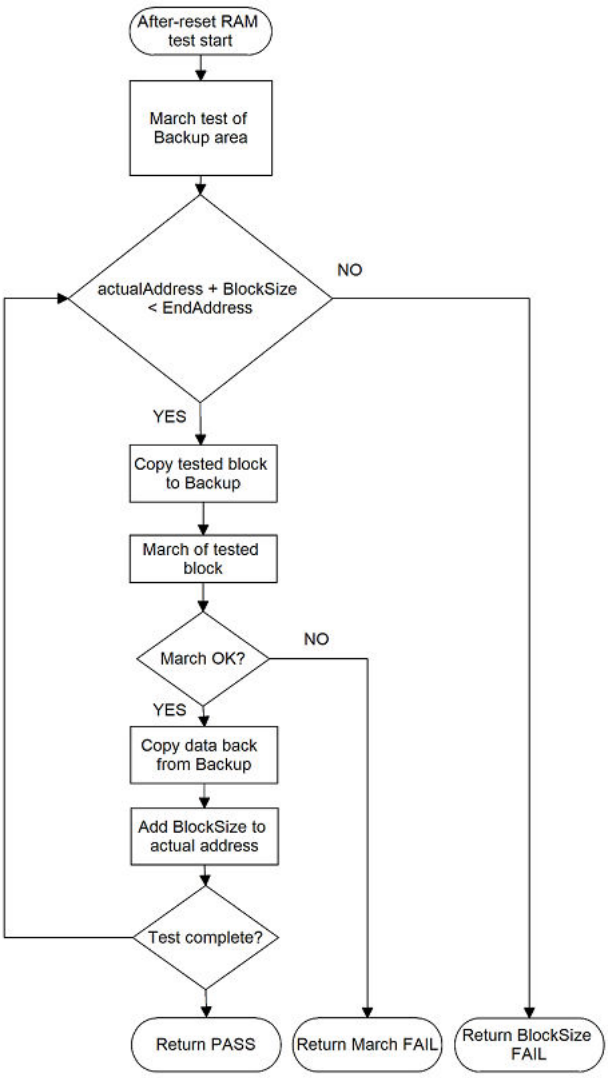

Figure 7. Block diagram for after-reset test of RAM

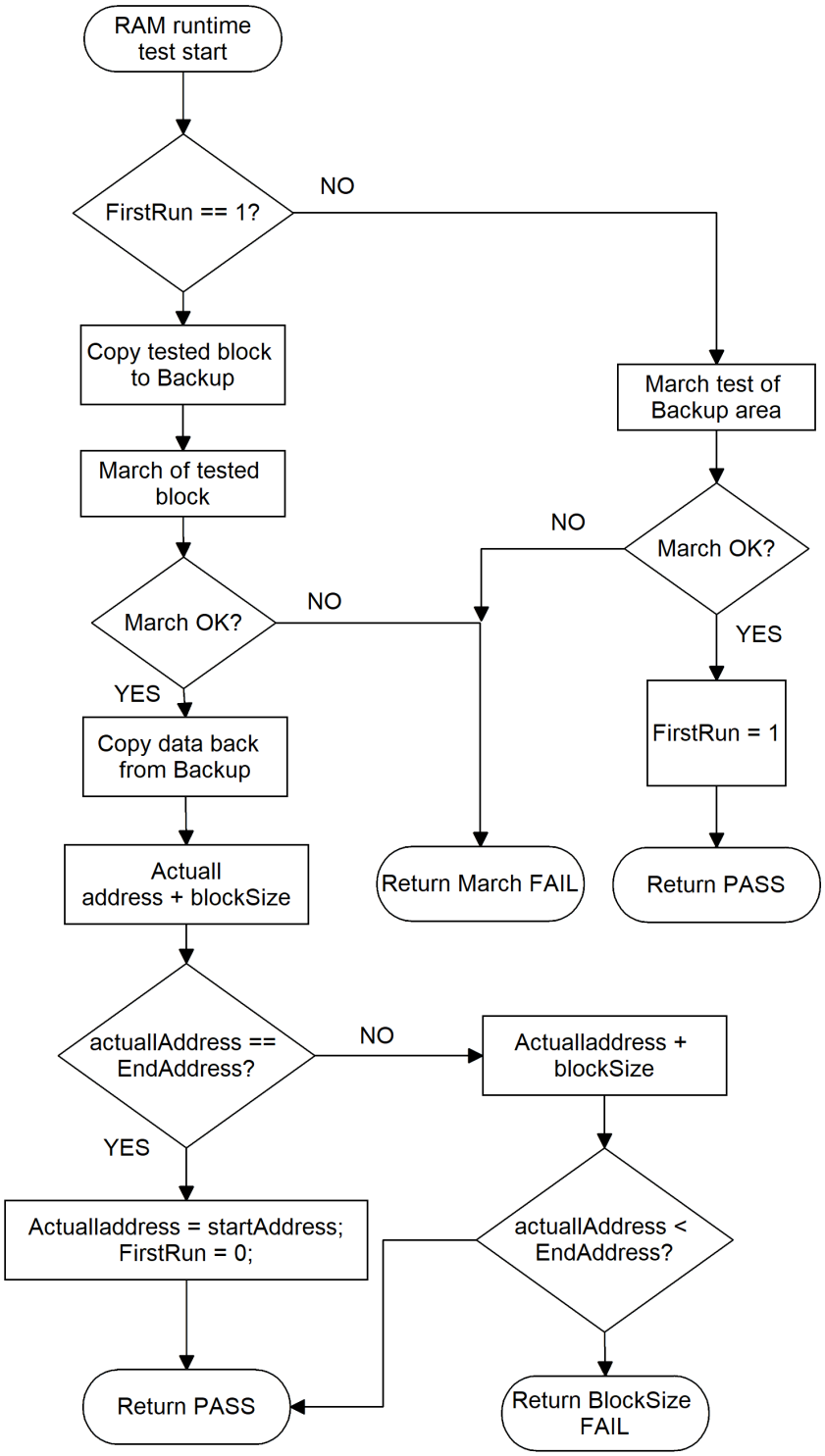

Figure 8. Block diagram for runtime test of RAM

7.1 Variable memory test in compliance with IEC/UL standards

The overload test performed fulfils the safety requirements according to the IEC 60730-1, IEC 60335, UL 60730, and UL 1998 standards, as described in this table:

Table 8. Variable memory test in compliance with IEC and UL standards

Test	Component	Fault / Error	Software / Hardware Class	Acceptable Measures
Variable memory	4.2 – Variable memory	DC fault	B/R.1	Periodic self-test using March test

7.2 Variable memory test implementation

The test functions for the variable memory (RAM) test are placed in the *iec60730b_s08_ram.S* file and written as assembler functions. The header file with the return values and function prototypes is *iec60730b_s08_ram.h.* The *iec60730b.h* and *iec60730b_core.h* files are included in the *iec60730b_s08_ram.S* file and must be also placed in the application.

The RAM test consists of these public functions:

- FS_RAM_AfterReset()
- FS_RAM_Runtime()
- FS_RAM_CopyMemory()
- FS_RAM_SegmentMarchC()
- FS_RAM_SegmentMarchX()

The first two functions provide a complex RAM test. You do not have to work directly with the next functions.

7.2.1 FS_RAM_AfterReset()

The after-reset test is done by the FS_RAM_AfterReset function. This function is called once after a reset, when the execution time is not critical. The application developer must reserve free memory space for the backup area. The block size parameter cannot be larger than the size of the backup area and also must divide the tested area without a rest. This causes the align error. The function firstly checks the backup area. Then the loop begins. Blocks of memory are copied to the backup area and their locations are checked with the respective March test. The data is copied back to the original memory area and the actual address with the block size is updated. This is repeated until the last block of memory is tested. If a DC fault is detected, the function returns a fail pattern. If the last memory block is smaller than block_size, an align error occurs. The block diagram is shown in Figure 7.

This is an example of the function call:

```
#include “iec60730b.h”
if(FS_RAM_FAIL == FS_RAM_AfterReset(start_address, end_address, block_size, backup_address, 0x55, 0xAA, FS_RAM_SegmentMarchC))
SafetyError();
```

Function prototype:

FS_RESULT FS_RAM_AfterReset(uint16_t startAddress, uint16_t endAddress, uint8_t blockSize, uint16_t backupAddress, uint8_t Pattern1, uint8_t Pattern2, uint16_t pMarchType);

Function inputs:

startAddress – The starting address for the test.

endAddress – The first address behind the tested area.

blockSize – The size of the block for test iterations. It is limited by the size of the backup area. It should be a multiple of 0x4. It must fill the tested area without a rest.

backupAddress – The address of the backup area in the RAM.

Pattern1 – The first pattern for the test (usually 0x55).

Pattern2 – The second pattern for the test, complementary to Pattern1 (usually 0xAA).

pMarchType – The pointer to the function to be used for the March test (FS_RAM_SegmentMarchC or FS_RAM_SegmentMarchX).

Function output:

typedef unsigned long FS_RESULT;

Can have three values:

FS_RAM_FAIL (0x41)

FS_RAM_ALIGN_FAIL (0x42)

FS_RAM_PASS (0)

Function performance:

The function size is 183 B.[1]

The execution time depends on the memory size. It also varies with different block sizes and different March methods used.[1]

Table 9. FS_RAM_AfterReset duration

Memory size (B)	Block size (B)	Duration - March X	Duration – March C
0x100	0x4	7.22 ms	9.13 ms
0x100	0x8	6.22 ms	8.04 ms
0x100	0x20	5.68 ms	7.58 ms
0x200	0x4	14.38 ms	18.15 ms
0x200	0x8	12.32 ms	15.92 ms
0x200	0x20	11.00 ms	14.6 ms

Calling restrictions:

This function is used once after the MCU reset, when the execution time is not critical. It cannot be interrupted. The backup area must be at least the same size as the tested block size defined by the block_size parameter. The block size parameter must divide the tested area without a rest.

7.2.2 FS_RAM_Runtime()

The runtime test is done by the FS_RAM_Runtime() function. Reserve free memory space for the backup area. The block size parameter cannot be larger than the size of the backup area and must divide the tested area without a rest. During the first call, the function checks the backup area. After the call, the blocks of memory are processed in a sequence. They are copied to the backup area and their locations are checked with the respective March test. The data is copied back to the original memory area and the actual address is updated. This is repeated until the last block of memory is tested. If a DC fault is detected, the function returns a fail pattern. If the rest of memory is smaller than block_size, an align error occurs. The block diagram is shown in Figure 8. This is an example of a function call:

```
#include "iec60730b.h"
```

```
if(FS_RAM_FAIL == FS_RESULT FS_RAM_Runtime(start_address, end_address, &actual_address,
&FirstRun, block_size, backup_address,0x55,0xAA,IEC60730B_RAM_SegmentMarchX))
SafetyError();
```

Function prototype:

*FS_RESULT FS_RAM_Runtime(uint16_t startAddress, uint16_t endAddress, uint16_t * pActualAddress,uint8_t * pFirstRun, uint8_t blockSize, uint16_t backupAddress, uint8_t Pattern1, uint8_t Pattern2, uint16_t pMarchType);*

Function inputs:

startAddress – The starting address of the test.

endAddress – The first address behind the tested area.

*pActualAddress – The pointer to the variable for the actual address. It is updated within the function.

pFirstRun – The pointer to the variable for the indication of the first execution of FS_RAM_Runtime. It is updated within the function.

blockSize – The size of the block for test iterations. It is limited by the size of the backup area. It must be a multiple of 0x4 and divide the tested area without a rest.

backupAddress – The address of the backup area in the RAM.

Pattern1 – The first pattern for the test (usually 0x55).

Pattern2 – The second pattern for the test, complementary to Pattern1 (usually 0xAA).

pMarchType – The pointer to the function to be used for the March test. (FS_RAM_SegmentMarchC, or FS_RAM_SegmentMarchX)

Function output:

typedef unsigned long FS_RESULT;

Can have three values:

FS_RAM_FAIL (0x41)

FS_RAM_ALIGN_FAIL (0x42)

FS_RAM_PASS (0)

Function performance:

The function size is 152 B.[1]

The execution time depends on the block size and is different for the March C and March X methods. [1]

Table 10. FS_RAM_Runtime duration

Block Size (B)	Duration - March X	Duration - March C
0x4	131,1 µs	160.3 µs
0x8	210.7 µs	265.9 µs
0x20	684,9 µs	896.2 µs

Calling restrictions:

The function cannot be interrupted. The backup area must have at least the same size as the tested block size defined by the block_size parameter. The block size must divide the tested area without a rest. The execution time depends on the block size.

7.2.3 FS_RAM_CopyMemory()

This function copies data from "sourceAddress" to "destinationAddress".

Function prototype:

void FS_RAM_CopyMemory(uint16_t sourceAddress, uint8_t blockSize, uint16_t destinationAddress, uint16_t PLACE_HOLDER);

Function inputs:

sourceAddress – The starting address of the memory with the data to be copied.

blockSize – The size of the block to be copied.

destinationAddress – The destination address of the memory.

PLACE_HOLDER – The dummy parameters. Due to a lack of registers, this content is ignored.

Function output:

Void. The function does not return a value.

Function performance:

The function duration is included in the duration of the FS_RAM_AfterReset()/FS_RAM_Runtime() function. The function size is 23 B.[1]

7.2.4 FS_RAM_SegmentMarchC()

This function performs the March C test of the memory block that is given by the start address and the block size. The content of the tested memory remains changed after the execution of this function.

Function prototype:

FS_RESULT FS_RAM_SegmentMarchC(uint16_t startAddress, uint8_t blockSize, uint8_t Pattern1, uint8_t Pattern2, uint16_t PLACE_HOLDER);

Function inputs:

startAddress – The starting address for the test.

blockSize – The size of the block to be tested.

Pattern1 – The first pattern for the test, usually 0x55.

Pattern2 – The second pattern for the test, complementary to Pattern1 (usually 0xAA).

PLACE_HOLDER – The dummy parameters. Due to the lack of registers, their content is ignored.

Function output:

typedef unsigned long FS_RESULT;

This function can have two values:

FS_RAM_FAIL (0x41)

FS_RAM_PASS (0)

Function performance:

The function duration is included in the duration of the FS_RAM_AfterReset()/FS_RAM_Runtime() function. The function size is 151 B.[1]

7.2.5 FS_RAM_SegmentMarchX()

This function performs the March X test of the memory block that is given by the start address and the block size. The content of the tested memory remains changed after the execution of this function.

Function prototype:

FS_RESULT FS_RAM_SegmentMarchX(uint16_t startAddress, uint8_t blockSize, uint8_t Pattern1, uint8_t Pattern2, uint16_t PLACE_HOLDER);

Function inputs:

startAddress – The starting address for the test.

blockSize – The size of the block to be tested.

Pattern1 – The first pattern used for the March test (usually 0x55).

Pattern2 – The second pattern for the test, complementary to Pattern1 (usually 0xAA).

PLACE_HOLDER – This parameter is there only due to the lack of a register (its content is ignored).

Function output:

typedef unsigned long FS_RESULT;

Can have two values:

FS_RAM_FAIL (0x41)

FS_RAM_PASS (0)

Function performance:

The function duration is included in the duration of the FS_RAM_AfterReset()/FS_RAM_Runtime() function. The function size is 101 B.[1]

Chapter 8
CPU register test

The CPU register test procedure tests all S08 CPU registers (except for the program counter register) for the stuck-at condition. The program counter test is implemented as a standalone safety routine. There is a set of tests performed once after the MCU reset and also during runtime. This set of tests includes the tests of these registers:

- Accumulator - A
- Index register (high) - H
- Index register (low) - X
- StackPointer - SP
- Condition code register - CCR

An error in the register test is not reported by the fail return, because it requires the action of a corrupted register. In that case, the function waits in an endless loop for reset.

The principle of a stuck-at error test of the CPU registers is to write and compare two test patterns in every register. The content of the register is compared with the constant. Patterns are defined to check the logical one and logical zero values in all register bits.

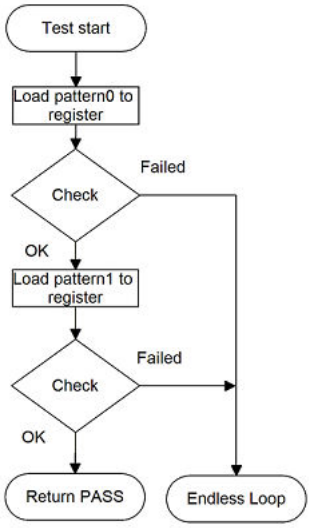

Figure 9. Block diagram for registers test

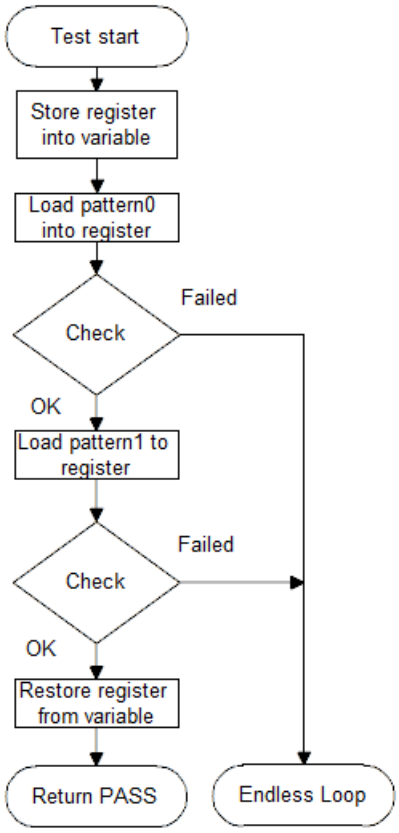

Figure 10. Block diagram for stack pointer and CCR test

8.1 CPU register test in compliance with IEC/UL standards

The performed overload test fulfils the safety requirements according to the IEC 60730-1, IEC 60335, UL 60730, and UL 1998 standards, as described in this table:

Table 11. CPU register test in compliance with IEC and UL standards

Test	Component	Fault / Error	Software / Hardware Class	Acceptable Measures
CPU registers test	CPU (1.1 – Registers)	Stuck at	B/R.1	Periodic self test

8.2 CPU register test implementation

The test functions for the CPU registers are placed in the *iec60730b_s08_reg.S* file and written as assembler functions. The header file with return values and function prototypes is *iec60730b_s08_reg.h*. The *iec60730b.h* and *iec60730b_core.h* files are included in the *iec60730b_reg.S* file and must be also placed into the application.

These functions are called to test the corresponding registers:

- FS_CPU_Register()
- FS_SP()

The error detection is provided by the endless loop instead of returning an error value. If some tested registers are corrupted, the application cannot make standard operations to identify the safety error (to compare something, to move out from the function, or to return a value).

The use of functions is the same after the reset and during runtime. The developer must pay attention to the functions during runtime, as described in the following sections.

This is an example of a function call:

```
#include “iec60730b.h”
uint16_t fs_reg_backup; /*This global variable is used by function to backup register content */
psSafetyCommon->CPU_reg_test_result = FS_CPU_Register(); /* Test of Accumulator, H:X and CCR */
psSafetyCommon->CPU_sp_test_result = FS_SP(); /* Test stack pointer */
```

8.2.1 FS_CPU_Register()

This function checks the A, H, X, and CCR in a sequence. The A, H, and X registers are tested according to the diagram in Figure 9. The CCR register is tested according to the diagram in Figure 10.

Function prototype:

FS_RESULT FS_CPU_Register(void);

Test patterns for respective registers:

A, H, X, CCR : 0x55, 0xAA

Function inputs:

Void, no inputs are passed into the function.

Function output:

typedef unsigned long FS_RESULT;

When passes:

FS_CPU_PASS(0)

If some registers are corrupted, the function is in an endless loop. This state must be observed by another safety mechanism (for example, watchdog).

Function performance:

The information about function performance is in Core self-test library – source code version.

Calling restrictions:

This function cannot be interrupted.

8.2.2 FS_SP()

This function checks the stack pointer according to the diagram in Figure 10.

Function prototype:

FS_RESULT FS_SP(void);

Test pattern:

SP_main: 0x5556, 0xAAAB

Function inputs:

Void, no inputs are passed into the function.

Function output:

typedef unsigned long FS_RESULT;

When it passes:

FS_CPU_PASS(0)

If the SP is corrupted, the function is in an endless loop. This state must be observed by another safety mechanism (for example, watchdog).

Function performance:

The information about the function performance is in Core self-test library – source code version.

Calling restrictions:

This function cannot be interrupted.

Chapter 9
Stack test

This test routine is used to test the overflow and underflow conditions of the application stack. The testing of stuck-at faults in the memory area occupied by the stack can be covered by the variable memory test. The overflow or underflow of the stack can occur if the stack is incorrectly controlled, or by defining a very low stack area for the given application.

The principle of the test is to fill an area below and above the stack with a known pattern. These areas must be defined in the linker configuration file, like the stack. The initialization function then fills these areas with a pattern defined by the application developer. The pattern must have a value that does not appear elsewhere in the application. The test is performed after the reset and during the application runtime in the same way. The purpose is to check if the exact pattern is still written in these areas. If it is not, it is a sign that there is an incorrect stack behavior. If this occurs, then the fail return value from the test function must be processed as a safety error.

9.1 Stack test in compliance with IEC/UL standards

The stack test is an additional test. It is not directly specified in the IEC60730 annex H table.

9.2 Stack test implementation

The test function for the stack and the initialization function are placed in the *iec60730b_s08_stack.S* file and written as assembler functions. The header file with the return values and the function prototypes is *iec60730b_s08_stack.h*. The *iec60730b.h* and *iec60730b_core.h* files are included in the *iec60730b_s08_stack.S* file. Therefore, they must be also placed in the application. The example for the linker setup, process of initialization, and implementation is shown in the following sections.

9.2.1 Linker setup

The size and placement of the application stack is generally defined in the linker configuration file. Therefore, define the areas below and under the stack as well. There are several methods to achieve this and here is one of them:

```
SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in PLACEMENT below. */
...
STACK_TEST_1 = READ_WRITE 0x0379 TO 0x037B;
/* STACK AREA 0x37C to 0x3FA */
STACK_TEST_2 = READ_WRITE 0x03FC TO 0x3FF;
...
/* INTVECTS = READ_ONLY 0xFFB0 TO 0xFFFF; Reserved for Interrupt Vectors */
END

// _____________
// |_____________| --> STACK_TEST_1 ....ADR
// |_____________| ....ADR + 0x2
// | STACK TOP |
// | ...|
// | STACK AREA |
// | ... |
// | STACK BASE |
```

```
// |_____________| -- STACK_TEST_2 .. ADR2
// |_____________| ..ADR2 + 0x2
```

In the example, the size is set to 0x2. STACK_TEST_1 is the first address in the tested block above the stack. STACK_TEST_2 defines the first tested address under the stack. These areas are not included in the RAM region, so the compiler cannot reserve this place for variables or other parameters. However, you can put the variables directly to this address:

```
#define STACK_TEST_ADDR_1 0x379
#define STACK_TEST_ADDR_3 0x3FC
uint16_t ui16_stack_test_address_1[STACK_TEST_BLOCK_SIZE]@STACK_TEST_ADDR_1;
uint16_t ui16_stack_test_address_3[STACK_TEST_BLOCK_SIZE]@STACK_TEST_ADDR_3;
```

Now there are two array variables in the desired destination.

9.2.2 FS_STACK_Init

The purpose of the initialization is to fill the defined areas with a given pattern. Put the values from the linker configuration file into the variables. Then define the rest of the parameters needed for the initialization function.

Example of initialization:

```
#include "iec60730b.h"
#define STACK_TEST_ADDR_1 0x379
#define STACK_TEST_ADDR_3 0x3FC
#define STACK_TEST_BLOCK_SIZE 0x2
#define STACK_TEST_PATTERN 0x77
uint16_t ui16_stack_test_address_1[STACK_TEST_BLOCK_SIZE]@STACK_TEST_ADDR_1;
uint16_t ui16_stack_test_address_3[STACK_TEST_BLOCK_SIZE]@STACK_TEST_ADDR_3;
FS_Stack_Init(STACK_TEST_PATTERN, &ui16_stack_test_address_1, &ui16_stack_test_address_3,
STACK_TEST_BLOCK_SIZE);
```

Function prototype:

*void FS_Stack_Init(uint8_t stack_test_pattern, uint8_t *test_address_1, uint8_t *test_address_3, uint8_t block_size);*

Function inputs:

stack_test_pattern – The pattern to be written into the areas.

test_address_1 – The first address in the block under the stack area.

test_address_3 – The first address in the block above the stack area.

block_size – The size of the areas under and above the stack.

Function output:

Void. The function does not return a value.

Function performance:

The information about the function performance is in Core self-test library – source code version.

Calling restrictions:

None.

9.2.3 FS_STACK_Test

The testing procedure is the same after the reset and during runtime. This function checks if the areas are not rewritten with content different from the defined pattern. The inputs for the testing functions must be the same as those for the initialization function.

Function prototype:

*FS_RESULT FS_Stack_Test(uint8_t stack_test_pattern, uint8_t *p_test_address_1, uint8_t *p_test_address_3, uint8_t block_size);*

Function inputs:

stack_test_pattern – The pattern to be written into the areas.

p_test_address_1 – The pointer to the first test block (above the stack); see Linker setup.

p_test_address_3 – The pointer to the second test block (under the stack); see Linker setup.

block_size – The size of the areas under and above the stack.

Function output:

typedef unsigned long FS_RESULT;

Can have two values:

FS_STACK_FAIL (0x51)

FS_STACK_PASS (0)

Function performance:

The information about the function performance is in Core self-test library – source code version.

Calling restrictions:

None.

Chapter 10
Watchdog test

The watchdog test tests the watchdog timer functionality. The test checks whether the watchdog timer can cause a reset and whether the reset happens at the expected time. Before the start of the test, the watchdog must be configured for use in the respective application. The next step before the test is the setup of the independent device timer, which is used for the watchdog timeout comparison. The first function for watchdog testing is called after that. This function refreshes the watchdog timer, activates the device timer, and captures the device timer counter value during an endless loop. This function should be called only once after the Power-On Reset (POR). After the watchdog reset, the second function must be called. This function should be called after every reset, except for the POR. This function checks whether the captured device timer counter value corresponds to the expected watchdog timeout value. The next check is whether the number of watchdog resets does not exceed the limit value. You can choose what action must be made after an incorrect result. Due to safety requirements, you have limited options for choosing the clock source for the watchdog and for the device timer. The first condition is that the watchdog timer clock cannot be the same as the watchdog bus interface clock. Check the device reference manual for the watchdog timer clock source options. The second condition is that the watchdog timer clock cannot be the same as the device timer clock.

10.1 Watchdog test in compliance with IEC/UL standards

The watchdog test is not directly specified in the IEC60730 - annex H table, but it partially fulfils the safety requirements according to the IEC 60730-1, IEC 60335, UL 60730, and UL 1998 standards, as described in this table:

Table 12. Watchdog test in compliance with the standards

Test	Component	Fault / Error	Software / Hardware Class	Acceptable Measures
Watchdog test	3. Clock	Wrong frequency	B/R.1	Frequency monitoring
Watchdog test	8. Monitoring devices and comparators	Any output outside the static and dynamic functional specification	B/R.1	Tested monitoring

10.2 Watchdog test implementation

The test functions for the watchdog are placed in the *iec60730b _s08_wdg.c* file. The header file is *iec60730b_s08_wdg.h*. The *iec60730b.h* and *iec60730b_core.h* files must be placed in the application as well.

Allocate space in the RAM memory, which is not corrupted after the non-POR.

This memory is used for the user-defined variable of the *fs_wdog_test* type, which is a structure with five members. It is defined in the *iec60730b _s08_wdg.h* file.

The important condition for performing the watchdog test is to have the watchdog module and device timer confirmed before starting the test.

Ensure the handling of the functions. To identify the source of the reset, use the reset control module. By default, if an unwanted result is found by the “check” function, the program stays in an endless loop in the function. This causes the application to stay in the loop of watchdog resets. By entering zero as the fourth input value of the check function, the endless loop is not activated. In that case, ensure that the application is put into a safe state.

This is an example of the watchdog test implementation :

```
#include “iec60730b.h”
#define Watchdog_refresh __RESET_WATCHDOG()
#define ENDLESS_LOOP_ENABLE 0 /* set 1 or 0 */
```

```
#define WATCHDOG_RESETS_LIMIT 1000
#define WATCHDOG_TIMEOUT_VALUE 1024
#define WD_REF_TIMER_CLOCK_FREQUENCY 32768
#define WATCHDOG_CLOCK 1000
#define WD_TEST_TOLERANCE 50 /* % */
fs_wdog_test c_wdBackupAddress@WDOG_VARIABLE_ADDRESS;
uint8_t tmp;
/*Initialization of RTC - reference timer */
ICS_C1 |= ICS_C1_IRCLKEN_MASK; // Enable clock 32kHz
RTC_MODH = 0xFF; // overflow only on FULL
RTC_MODL = 0xFF; //
RTC_SC2 = RTC_SC2_RTCPS0_MASK | RTC_SC2_RTCLKS1_MASK; // Select Internal clock (~32kHz) , divide 1
RTC_SC1 = RTC_SC1_RTIF_MASK | RTC_SC1_RTIE_MASK; // interrupt cleared and enabled
/* Reset the timer */
tmp = RTC_SC2;
RTC_SC2 = 0U;
RTC_SC2 = tmp;

if(SYS_SRS != SYS_SRS_WCOP_MASK) /*Any NO WCOP reset*/
{
FS_watchdog_setup(&c_wdBackupAddress, (FS_RTC_Type *)RTC_BASE);
}
if(SYS_SRS == SYS_SRS_WCOP_MASK) /*If WCOP reset*/
{
FS_watchdog_check(counterLimitHigh, counterLimitLow, WATCHDOG_RESETS_LIMIT, ENDLESS_LOOP_ENABLE,&c_wdBackupAddress);
}
```

10.2.1 FS_WDOG_Setup()

This function clears the reset counter, which is a member of the *fs_wdog_test* structure. It refreshes the watchdog to start counting from zero. It starts the reference timer, which must be configured before the function call. Within the waiting endless loop, the value from the reference timer is periodically stored in the reserved area in the RAM.

Function prototype:

void FS_watchdog_setup(fs_wdog_test pWatchdogBackup, FS_RTC_Type* rtc_base)*

Function inputs:

pWatchdogBackup - The pointer to a structure of the *fs_wdog_test* type, defined in the header file.

rtc_bace - The pointer to the base address of the RTC counter, which is used as a reference.

Function output:

Void. The function does not return a value.

Function performance:

The function duration depends on the watchdog timeout setings, because the function waits during the watchdog reset.

Calling restrictions:

The watchdog timer and the RTC timer must be configured correctly. The variable of the *fs_wdog_test* type must be declared and placed into a reliable place. Interrupts should be disabled.

10.2.2 FS_WDOG_Check()

This function compares the captured value of the RTC counter with pre-calculated limit values and checks whether the watchdog reset counter overflows. If the function is called after a non-watchdog reset, the wd_test_uncomplete_flag is set. With the endless_loop_enable parameter, the endless loop within the function is enabled or disabled (by setting it to values "1" or "0"). If enabled, the function ends up in an endless loop under these conditions:

- Entering after the non-watchdog or non–POR resets.
- The counter from the watchdog test does not fit into the limit values.
- The watchdog resets exceed the defined limit value.

Function prototype:

void FS_watchdog_check(uint32_t limit_high, uint32_t limit_low, uint16_t resets_limit, uint8_t endless_loop_enable, fs_wdog_test pWatchdogBackup)*

Function inputs:

limit_high - The pre-calculated upper limit value of the RTC counter in the watchdog test.

limit_low - The pre-calculated lower limit value of the RTC counter in the watchdog test.

resets_limit - The defined limit of the watchdog resets.

endless_loop_enable - If this is 1, the endless loop is activated in the function.

pWatchdogBackup - The pointer to a structure of the *fs_wdog_test* type, defined in the header file.

Function output:

Void. The function does not return a value.

Function performance:

The information about the function performance is in Core self-test library – source code version.

Calling restrictions:

The respective setup function must be executed first.

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: 01/2020

Document identifier: IEC60730BHCS08LUG40

arm