Document identifier: IEC60730B56800EXLUG40

IEC60730_B_56800EX_Library_UG_v4_0

IEC60730B Library User's Guide

Contents

Chapter 1
Core self-test library

The purpose of the core self-test library is to provide 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 the following 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, implementation, test, and validation of the corresponding tests is comprehensively described in independent sections for each test.

The library supports the 56F83xxx family based on the 56800EX core.

The core self-test library has two versions: source code version and 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.

The following is the object file:

- CodeWarrior: *IEC60730B_DSC_Class_B_CW_v4_0.lib*

1.2 Core self-test library – source-code version

The library name is IEC60730B_DSC_Class_B_CW_v4_0. The main header files are *iec60730b.h* and *iec60730b_core.h*.

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

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 depend library header file	-	-	-
iec60730b_dsc_aio.c	Analog I/O test	FS_AIO_InputInit_CYCLIC()	58[1]	2.13[1]
	Analog I/O test	FS_AIO_InputTrigger()	28[1]	1.13[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_InputSet_CYCLIC()	172[1]	4.63[1]
	Analog I/O test	FS_AIO_InputCheck_CYCLIC()	280[1]	4.38[1]
iec60730b_dsc_clock.c	Clock test	FS_CLK_QTIMER_Check()	86[1]	2.88[1]
iec60730b_dsc_dio.c	Digital I/O test	FS_DIO_Output()	218[1]	41.38[1]
iec60730b_dsc_dio_ext.c	Extended Digital I/O test	FS_DIO_InputExt()	372[1]	11.25[1]
	Extended Digital I/O test	FS_DIO_ShortToSupplySet()	200[1]	5.88[1]
	Extended Digital I/O test	FS_DIO_ShortToAdjSet()	314[1]	8.75[1]
iec60730b_dsc_flash.c	Invariable memory test (Flash)	FS_flash_HW16_program()	52[1]	See the function dedicated chapter
	Invariable memory test (Flash)	FS_flash_HW16_data()	52[1]	See the function dedicated chapter
	Invariable memory test (Flash)	FS_flash_SW16_program()	128[1]	See the function dedicated chapter
	Invariable memory test (Flash)	FS_flash_SW16_data()	128[1]	See the function dedicated chapter
iec60730b_dsc_pc.c	Program Counter test	FS_PC_Test()	36[1]	See the function dedicated chapter
	Program Counter test	FS_PC_subroutine()	14[1]	-
iec60730b_dsc_pc_object1.c	Program Counter test	FS_PC_object_1()	6[1]	-
iec60730b_dsc_pc_object2.c	Program Counter test	FS_PC_object_2()	6[1]	-
iec60730b_dsc_ram.c	Variable memory test (RAM)	FS_RAM_AfterReset()	92[1]	See the function dedicated chapter
	Variable memory test (RAM)	FS_RAM_Runtime()	118[1]	See the function

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]
				dedicated chapter
	Variable memory test (RAM)	FS_RAM_CopyMemory()	12[1]	-
	Variable memory test (RAM)	FS_RAM_SegmentMarchC()	112[1]	-
	Variable memory test (RAM)	FS_RAM_SegmentMarchX()	88[1]	-
iec60730b_s08_reg.c	Register test	FS_CPU_DataRegisters()	434[1]	8.63[1]
	Register test	FS_CPU_PointerRegisters()	186[1]	7.88[1]
	Register test	FS_CPU_ShadowRegistersE()	208[1]	4.38[1]
	Register test	FS_CPU_ShadowRegistersEX()	194[1]	4.25[1]
	Register test	FS_CPU_StatusRegister()	192[1]	4.13[1]
	Register test	FS_CPU_StackPointer()	52[1]	1.63[1]
	Register test	FS_CPU_LoopRegisters()	558[1]	12.37[1]
iec60730b_s08_Stack.c	Stack test	FS_StackInit()	18[1]	1.88[1]
	Stack test	FS_StackTest()	38[1]	3.50[1]
iec60730b_s08_wdg.c	Watchdog test	FS_watchdog_setup()	52[1]	See the function dedicated chapter
	Watchdog test	FS_watchdog_check()	164[1]	4.50[1]

1.2.1 56F83xxx dedicated functions

This library is dedicated for the 56F83xxx family. Due to this, all functions mentioned in the table above are dedicated for all 56F83xxx DSCs.

1.3 Functions performance measurement

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

1. The function parameter was measured in the CodeWarrior 11.1. IDE on MC56F83789 with the clock frequency of 50 MHz.

Chapter 2
Analog input/output 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 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 it is equal to the FAIL return, then a jump into 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 checks the analog input interface and the defined number of reference values.

The block diagram for the analog IO test is shown in the following figure:

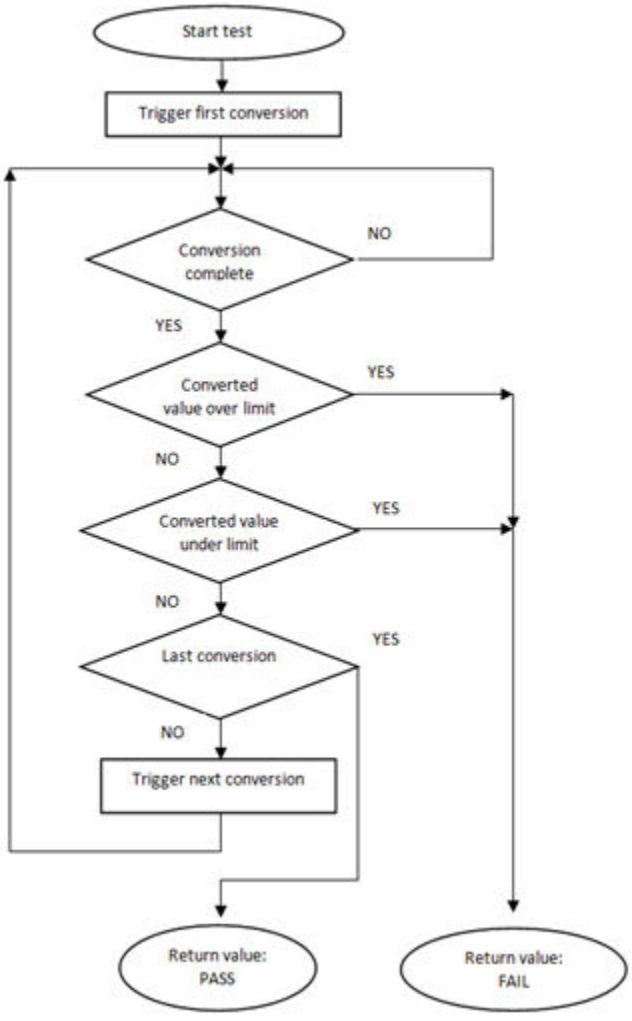

Figure 1. Block diagram for analog input test

2.1 Analog input/output test in compliance with IEC/UL standards

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

Table 2. Analog 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 (7.2 – A/D conversion)	Abnormal operation	B/R.1	Plausibility check

2.2 Analog input/output test implementation

The test functions for the analog IO test are placed in the *iec60730b_dsc_aio.c* file. The functions are written in the C language. The prototypes and definitions are in the *iec60730b_dsc_aio.h* file. The library common header file *iec60730b.h* must be included in the project as well.

The following functions are called to test the analog input:

- *FS_AIO_InputInit_CYCLIC ()*
- *FS_AIO_InputTrigger()*
- *FS_AIO_InputSet_CYCLIC()*
- *FS_AIO_InputCheck_CYCLIC()*

The principle of the analog input test is based on a conversion of analog inputs with known voltage values and the test checks whether the converted values fit into the limits defined by you. Normally, it should be about 10 % around the desired reference values. The AIO test implementation is as follows:

```
/**********define parameters**********/
#define CHANNELS_CNT 3
#define CHANNELS_INIT {4,5,7} /* Channels to scan*/
#define SAMPLES_INIT {0,1,1} /* SAMPLES from CLISTx registers */
#define CHANNELS_LIMITS \
{\
{3900, 4100},\
{ 0, 30},\
{ 900, 1200}\
}
static fs_aio_test_t aio_Str;
const fs_aio_limits_t adc_limits[CHANNELS_CNT] = CHANNELS_LIMITS;
const uint8_t ADC_inputs[CHANNELS_CNT] = CHANNELS_INIT;
const uint8_t ADC_samples[CHANNELS_CNT] = SAMPLES_INIT;

/**********initialization of hardware**********/
ioctl(PMC, PMC_INIT, NULL); /* enable internal reference for ANA 7 channel */
ioctl(GPIO_A, GPIO_INIT, NULL); /* set GPIOA4. GPIOA5 as ANA4, ANA5 inputs */
```

```
ioctl(ADC, ADC_INIT, NULL); /* initialisation of ADC module */

/**********initialization of test**************/
FS_AIO_InputInit_CYCLIC(&aio_Str, (fs_aio_limits_t*)adc_limits, (uint8_t *)ADC_inputs,
(uint8_t*)ADC_samples, CHANNELS_CNT);
FS_AIO_InputTrigger(&aio_Str);

/***************test loop********************/
for (i=0;i<6;i++)
{
test_result = FS_AIO_InputCheck_CYCLIC(&aio_Str, (unsigned long *)((FS_ADC_CYCLIC_Type*) 0xE500));
switch(test_result)
{
case FS_AIO_START:
FS_AIO_InputSet_CYCLIC(&aio_Str, (unsigned long *)((FS_ADC_CYCLIC_Type*) 0xE500));
ioctl(ADC, ADC_START, ADC_CONVERTER_0);
break;
case FS_AIO_FAIL:
safetyErrors |= AIO_TEST_ERROR;
SafetyErrorHandling();
FS_AIO_InputTrigger(&aio_Str);
break;
case FS_AIO_INIT:
FS_AIO_InputTrigger(&aio_Str);
break;
case FS_AIO_PASS:
FS_AIO_InputTrigger(&aio_Str);
break;
default:
break;
}
}
```

2.2.1 FS_AIO_InputTrigger()

This function sets up 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_CYCLIC()

This function initializes one instance of the analog input test.

Function prototype:

*void FS_AIO_InputInit_CYCLIC(fs_aio_test_t *pObj, fs_aio_limits_t *pLimits, uint8_t *pInputs, uint8_t *pSamples, 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.

pSamples - The pointer to the array of sample numbers used in the test.

cntMax - The size of the input and the limits 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_CYCLIC()

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_CYCLIC(fs_aio_test_t *pObj, uint32_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 can have the following values:

- FS_AIO_PASS (0x00000000)
- FS_AIO_FAIL (0x00000701)
- FS_AIO_PROGRESS (0x00000702)
- FS_AIO_INIT (0x00000704)

Function performance:

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

2.2.4 FS_AIO_InputCheck_CYCLIC()

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

Function prototype:

*FS_RESULT FS_AIO_InputCheck_CYCLIC(fs_aio_test_t *pObj, uint32_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 the following values:

- FS_AIO_PASS (0x00000000)
- FS_AIO_FAIL (0x00000701)
- FS_AIO_PROGRESS (0x00000702)
- FS_AIO_START (0x00000703)
- FS_AIO_INIT(0x00000704)

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 frequency fault. The clock test is performed during application runtime.

The identification of a safety error is ensured by the specific FAIL return in case of a clock fault. Assess the return value of the test function and if it is equal to the FAIL return, then a jump into 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 is based on a 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 the QTIMER which is fed by two independent clocks. The primary clock drives the counter. The secondary clock triggers a capture window on every rising edge of the signal. The capture window stores the current value of counter into the timer channel capture register and reloads the counter. The value in the capture register is then compared to the expected precalculated limit values by the check function. The check function is called periodically, at an appropriate time within an application. Its calling can be asynchronous to the QTIMER capture window.

The block diagram of test is shown in the following figure:

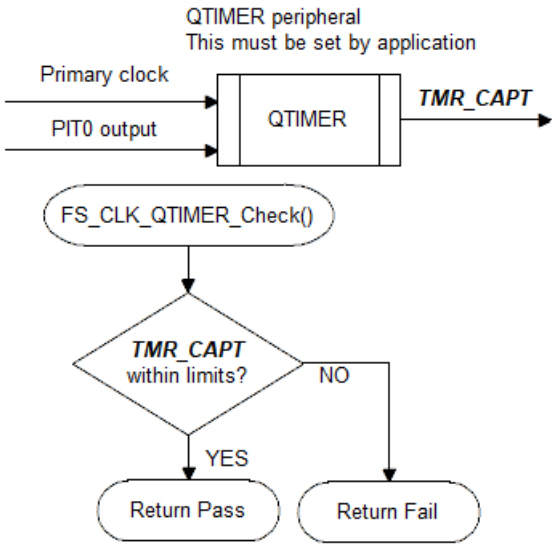

Figure 2. Block diagram for clock test

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 the following 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

The test functions for the clock test are placed in the *iec60730b_dsc_clock.c* file and written as C functions. The header file with the function prototypes is *iec60730b_dsc_clock.h*. The common library header files are *iec60730b.h* and *iec60730b_core.h*.

The safety library contains only one function to test the clock:

- FS_CLK_QTIMER_Check()

The other steps that are required for the clock test must be done within the application code.

Configure the QTIMER, choose an appropriate periodical event, and calculate the limit values. A structure of the fs_clock_test_t type must be defined. The type is defined in the *iec60730b_dsc_clock.h* file.

The following features must be set in the QTIMER:

- One of the independent clocks (the primary clock) goes to the counter.
- The second clock triggers the capture window. This window stores the value from the counter and reloads it.

The other settings can be customized to fit the respective application.

In the example code shown below, the following approach is used:

The PIT0 timer is clocked by a 200-KHz oscillator and the roll-over time is set to 10 us. The PIT0 output is routed to the QTIMER secondary input via the XBAR module. In the roll-over time period, the SYNC_OUT signal triggers the capture window. At this moment, the counter value is stored to the capture register and the counter reloads. The primary source of the QTIMER is the 50-MHz IPB clock.

The example of the test implementation is as follows:

```
#include "iec60730b.h"

/******************appconfig.h*****************/
#define SIM_CLKOSR_INIT 0xD020U
#define PMC_CONTROL_INIT 0x7080U
#define SIM_GPSAH_INIT 0x00C0U
#define SIM_GPSCL_INIT 0x0480U
#define SIM_GPSGL_INIT 0x0004U
#define SIM_PCE0_INIT 0xC05FU
#define SIM_PCE1_INIT 0x0402U
#define SIM_PCE2_INIT 0x00ACU
#define SIM_PCE3_INIT 0x0000U
#define SIM_IPS0_INIT 0x4300U
#define SIM_PWM_SEL_INIT 0x0200U
#define QT_A0_CTRL_INIT 0x3000U
#define QT_A0_SCR_INIT 0x0040U
#define QT_A0_COMSCR_INIT 0x0800U
#define PIT_0_CTRL_INIT 0x0301U
#define PIT_0_MOD_INIT 0x0001U
#define XBAR_A_SEL19_INIT 0x2C2CU
/*********************************************/
```

```
#define QTIMER_PRIMARY_CLK 50000 /*Frequency in kHz*/
#define SECONDARY_CAPTURED_CLK 100 /*Frequency in kHz*/
#define CLOCK_TEST_TOLERANCE 20 /* % */
fs_clock_test_t g_sClock;

g_sClock.qtimer_channel_base = (FS_CHAN_QTIMER_Type *)TMRA_BASE;
g_sClock.clockTestExpected = (QTIMER_PRIMARY_CLK/ SECONDARY_CAPTURED_CLK);
g_sClock.clockTestTolerance = (g_sClock.clockTestExpected * (uint32_t)CLOCK_TEST_TOLERANCE) /
(uint32_t)100 ;
g_sClock.clockTestLimitHigh = g_sClock.clockTestExpected + g_sClock.clockTestTolerance;
g_sClock.clockTestLimitLow = g_sClock.clockTestExpected - g_sClock.clockTestTolerance;

ioctl(SYS, SYS_INIT, NULL);
ioctl(SIM, SYS_INIT, SIM_PCE2_INIT);
ioctl(XBAR_A, XBAR_A_INIT, NULL); /* QTA0_input PIT0_SYNC_OUT, QTA1_input PIT0_SYNC_OUT */
ioctl(PIT_0, PIT_INIT, NULL); /* RC oscilator 200kHz, roll over time 10us */
ioctl(QTIMER_A0, QT_INIT, NULL); /* 50 MHz */

while(1)
{
.
.
CLOCK_test_result = FS_CLK_QTIMER_Check(g_sClock);
if (CLOCK_test_result == FS_CLK_FAIL)
{
safetyErrors |= CLOCK_TEST_ERROR;
SafetyErrorHandling(g_sSafetyCommon);
}
.
.
}
```

3.2.1 FS_CLK_QTIMER_Check()

This function handles the clock test. It evaluates the captured value in the TMRx_CAPTn register with precalculated limits. Until the first capture, the function returns FS_CLK_PROGRESS.

Function prototype:

*FS_RESULT FS_CLK_QTIMER_Check(fs_clock_test_t *pClockTest);*

Function inputs:

pClockTest - The pointer to the structure of an fs_clock_test type, defined in the test header file.

Function output:

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

- FS_CLK_PASS (0x000)
- FS_CLK_FAIL (0x601)
- FS_CLK_PROGRESS (0x602)

Function performance:

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

Chapter 4
Digital input/output test

The Digital Input/Output (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 DIO error. Assess the return value of the test function and if it is equal to the FAIL return, a move into 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 safe state.

The DIO test functions check the digital input and digital output functionality and the short circuit conditions between the tested pin and the supply voltage, GND, or optional adjacent pin. The execution of DIO tests 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 while the application runs. When performing the digital output test, ensure that there is enough time between the test arrangement and the reading of a result.

4.1 Digital input/output 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 the following 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 input/output test implementation

Test functions for the digital IO test are placed in the *iec60730b_dsc_dio.c* and *iec60730b_dsc_dio_ext.c* files. The header files with the function prototypes are *iec60730b_dsc_dio.h* and *iec60730b_dsc_dio_ext.h*. The *iec60730b.h* and *iec60730b_core.h* are the common header files that must be included in the project as well.

The digital input/output tests can be executed using the following functions:

- FS_DIO_Output()
- FS_DIO_InputExt()
- FS_DIO_ShortToSupplySet()
- FS_DIO_ShortToAdjSet()

The pointer to the variable of an fs_dio_test_t structure type is a parameter of each function. The structure is defined in the *iec60730b_dio.h* file.

```
typedef struct
{
uint16_t PUR; /* GPIO Pull Resistor Enable Register */
uint16_t PUS; /* GPIO Pull Resistor Type Select */
uint16_t DDR; /* GPIO Data Direction Register */
uint16_t DR; /* GPIO Data Register */
```

```
} fs_dio_backup_t;

/* Safety DIO test item */
typedef struct
{
uint16_t gpio; /* Pointer to GPIO module */
uint8_t pinNum;
uint8_t pinDir;
uint8_t pinMux;
fs_dio_backup_t sTestedPinBackup;
} fs_dio_test_t;
```

This variable/variables must be initialized before the call of a test function. Example of initialization and test execution is shown below.

```
fs_dio_test_t test_item_0 = /* P1_14 */
{
/*.gpio =*/ FGPIOG_BASE, /* Pointer to GPIO module */
/*.pinNum =*/ 1, /* Pin Number */
/*.pinDir =*/ 0, /* In or Out 0 and 1*/
/*.pinMux =*/ 0, /* */
};

fs_dio_test_t test_item_1 = /* P1_20 */
{
/*.gpio = */FGPIOG_BASE, /* Pointer to GPIO module */
/*.pinNum =*/ 2,
/*.pinDir =*/ 0,
/*.pinMux =*/ 1,
};

/* NULL terminated array of pointers to dio_test_t items for safety DIO test */
fs_dio_test_t *test_items[3]={ &test_item_0, &test_item_1, NULL };
```

4.2.1 FS_DIO_Output()

This function tests the digital output functionality of the pin. The principle of the test is to set up and read both logical values on the tested pin. Enter a suitable delay parameter. It must ensure a time interval that is long enough for the device to reach the desired logical value on the pin. A very low delay parameter causes the fail return value of the function.

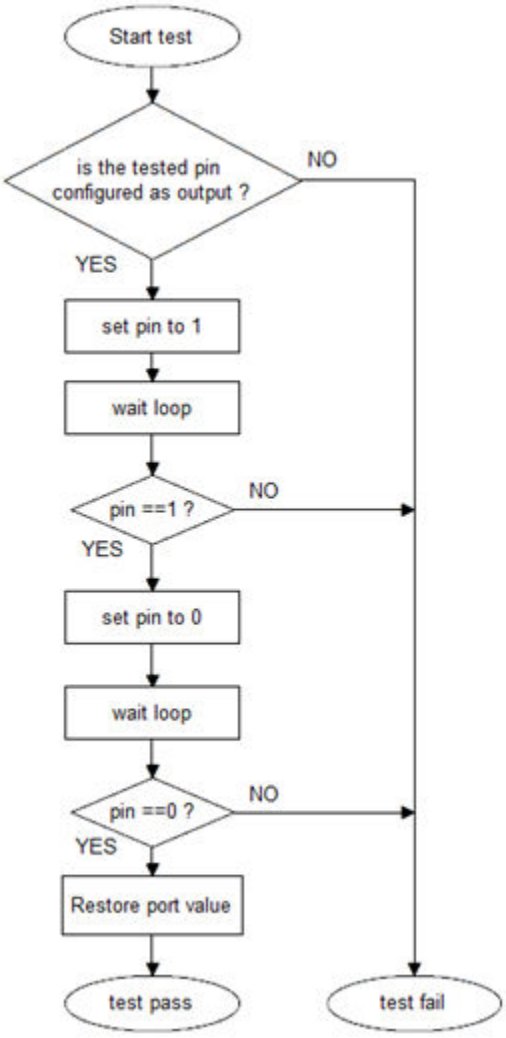

Figure 3. Block diagram for digital output test

Function prototype:

*FS_RESULT FS_DIO_Output(fs_dio_test_t *pTestedPin, uint16_t delay);*

Function inputs:

pTestedPin – The pointer to a variable of the fs_dio_test_t type. Specifies the tested pin. See Digital input/output test implementation.

delay - The delay needed to recognize the value change on the tested pin.

Function output:

The function returns a value of the unsigned short data type and has either of the following values:

- FS_DIO_FAIL (0x00000801)
- FS_DIO_PASS (0)

Example of function call:

PortInit((GPIO_Type)test_item_0.gpio, PIN_DIRECTION_OUT, test_item_0.pinNum, PIN_PULL_UP);*

DIO_out_result = (&test_item_0, 50);

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 digital output. An appropriate delay must be defined for proper functionality.

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 tested pin must be configured as a GPIO input pin and the adjacent pin must be configured as a GPIO output pin. The function block diagram is shown in the following figure. Similarly to the short-to-supply 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 for the input test function.

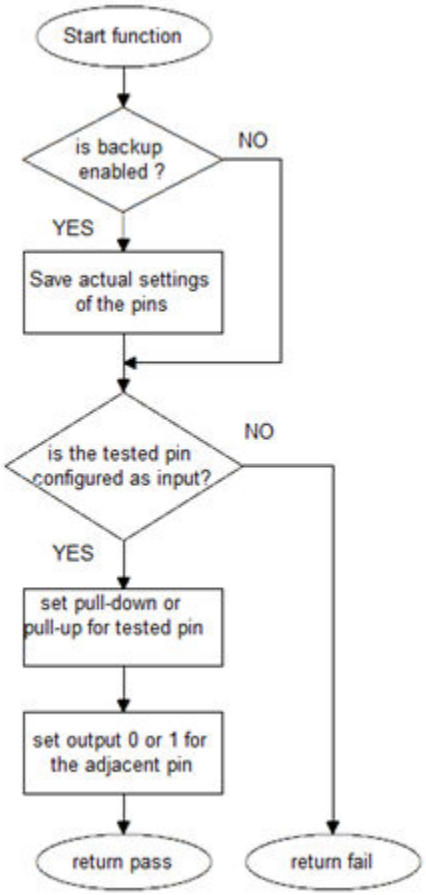

Figure 4. Block diagram of FS_DIO_ShortToAdjSet() function

Function prototype:

*FS_RESULT FS_DIO_ShortToAdjSet(fs_dio_test_t *pTestedPin, fs_dio_test_t *pAdjPin, bool_t testedPinValue, bool_t backupEnable);*

Function inputs:

pTestedPin - The pointer to a variable of the fs_dio_test_t type. It specifies the tested pin.

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

testedPinValue - The value that is set on the tested pin (logical 0 or logical 1).

backupEnable - The flag. If it is non-zero, the backup functionality is enable/active.

Function output:

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

• FS_DIO_FAIL (0x00000801)

• FS_DIO_PASS (0)

Example of function call:

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

```
PortInit((GPIO_Type*)test_item_0.gpio, PIN_DIRECTION_IN, test_item_0.pinNum, PIN_PULL_UP);
PortInit((GPIO_Type*)test_item_1.gpio, PIN_DIRECTION_OUT, test_item_1.pinNum, PIN_PULL_UP);

dioShortSetResult = FS_DIO_ShortToAdjSet(&test_item_0, &test_item_1, LOGICAL_ONE, BACKUP_EN);

/* if needed, place some delay between the two functions */

dioInputResult = FS_DIO_InputExt(&test_item_0, &test_item_1, LOGICAL_ONE, BACKUP_EN);
```

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 the GPIO input and the adjacent pin must be configurted as the GPIO output before calling the function. If the backup functionality is enabled, the function sets the directions for both pins. If not, configure the directions (tested pin as input, adjacent pin as output). After the end of the function, the application cannot manipulate neither the tested nor the adjacent pin, until the FS_DIO_InputExt function is called for these pins.

4.2.3 FS_DIO_ShortToSupplySet()

This function is the first part of the short-to-supply test. It can be used for short circuit testing between the tested pin and the hardware supply voltage (VCC, VDD) or between the tested pin and 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_InputExt() function which is described in the respective section. The tested pin must be configured as a GPIO input pin. The main purpose of the FS_ShortToSupplySet() function is to set the pull-up or pull-down resistor connection on the tested pin. It also ensures the test whether the pin is correctly configured and makes a backup of its settings (if needed).

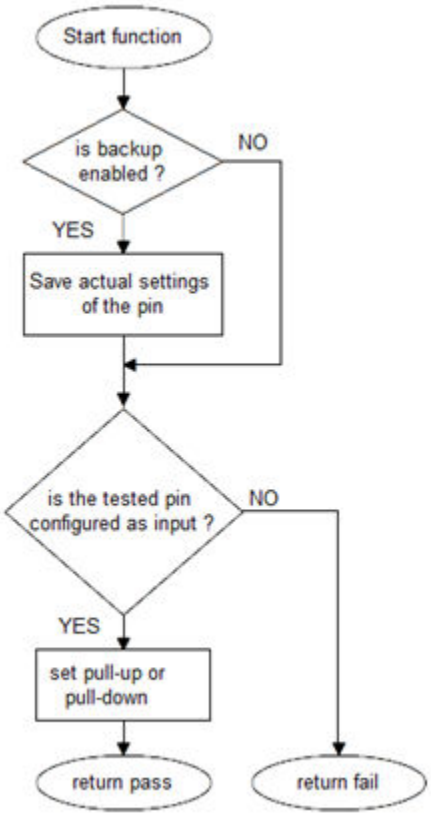

Figure 5. Block diagram of FS_DIO_ShortToSupplySet function

Function prototype:

*FS_RESULT FS_DIO_ShortToSupplySet(fs_dio_test_t *pTestedPin, bool_t shortToVoltage, bool_t backupEnable);*

Function inputs:

pTestedPin - The pointer to a variable of the fs_dio_test_t type. It specifies the tested pin.

shortToVoltage - This specifies whether the pin is tested for the short against GND or VDD. For GND, enter 1 or non-zero. For VDD, enter 0.

BackupEnable - The flag. If it is non-zero, the backup functionality is enable/active.

Function output:

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

• FS_DIO_FAIL (0x00000801)

• FS_DIO_PASS (0)

Example of function call:

The following is the code example of the short-to-GND test. The implementation difference between the short-to-GND and short-to-VDD tests is only in one parameter. For the short-to-GND test, the parameter must have a non-zero value and the other way round.

```
PortInit((GPIO_Type*)test_item_0.gpio, PIN_DIRECTION_IN, test_item_0.pinNum, PIN_PULL_UP);
dioShortSetResult = FS_DIO_ShortToSupplySet(&test_item_0, 1, BACKUP_ENABLE);
/* if needed, place some delay between the two functions */
```

```
dioInputResult = FS_DIO_InputExt(&test_item_0, &test_item_0, 1, BACKUP_ENABLE);
```

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. If the backup functionality is enabled, the function sets the input direction for the tested pin. If not, configure the input direction. After the end of the function, the application cannot manipulate the tested pin until the FS_DIO_InputExt function is called for the tested pin.

4.2.4 FS_DIO_InputExt()

This function a get function for the "short-to" tests, but it can be also used independently (standalone) as a test of an input pin with a known value. The function is applied to the pin that is already configured as the GPIO input and you know what logical level is expected at the time of the test. The logical level can either result from the actual configuration in the application or it can be initialized for the test (if possible). The block diagram is shown in the following figure. If the function is used as a standalone test, the pAdjPin parameter has no effect. Entering the same inputs as for the tested pin is recommended.

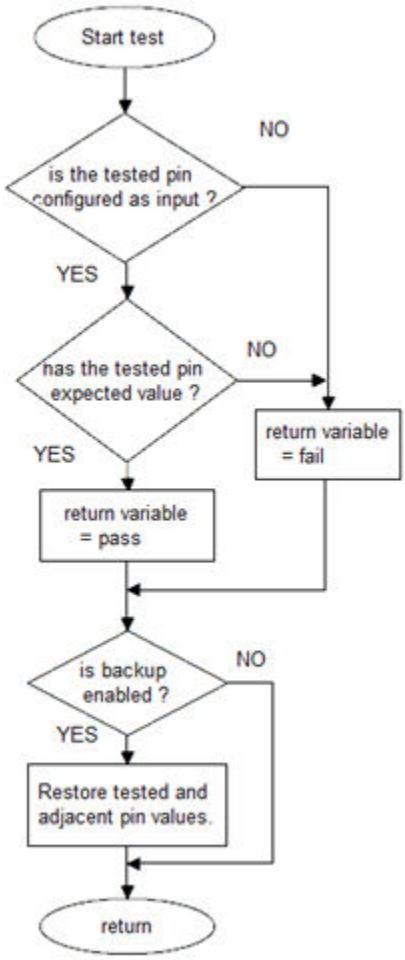

Figure 6. Extended digital input test

Function prototype:

*FS_RESULT FS_DIO_InputExt(fs_dio_test_t *pTestedPin, fs_dio_test_t *pAdjPin, bool_t testedPinValue, bool_t backupEnable);*

Function inputs:

pTestedPin - The pointer to a variable of the fs_dio_test_t type. It specifies the tested pin.

pAdjPin - The pointer to a variable of the fs_dio_test_t type. It specifies the adjacent pin.

testedPinValue - The expected value of the tested input pin (0 or 1). Adjust this parameter correctly.

backupEnable - The flag. If it is non-zero, the backup functionality is enabled.

Function output:

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

- FS_DIO_FAIL (0x00000801)
- FS_DIO_PASS (0)

Example of function call:

See the short-to-supply or short-to-adjacent-pin examples.

Function performance:

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

Calling restrictions:

The function cannot be used on the MKE0x devices. The tested pin must be configured as the GPIO input before the function call. Even if no adjacent pin is involved in the test, the AdjacentPin parameter must be specified. It is recommended is to enter the same input as for the TestedPin parameter.

Chapter 5
Invariable memory test

The invariable memory on the MC56F83xxx DSCs 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 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 that the checksum is calculated for. In the runtime and after reset, the same algorithm must be implemented in the application to calculate the checksum. The results are compared. If they are not equal, a safety error state occurs.

The algorithm that calculates the checksum parameter (signature) is set as the 16-bit CRC polynomial (0x1021) to generate a CRC code for error detection.

The DSCs feature 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 has better performance levels. The software version of the test can be used as an alternative when the CRC hardware module is not used for any reasons.

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 the following 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 are placed in the *iec60730b_dsc_flash.c* file. The functions are written in the assembler language. The prototypes and definitions are in the *iec60730b_dsc_flash.h* file. The library common header files are *iec60730b.h* and *iec60730b_core.h*. These files must be included in the project as well. There are four functions with identical functionality in the *iec60730b_dsc_flash.c* file:

- FS_flash_HW16_program()
- FS_flash_HW16_data()
- FS_flash_SW16_program()
- FS_flash_SW16_data()

The functions with “HW” in their names use the hardware CRC module. The functions marked as "SW" do calculations without the hardware module. Use the functions with the “program” suffix when checking the program memory. Use the functions with the “data” suffix to check the data memory.

5.2.1 Computing of CRC value in linking phase of application

In the final application, the checksum of a given memory block must be calculated before it is written into the flash memory. The rule is that the calculated checksum is stored at a known address in flash which is apart from the tested block.

5.2.2 Test performed once after MCU reset

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

```
#include "iec60730b.h"

extern UInt16 _checksum;

FLASH_test_result = FS_flash_HW16_program((UInt16 *) startAddress, (UInt16)size, (UInt16 *)CRC_BASE, (UInt16)startSeed);

if(_checksum != FLASH_test_result) SafetyError();
```

Where:

- 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.
- startSeed - The start condition seed (usually 0xFFFF).

5.2.3 Runtime test

In application runtime and with limited time for execution, the CRC is computed in a sequence. It means that the input parameters have different meaning when called after the reset. The start address is replaced with the actual address as the first parameter in the after-reset case. The implementation can be as follows:

```
psFlashCrc->partCrc = FS_flash_HW16_program((UInt16*)psFlashCrc->actualAddress, (UInt16)psFlashConfig->blockSize, (UInt16*)CRC_BASE, (UInt16)psFlashCrc->partCrc);

if(FS_FLASH_FAIL == SafetyFlashTestHandling( psFlashCrc, psFlashConfig))

{ psSafetyCommon->safetyErrors |= FLASH_TEST_ERROR; SafetyErrorHandling(psSafetyCommon); }
```

Where:

- psFlashCrc->partCrc - The particular CRC result and the seed parameter for the next iteration.
- psFlashCrc->actualAddress - The actual address of the memory block to be tested.
- CRC_BASE - The base address of the CRC module.
- psFlashConfig->blockSize - The size of the memory block to be tested.
- psFlashCrc->partCrc - The CRC value computed in the previous call.

Carry out the handling of the function. 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 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_data()

This function calculates the 16-bit CRC polynomial (0x1021) of the data stored in the data memory using the CRC module.

Function prototype:

*unsigned short FS_flash_HW16_data(unsigned long *startAddress, unsigned long size, unsigned long *moduleAddress, unsigned short crcVal);*

Function inputs:

*startAddress - The pointer to 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 calculation (for the first iteration it is 0; for the next iterations it is the result from the previous function call)

Function output:

unnt16_t - The checksum of the block of memory defined by the input parameters.

Function performance:

The function size is 52 bytes.[1]

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

Table 6. Duration of FS_flash_HW16_data() in dependence of tested block size

Block size (Bytes)	Execution time (approximately)
0x20	6.88 µs
0x40	11.75 µs
0xA0	27.00 µs

Calling restrictions:

The hardware CRC module must be enabled and turned on before the function call. The function cannot be interrupted by a function that changes the content or setup of the hardware CRC module. The addressed memory always stands for the data memory space.

5.2.5 FS_flash_HW16_program()

This function calculates a 16-bit CRC polynomial (0x1021) of the data stored in the program memory using the CRC module.

Function prototype:

*unsigned short FS_flash_HW16_program(unsigned long *startAddress, unsigned long size, unsigned long *moduleAddress, unsigned short crcVal);*

Function inputs:

*startAddress - The pointer to the first address of 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 (for the first iteration it is 0; for the next iterations it is the result from the previous function call).

Function output:

The checksum of the block of memory defined by the input parameters.

Function performance:

The function size is 52 bytes.[1]

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

Table 7. Duration of FS_flash_HW16_program() in dependence of tested block size

Block size (Bytes)	Execution time (approximately)
0x20	8.50 µs
0x40	14.38 µs
0xA0	34.00 µs

Calling restrictions:

The hardware CRC module must be enabled and turned on before the function call. The function cannot be interrupted by a function that changes the content or setup of the hardware CRC module. The addressed memory always stands for the program memory space.

5.2.6 FS_flash_SW16_data()

This function calculates a 16-bit CRC polynomial (0x1021) of the data stored in the data memory without the CRC module.

Function prototype:

*unsigned short FS_flash_SW16_data(unsigned long *startAddress, unsigned long size, unsigned long *moduleAddress, unsigned short crcVal);*

Function inputs:

*startAddress - The pointer to 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. This parameter has no impact.

crcVal - The input seed value for the calculation (for the first iteration it is 0; for the next iterations it is the result from the previous function call)

Function output:

uint16_t - The checksum of the block of memory defined by the input parameters.

Function performance:

The function size is 128 bytes.[1]

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

Table 8. Duration of FS_flash_SW16_data() in dependence of tested block size

Block size (Bytes)	Execution time (approximately)
0x20	115.80 µs
0x40	198.00 µs
0xA0	538.00 µs

Calling restrictions:

The addressed memory always stands for the data memory space.

5.2.7 FS_flash_SW16_program()

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

Function prototype:

*unsigned short FS_flash_SW16_program(unsigned long *startAddress, unsigned long size, unsigned long *moduleAddress, unsigned short crcVal);*

Function inputs:

*startAddress - The pointer to 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. This parameter has no impact

crcVal - The input seed value for the calculation (for the first iteration it is 0; for the next iterations it is the result from the previous function call).

Function output:

unnt16_t - The checksum of the block of memory defined by the input parameters.

Function performance:

The function size is 128 B.[1]

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

Table 9. Duration of FS_flash_SW16_program() in dependence of tested block size

Block size (Bytes)	Execution time (approximately)
0x20	120.10 µs
0x40	239.00 µs
0xA0	588.60 µs

Calling restrictions:

The addressed memory always stands for the program memory space.

Chapter 6
CPU program counter test

This test tests the functionality of the program counter register. This test is targeted for the "stuck-at" fault condition. The test can be performed once after the MCU reset and also during runtime.

Each value written to the program counter register is supposed to be an address at which the processor tries to jump immediately. Therefore, pay much attention to the test patterns that are used to test the program counter.

The principle of the test implemented on DSCs is as follows.

The test consists of four functions. The first function is a handling function. One of the two independent “object functions” and another subroutine function are called from it. At the beginning of the handling function is a flag, which is addressed when an input parameter is set. After successful execution of the test, this flag is cleared.

There are two independent functions, each of them appears in a separate source file (a separately linkable object). That is the key principle of the whole test. Place both object functions in appropriate and possible addresses in the program memory. By default, this setup is done in the linker. These addresses are test patterns and should test as much of the program counter as possible.

It is expected that a complete iteration of the test consists of two handling function executions. Both object functions are used as parameters of the handling function.

The advantage of the previously mentioned flag is that it can be useful in the case of a program counter test failure and a subsequent diagnostic. If the failure of the program counter register is a "stuck-at" failure of one bit, everything depends on the position of that bit in the register. If the bit is a part of the address used before the test itself, the application is damaged. Otherwise, if the stuck bit is detected in the test, the test itself is unable to proceed all the jumps and returns. Application would be probably damaged as well, but the flag would be set and it would prove the program counter register failure in the post-process diagnostic.

The following is the block diagram of the test principle:

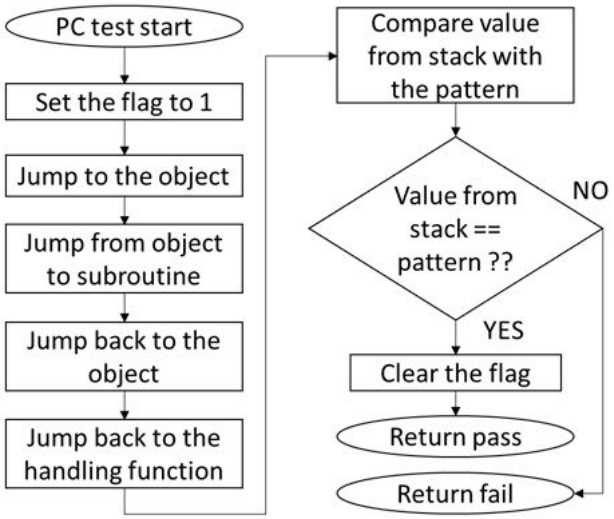

Figure 7. Block diagram of FS_DSC_PC_Test

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 the following table:

Table 10. CPU program counter test in compliance with the 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 main handling function and the subroutine are placed in the *iec60730b_dsc_pc.c* file. The separate object functions are in the *iec60730b_dsc_pc_object1.c* or *iec60730b_dsc_pc_object2.c* files, respectively. All these functions are written in the assembler language. The function prototypes and definitions are in the *iec60730b_dsc_pc.h* file. The library common header files are *iec60730b.h* and *iec60730b_core.h*. These files must be included in the project as well. The correct implementation of the program counter test is as follows.

In the linker command file (**.cmd*), specify the placement of the program counter test objects.

```
SECTIONS
{
PC_test_address_1 = 0x0AAAA; # must be a valid address in the program memory
PC_test_address_2 = 0x15555; # must be a valid address in the program memory
-
.executing_code :
{
-
.= PC_test_address_1;
*(.pc_test_1.text)
.= PC_test_address_2;
*(.pc_test_2.text)
}
```

Note that the *.pc_test_1.text* and *.pc_test_2.text* are the names of the program counter test objects sections from the library.

Specify the address for the pc_test_flag. Be sure, that the data in this address remains unchanged after a non-power-on reset. The data can be specified in the linker command file as well.

```
F_pc_test_flag = F_wd_test_backup - 0x4;
```

The reference of the flag in the source code is as follows:

```
extern UInt16 _pc_test_flag; /* from Linker configuration file */
```

Note that the referenced symbols in the **.cmd* file have the "F_" prefix, while the symbol in the application is written without “F”.

Finally, call the test function separately with both objects.

```
IEC60730B_pc_test_result = FS_PC_Test(FS_PC_object_1, FS_PC_subroutine, (UInt16 *)&_pc_test_flag);
if(FS_PC_FAIL_CODE == IEC60730B_pc_test_result)
SafetyError();
```

```
IEC60730B_pc_test_result = FS_PC_Test(FS_PC_object_2, FS_PC_subroutine, (UInt16 *)&_pc_test_flag);
if(FS_PC_FAIL_CODE == IEC60730B_pc_test_result)
SafetyError();
```

6.2.1 FS_PC_Test()

This is the main handling function of the program counter test. This function calls two other functions (the subroutine and object functions). Both are entered as the input parameters. The third parameter is a flag, which can be useful in the diagnostics after a failure of the program counter register.

Function prototype:

FS_RESULT FS_PC_Test(tFcn_pc pObject_function, tFcn_pc pSubroutine, unsigned short PC_flag);*

Function inputs:

pObject_function - The name of the object function (FS_PC_object_1 or FS_PC_object_2).

pSubroutine - This must be always the name of the subroutine (FS_PC_subroutine).

* PC_flag - The pointer to a flag that would not be overwritten after a non-power-on reset.

Function output:

typedef uint16_t FS_RESULT;

It can have the following two values:

FS_PC_FAIL (0x00000201)

FS_PC_PASS (0)

In case of an incorrect test execution, the PC_flag has a value of 1. TBD

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_subroutine()

The only interaction with this function is that it must be entered as the second input parameter of the FS_PC_Test() 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 36 B.[1]

Calling restrictions:

It must be used only as the parameter of the FS_PC_Test function. It cannot be used separately.

6.2.3 FS_PC_Object()

These are two standalone functions (vith suffixes "_1" and "_2") and they both occupy the standalone source file. These files should be placed in appropriate addresses in the program memory as linkable objects.

Function prototype:

void FS_PC_object_1(void);

void FS_PC_object_2(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 36 B.[1]

Calling restrictions:

It must be used only as the parameter of the FS_PC_Test function. It cannot be used separately.

Chapter 7
Variable memory test

The variable memory (RAM) test checks the on-chip RAM for DC faults. The March C and March X schemes are used as the control mechanisms. The March C or March X schemes can be selected to perform the test. The handling functions for the after-reset test ar different from the functions for the runtime test. Reserve some backup area defined in the RAM for both functions. The size of this area must be at least the same as the tested block size. A RAM test is destructive. This is because the data from the memory area with the variables, the stack area, and the functions placed in the RAM are moved away, rewritten multiple times, 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 the following figures:

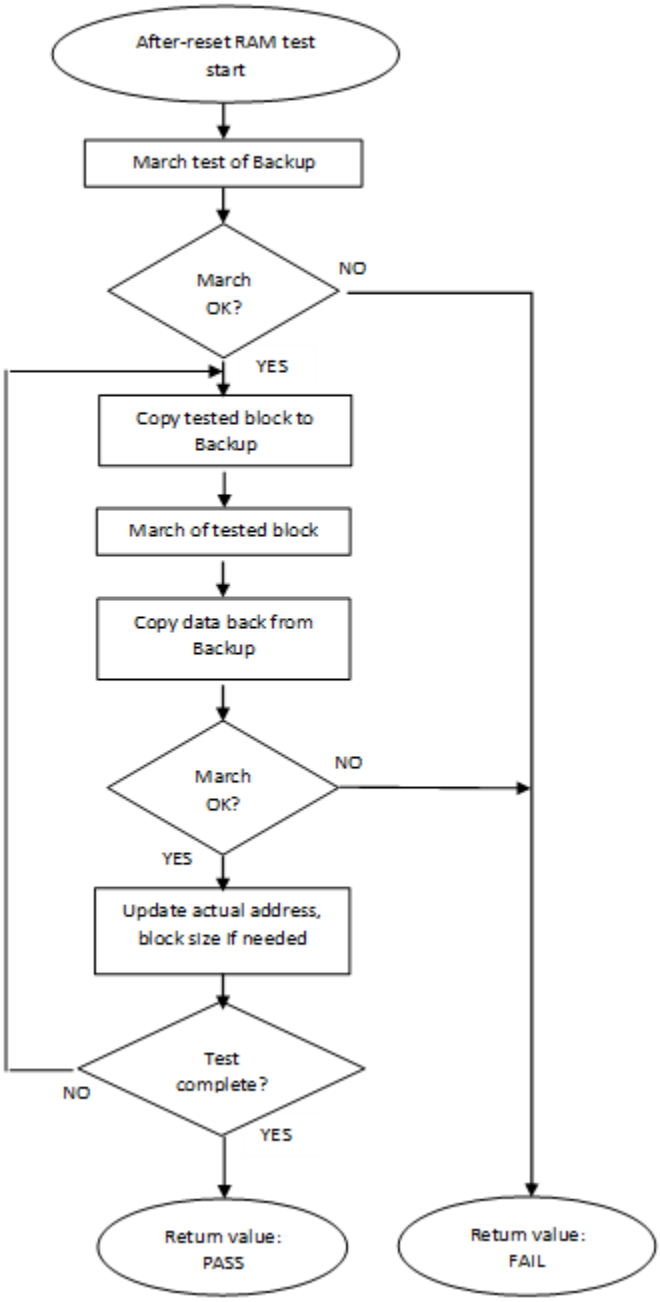

Figure 8. Block diagram for after-reset test of RAM

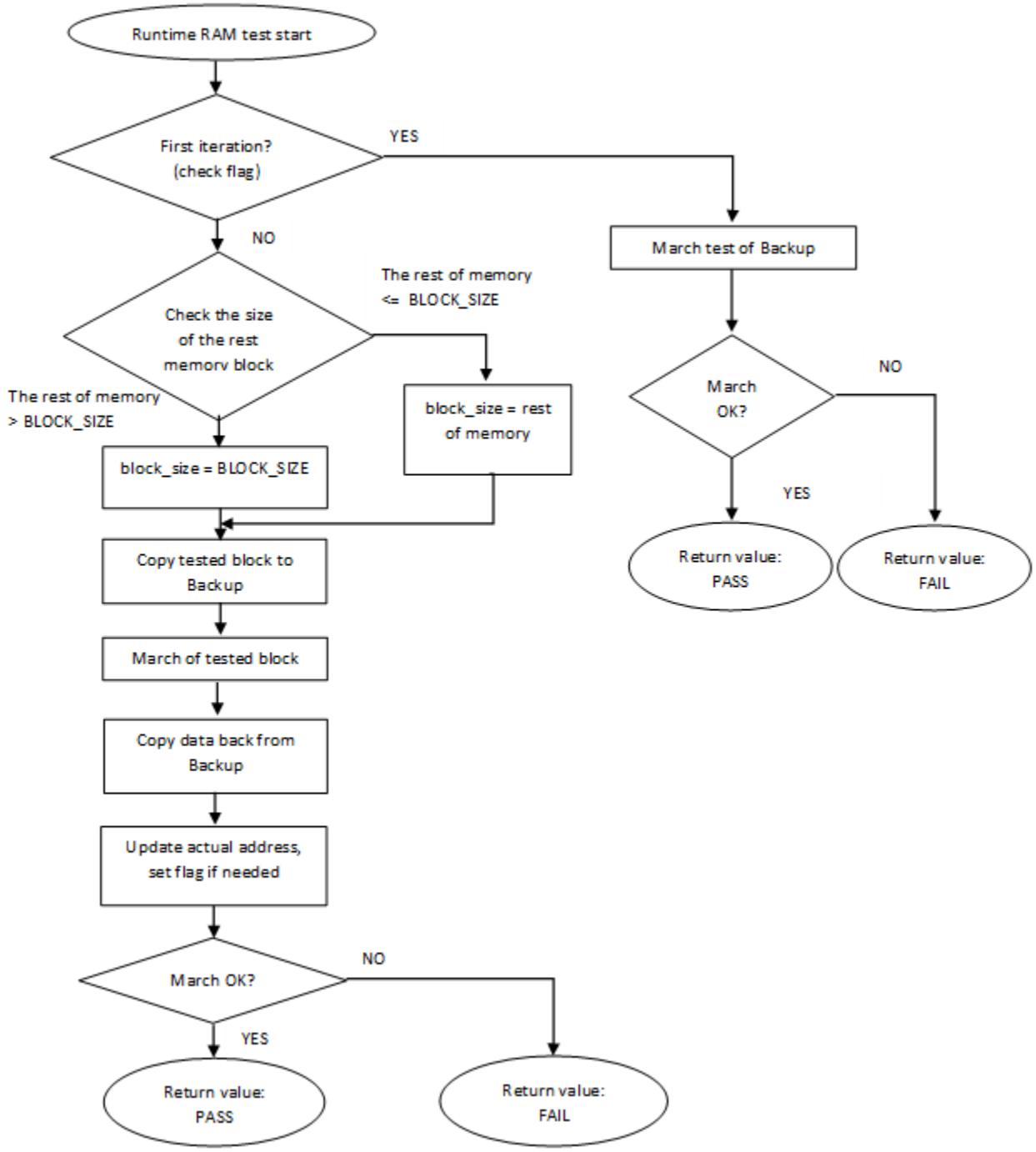

Figure 9. Block diagram for runtime test of RAM

7.1 Variable 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 the following table:

Table 11. 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 are in the *iec60730b_dsc_ram.c* file. The functions are written in the assembler language. The prototypes and definitions are placed in the *iec60730b_dsc_ram.h* file. The library common header files are *iec60730b.h* and *iec60730b_core.h*. These must be included in the project as well.

The correct implementation of the variable memory test can be as follows.

First of all, define the backup memory properly. It must have at least the same size as the value of the block_size parameter. Make sure that this memory is not used for any other purposes.

Have the tested data grouped in a specific section. While working on the application, you do not have to adjust the start address and test range manually with every new variable added to the source code.

The RAM test consists of the following 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.

```
#include "iec60730b.h"

extern uint32_t _ram_test_backup; /* from Linker command file */
extern uint32_t _ram_test_backup_size; /* from Linker command file */
#define RAM_TEST_BACKUP_SIZE (uint32_t)&_ram_test_backup_size

#define RAM_TEST_BLOCK_SIZE 0x4

uint32_t pui32SafetyRamSectionStart = (uint32_t)&_safety_ram;
uint32_t pui32SafetyRamSectionEnd = (uint32_t)&_end_safety_ram;
uint32_t *ui32SafetyRamStart = (uint32_t *)pui32SafetyRamSectionStart;
uint32_t *ui32SafetyRamEnd = (uint32_t *)pui32SafetyRamSectionEnd;

ui32RamTestStartAddress = (uint32_t)pui32SafetyRamStart;
ui32RamTestEndAddress = (uint32_t)pui32SafetyRamEnd;
ui16DefaultBlockSize = RAM_TEST_BACKUP_SIZE;
ui16BlockSize = RAM_TEST_BLOCK_SIZE;
ui32ActualAddress = ui32RamTestStartAddress;
ui32BackupAddress = (uint32_t)&_ram_test_backup;
```

```
ui16RamTestRange = ui32RamTestEndAddress - ui32RamTestStartAddress;

/* after reset test */
RAM_test_result = FS_RAM_AfterReset(
(unsigned long *)ui32RamTestStartAddress,
(unsigned long *)ui32BackupAddress,
FS_RAM_SegmentMarchX,
(unsigned short)ui16BlockSize,
(unsigned short)ui16RamTestRange
);

/* runtime test iteration */
RAM_test_result = FS_RAM_Runtime(
(unsigned long *)ui32RamTestStartAddress,
(unsigned long *)ui32BackupAddress,
FS_RAM_SegmentMarchX,
(unsigned long *)&ui32ActualAddress,
(unsigned short)ui16BlockSize,
(unsigned short)ui16RamTestRange
);

/* handling an error */
if(RAM_test_result == FS_RAM_FAIL)
{
safetyErrors |= RAM_TEST_ERROR;
SafetyErrorHandling();
}
```

7.2.1 FS_RAM_AfterReset()

The after-reset test is done by the FS_RAM_AfterReset function. This function is called once after reset, when the execution time is not critical. Reserve some free memory space for the backup area. The block size parameter cannot be larger than the size of the backup area. 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 using 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. The block diagram is shown in Figure 8.

Function prototype:

*FS_RESULT FS_RAM_AfterReset(unsigned long *start_address, unsigned long *backup_address, tFcn_ram pMarch_type, unsigned short block_size, unsigned short test_range);*

Function inputs:

*start_address - The first address of the tested memory.

*backup_address - The address of the dedicated memory where the data from the tested memory are backed up.

pMarch_type - The pointer to the March function from the library. It specifies which March pattern is used.

block_size - The size of the test iteration. It cannot be larger than the size of the backup memory.

test_range - The size of the tested memory.

Function output:

typedef unsigned long FS_RESULT;

It can have the following two values:

FS_RAM_FAIL (0x00000401)

FS_RAM_PASS (0)

Function performance:

The function size is 92 B.[1]

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

Table 12. FS_RAM_AfterReset duration

Memory size (Bytes)	Block size (Bytes)	Duration time - March X	Duration time – March C
0x100	0x20	215.4 µs	321.2 µs
0x100	0x40	220.5 µs	337.0 µs
0x100	0x80	249.1 µs	388.1 µs
0x200	0x20	408.6 µs	608.7 µs
0x200	0x40	401.6 µs	611.3 µs
0x200	0x80	424.1 µs	655.8 µs
0x400	0x20	795.2 µs	1183.5 µs
0x400	0x40	763.9 µs	1159.5 µs
0x400	0x80	774.1 µs	1191.5 µs

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 have at least the same size as the tested block size defined by the block_size parameter.

7.2.2 FS_RAM_Runtime()

The runtime test is done by the FS_RAM_Runtime() function. This function performs the runtime test of the variable memory. Reserve some free memory space dedicated for the backup area. The block size parameter cannot be larger than the size of the backup area. 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 by the respective March test. The data is copied back to the original memory area and the actual address and the block size are updated. This is repeated until the last block of memory is tested. If a DC fault is detected, the function returns a fail pattern.

The block diagram is shown in Figure 9. The example of the function call is as follows:

Function prototype:

*FS_RESULT FS_RAM_Runtime(unsigned long *start_address, unsigned long *backup_address, tFcn_ram pMarch_type, unsigned long *actual_address, unsigned short block_size, unsigned short test_range);*

Function inputs:

*start_address - The first address of the tested memory.

*backup_address - The address of the dedicated memory where the data from the tested memory are backed up.

pMarch_type - The pointer to the March function from the library. It specifies the March pattern used.

*actual_address - The adress of the variable that holds the actual address value.

block_size - The size of the test iteration. It cannot be larger than the size of the backup memory.

test_range - The size of the tested memory.

Function output:

typedef unsigned long FS_RESULT;

It can have the following two values:

FS_RAM_FAIL (0x00000401)

FS_RAM_PASS (0)

Function performance:

The function size is 118 B.[1]

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

Table 13. FS_RAM_Runtime duration

Block size (Bytes)	Duration time - March X	Duration time - March C
0x4	8.50 µs	10.50 µs
0x8	11.12 µs	14.63 µs
0x20	21.63 µs	38.75 µs
0x40	43.00 µs	63.37 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 execution time depends on the block size.

7.2.3 FS_RAM_SegmentMarchC()

This function performs a 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. It is used as an input parameter of the FS_RAM_AfterReset and FS_RAM_Runtime functions.

Function prototype:

*FS_RESULT FS_RAM_SegmentMarchC(unsigned long *start_address, unsigned short block_size);*

Function inputs:

*start_address - The first address to perform the March test on.

block_size - The size of the memory to test.

Function output:

typedef unsigned long FS_RESULT;

This function can have the following two values:

FS_RAM_FAIL (0x00000401)

FS_RAM_PASS (0)

Function performance:

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

Calling restrictions:

This function cannot be interrupted.

7.2.4 FS_RAM_SegmentMarchX()

This function performs a 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(unsigned long *start_address, unsigned short block_size);*

Function inputs:

*start_address - The first address to perform the March test on.

block_size - The size of the memory to test.

Function output:

typedef unsigned long FS_RESULT;

This function can have the following two values:

FS_RAM_FAIL (0x00000401)

FS_RAM_PASS (0)

Function performance:

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

7.2.5 FS_RAM_CopyMemory()

this is the internal function from the safety library. This function is used within the FS_RAM_AfterReset and FS_RAM_Runtime functions to copy data from the tested memory to the backup and the other way round.

Function prototype:

*void FS_RAM_CopyMemory(unsigned long *pMemSource, unsigned long *pMemDest, unsigned short uintLength);*

Function inputs:

*pMemSource - The source address.

*pMemDest - The destination address.

uintLength - The size of the block to be copied.

Function output:

Void. The function does not return any values.

Function performance:

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

Chapter 8
CPU register test

The CPU register test checks all the CPU registers of the DSC 56800EX core, except for the program counter register which is checked by a separate test.

The registers are tested for a stuck-at condition by writing and reading of at least two opposite patterns. If possible, the patterns consist of an alternating sequence of zeros and ones (0101…) applied to all bits of the respective register.

The tested registers are as follows:

- ALU registers A, B, C, D, Y, X0
- AGU registers R0, R1, R2, R3, R4, R5, N, M01, N3
- shadowed AGU registers R0, R1, R2, R3, R4, R5, N, M01, N3
- status register SR
- stack pointer SP
- loop registers LC, LC2, LA, LA2, HWS, HWS1

The identification of the safety error/failure of a register is ensured by a specific return value of the corresponding function. In some special cases, the error is not reported by the fail return, because it would require the action of a corrupted register. In that case, the function waits in an endless loop for reset.

The block diagrams of the respective register tests are shown in the following figures:

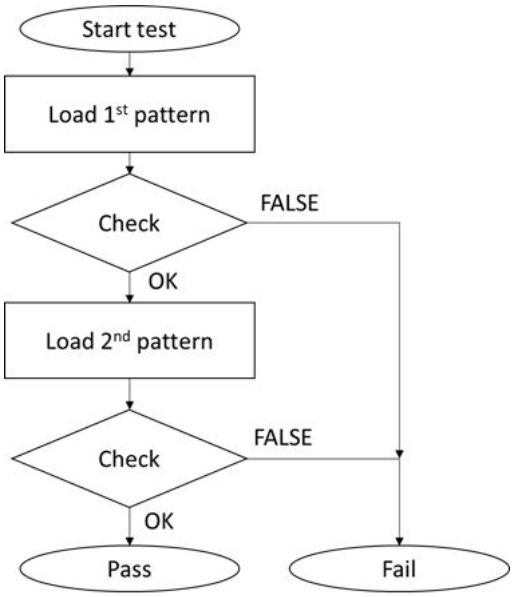

Figure 10. Block diagram for data registers X0, A, B, C, D, pointer registers R0, R1, R2, R3, R4, R5, N, M01, N3, included shadowed registers

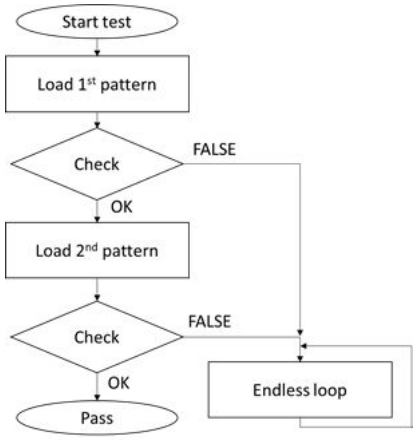

Figure 11. Block diagram for data register Y

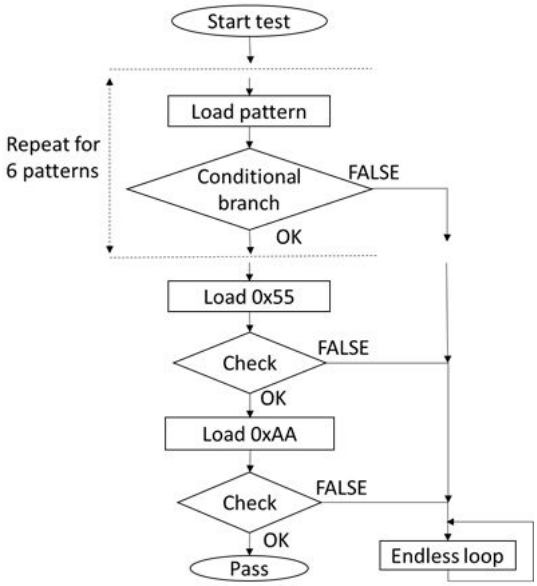

Figure 12. Block diagram for status register SR

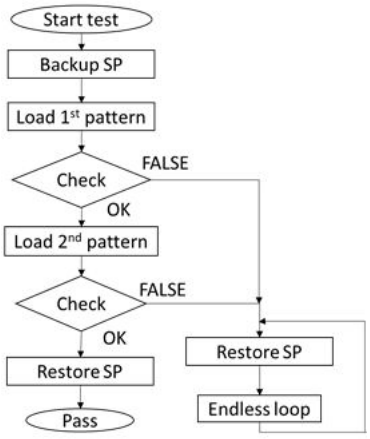

Figure 13. Block diagram for stack pointer register SP

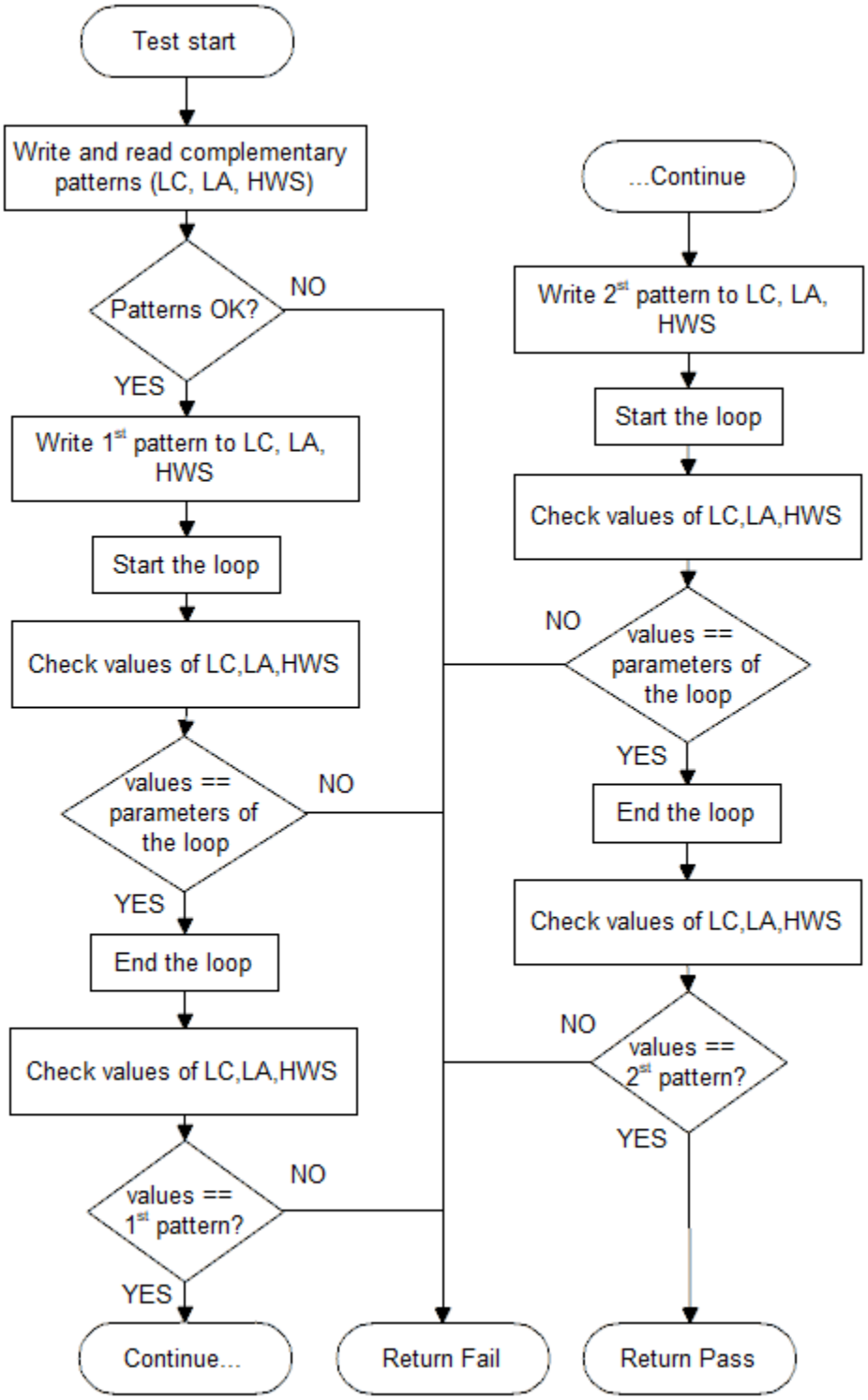

Figure 14. Block diagram for loop registers

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 the following table:

Table 14. CPU registers 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_dsc_reg.c* file and written as assembler functions. The header file with the return values and function prototypes is *iec60730b_dsc_reg.h*.

The *iec60730b_dsc_core.h* and *iec60730b_dsc.h* files are included and must be also placed in the project.

The CPU register test is performed by the following functions:

- FS_CPU_DataRegisters()
- FS_CPU_PointerRegisters()
- FS_CPU_ShadowRegistersE()
- FS_CPU_ShadowRegistersEX()
- FS_CPU_StatusRegister()
- FS_CPU_StackPointer()
- FS_CPU_LoopRegisters()

The error detection is recognized by the specific return value, as described in the following sections. There are several exceptions. If some of the registers (Y, SR, SP) are corrupted, the application stays in an endless loop instead of the return error value. The use of functions is the same after the reset and during runtime. The developer must pay attention to the functions when used during runtime, as described in the following chapters. The use of functions is the same after the reset and during runtime. Pay attention to the functions when used during runtime, as described in the calling restriction sub-chapters. The example of function calling is as follows.

The following is an example of a function call:

```
#include "iec60730b_dsc.h"
if(REG_OK != FS_CPU_DataRegisters())
SafetyError();
```

8.2.1 FS_CPU_DataRegisters()

This function checks registers Y, X0, A, B, C, and D in a sequence. The registers are tested according to the block diagrams in Figure 10 and Figure 11.

Function prototype:

FS_RESULT FS_CPU_DataRegisters(void);

Test patterns for respective registers:

Y : 0x5555_5555, 0xAAAA_AAAA

X0: 0x5555, 0xAAAA

A, B, C, D: 0x5555_5555_5, 0xAAAA_AAAA_A

Function inputs:

Void. No inputs are passed into the function.

Function output:

The function can return the following values:

REG_OK 0

REG_FAIL_X0 3

REG_FAIL_A 4

REG_FAIL_B 5

REG_FAIL_C 6

REG_FAIL_D 7

Else, it can stay in an endless loop if the Y register is corrupted. 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:

None.

8.2.2 FS_CPU_LoopRegisters()

This function checks registers LC, LC2, LA, LA2, HWS, HWS1 in a sequence. Registers are tested according to block diagram in Figure 14.

Function prototype:

FS_RESULT FS_CPU_LoopRegisters(void);

Test patterns for respective registers:

LC, LC2: 0x5555, 0xAAAA

LA, LA2, HWS, HWS1: 0x5555_55, 0xAAAA_AA

Function inputs:

Void. No inputs are passed into the function.

Function output:

typedef unsigned long FS_RESULT;

The function can return the following values:

REG_OK

REG_FAIL_LC

REG_FAIL_LA

REG_FAIL_HWS

REG_FAIL_HWS1

REG_FAIL_LA2

REG_FAIL_LC2

Function performance:

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

Calling restrictions:

None.

8.2.3 FS_CPU_PointerRegisters()

This function checks pointer registers R0, R1, R2, R3, R4, R5, N, M01, N3 in a sequence. Registers are tested according to block diagrams in Figure 10.

Function prototype:

FS_RESULT FS_CPU_PointerRegisters(void);

Test pattern:

R0 -R5, N: 0x5555_55, 0xAAAA_AA

N3, M01: 0x5555, 0xAAAA

Function inputs:

Void. No inputs are passed into the function.

Function output:

typedef unsigned long FS_RESULT;

This function can return the following values:

REG_OK 0

REG_FAIL_R0 15

REG_FAIL_R1 16

REG_FAIL_R2 17

REG_FAIL_R3 18

REG_FAIL_R4 19

REG_FAIL_R5 20

REG_FAIL_N 21

REG_FAIL_M01 22

REG_FAIL_N3 23

Function performance:

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

Calling restrictions:

None.

8.2.4 FS_CPU_ShadowRegistersE()

This function checks shadowed registers R2, R3, R4, R5, N3 in a sequence. The registers are tested according to the block diagram in Figure 10.

Function prototype:

FS_RESULT FS_CPU_ShadowRegistersE(void);

Test patterns for respective registers:

TBD !!!

Function inputs:

Void. No inputs are passed into the function.

Function output:

typedef unsigned long FS_RESULT;

The function can return the following values:

REG_OK 0

REG_FAIL_N3S 38

REG_FAIL_R2S 39

REG_FAIL_R3S 40

REG_FAIL_R4S 41

REG_FAIL_R5S 42

Function performance:

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

Calling restrictions:

This function cannot be interrupted.

8.2.5 FS_CPU_ShadowRegistersEX()

This function checks shadowed registers R0, R1, N, M01 in a sequence. The registers are tested according to the block diagram in Figure 10.

Function prototype:

FS_RESULT FS_CPU_ShadowRegistersEX(void);

Test pattern:

TBD !!

Function inputs:

Void. No inputs are passed into the function.

Function output:

typedef unsigned long FS_RESULT;

The function can return the following values:

REG_OK

REG_FAIL_R2S

REG_FAIL_R3S

REG_FAIL_R4S

REG_FAIL_R5S

REG_FAIL_N3S

Function performance:

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

Calling restrictions:

None.

8.2.6 FS_CPU_StackPointer()

This function checks the stack pointer register according to the block diagram in Figure 13.

Function prototype:

FS_RESULT FS_CPU_StackPointer(void);

Test pattern:

SP: 0x5555_55, 0xAAAA_AA

Function inputs:

Void. No inputs are passed into the function.

Function output:

typedef unsigned long FS_RESULT;

The function can return REG_OK, or it can stay in an endless loop in case of an error. 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:

None.

8.2.7 FS_CPU_StatusRegister()

This function checks the status register. It is tested according to the block diagram in Figure 12.

Function prototype:

FS_RESULT FS_CPU_StatusRegister(void);

Test pattern:

SR: 0x55, 0xAA, 0x8, 0x12, 0x20, 0x30

Function inputs:

Void. No inputs are passed into the function.

Function output:

typedef unsigned long FS_RESULT;

The function can return REG_OK, or it can stay in an endless loop in case of an error.

Function performance:

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

Calling restrictions:

None.

Chapter 9
Stack test

This test routine tests the overflow and underflow conditions of the application stack. The overflow or underflow of the stack can occur if the stack is controlled incorrectly, or when a very small stack area is defined for the given application.

The principle of the test is to fill the area below and above the stack with a known pattern. Define these areas in the linker configuration file in the same way as the stack. The initialization function then fills these areas with your defined pattern. The pattern must have a value that does not appear elsewhere in the application. The test is performed after the reset and during 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 of 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, not directly specified in the IEC60730 annex H table.

9.2 Stack test implementation

The test functions for the stack and the initialization function are placed in the *iec60730b_dsc_stack.c* file. The functions are written in the assembler language. The prototypes and definitions are in the *iec60730b_dsc_stack.h* file. The library common header files are *iec60730b.h* and *iec60730b_core.h*. They must be included in the project as well.

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 here as well. There are several methods to achieve this and only one example is shown here. The higher the size dedicated for the tested areas, the greater the chance that a potential error is recognized in time.

The following is an example code from the CodeWarrior linker command file (**.cmd*).

```
_min_stack_size = 0x100;
F_stack_test_block_size = 0x8;
F_stack_test_p_1 = _HEAP_END + 1;
F_stack_test_p_2 = F_stack_test_p_1 + F_stack_test_block_size;
. = F_stack_test_p_2;
. = ALIGN(4);
_stack_addr = . + 1;
_stack_end = _stack_addr + _min_stack_size;
. = _stack_end;
F_stack_test_p_3 = _stack_end + 1;
F_stack_test_p_4 = F_stack_test_p_3 + F_stack_test_block_size;
. = F_stack_test_p_4 + 1;
} > .x_internal_RAM # internal RAM for data

// _____________
// |_____________| --> F_stack_test_p_1 ....ADR
// |_____________| ....ADR + 0x1
```

```
// |_____________| ....
// |_____________| --> F_stack_test_p_2 ....ADR + 0x8
// | |--> _stack_addr
// | |
// | |
// | STACK |
// | |
// | |
// | |
// | |
// |_____________|
// |_____________| --> F_stack_test_p_3
// |_____________|
// |_____________|
// |_____________| --> F_stack_test_p_4
```

9.2.2 FS_StackInit()

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

Example of initialization:

```
#include "iec60730b.h"

#define STACK_TEST_PATTERN 0x77777777

extern uint32_t _stack_test_block_size;
#define STACK_TEST_BLOCK_SIZE (uint32_t)&_stack_test_block_size

extern uint16_t _stack_test_p_2;
extern uint16_t _stack_test_p_3;

FS_StackInit(STACK_TEST_PATTERN, (uint16_t *)&_stack_test_p_2, (uint16_t *)&_stack_test_p_3,
STACK_TEST_BLOCK_SIZE);
```

Function prototype:

*void FS_StackInit(unsigned short test_pattern, unsigned short *first_address,unsigned short *second_address, unsigned short block_size);*

Function inputs:

test_pattern - The pattern to be written into the areas.

*first_address - The address closest to the beginning of the stack.

*second_address - The address closest to the end of the stack.

block_size - The size of areas.

Function output:

Void. The function does not return any values.

Function performance:

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

Calling restrictions:

None.

9.2.3 FS_StackTest()

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

Function prototype:

*FS_RESULT FS_StackTest(unsigned short test_pattern, unsigned short *first_address, unsigned short *second_address, unsigned short block_size);*

Function inputs:

test_pattern - The pattern to be written into the areas.

*first_address - The address closest to the beginning of the stack.

*second_address - The address closest to the end of the stack.

block_size - The size of the areas.

Function output:

typedef unsigned long FS_RESULT;

It can have the following two values:

FS_STACK_FAIL (0x00000501)

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 watchdog timeout comparison. The first function for the watchdog testing (setup function) is called after that. This function refreshes the watchdog timer, clears the reset counter, clears the “test uncomplete” flag, 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 check 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 when choosing the clock source for the watchdog and 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 described in the following table:

Table 15. 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_dsc _wdg.c* file. The header file is *iec60730b_dsc_wdg.h*. The *iec60730b.h* and iec60730b_core*.h* files must be placed in the application as well.

You must have some free 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 three members. It is defined in the *iec60730b_dsc _wdg.h* file.

An important condition for performing the watchdog test is to configure the watchdog module and the device timer before the beginning of the test.

Ensure the handling of the functions. To identify what is the source of the reset, use the reset status register (SIM_RSTAT). The common configuration is that 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.

An example of the watchdog test implementation is as follows:

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

```
#include "sys.h"
#include "intc.h"
#include "cop.h"
#include "qs.h"
#include "pit.h"

extern uint32_t _wd_test_backup; /* from Linker configuration file */
#define WATCHDOG_TEST_VARIABLES ((fs_wdog_test *) &_wd_test_backup)

#define ENDLESS_LOOP_ENABLE 0 /* set 1 or 0 */
#define WATCHDOG_RESETS_LIMIT 1000

typedef struct _wd_test
{
uint32_t ui32WdTestExpected;
uint32_t ui32WdTestTolerance;
uint16_t ui16WdTestLimitHigh;
uint16_t ui16WdTestLimitLow;
uint16_t ui16WatchdogResets;
uint16_t ui16WatchdogTimeoutCheck;
uint16_t ui16WatchdogRefreshRatio;
uint16_t ui16WatchdogErrorFlag;
} wd_test_t;

wd_test_t g_sSafetyWdTest;

ui16ResetPor = ioctl(SYS, SYS_TEST_RESET_SOURCE, SYS_POWER_ON_RESET);
ui16ResetCop = ioctl(SYS, SYS_TEST_RESET_SOURCE, SYS_COP_RESET);

if(ui16ResetPor)
{
ioctl(COP,COP_INIT,NULL);
ioctl(PIT_1, PIT_INIT, NULL);
FS_watchdog_setup(WATCHDOG_TEST_VARIABLES, PIT1_BASE);
}
if(ui16ResetCop)
{
FS_watchdog_check(12000, 8000, WATCHDOG_RESETS_LIMIT, ENDLESS_LOOP_ENABLE,
WATCHDOG_TEST_VARIABLES);
}

g_sSafetyWdTest.ui16WatchdogResets = WATCHDOG_TEST_VARIABLES->resets;
g_sSafetyWdTest.ui16WatchdogTimeoutCheck = WATCHDOG_TEST_VARIABLES->counter;
```

```
g_sSafetyWdTest.ui16WatchdogErrorFlag =
WATCHDOG_TEST_VARIABLES->wd_test_uncomplete_flag;
```

10.2.1 FS_watchdog_setup()

This function clears the reset counter and the “test uncomplete” flag. They are members of the fs_wdog_test structure. It refreshes the watchdog to start counting from zero. Within the waiting endless loop, the value from the PIT counter is periodically stored in the reserved area in the RAM.

The function never reaches its exit, it is waiting in an endless loop for a reset.

Function prototype:

void FS_watchdog_setup(WD_Test_Str pWatchdogBackup, unsigned short pit_base);*

Function inputs:

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

pit_base - The base address of the PIT timer.

Function output:

Void. The function does not return any values.

Function performance:

The function duration depends on the watchdog counter timeout value.

Calling restrictions:

The watchdog timer and the PIT must be configured correctly. A variable of the fs_wdog_test type must be declared and placed into a reliable place. Interrupts should be disabled during the function execution.

10.2.2 FS_watchdog_check()

This function compares the captured value of the PIT 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 the 1 or 0 values). If enabled, the function ends up in an endless loop in the following cases:

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

Function prototype:

void FS_watchdog_check(unsigned short limit_high, unsigned short limit_low, unsigned short resets_limit, unsigned short endless_loop_enable, WD_Test_Str pWatchdogBackup);*

Function inputs:

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

limit_low - The pre-calculated low limit value of the PIT counter in the watchdog test.

resets_limit - The defined limit of 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 any values.

Function performance:

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

Calling restrictions:

Execute the respective setup function 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:

Document identifier: IEC60730B56800EXLUG40