

# IEC60730\_B\_CM0\_Library\_UG\_v4\_0

IEC60730B Library User's Guide



# Contents

Chapter 1 Core self-test library.....	3
Chapter 2 Analog Input/Output (IO) test.....	14
Chapter 3 Clock test.....	24
Chapter 4 Digital input/output test.....	29
Chapter 5 Invariable memory test.....	51
Chapter 6 CPU program counter test.....	57
Chapter 7 Variable memory test.....	61
Chapter 8 CPU register test.....	69
Chapter 9 Stack test.....	78
Chapter 10 TSI tests.....	81
Chapter 11 Watchdog test .....	91

# Chapter 1

## Core self-test library

The core self-test library provides 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 IAR, Keil, and MCUXpresso IDEs. 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
- Touch sensing interface test

The test architecture, implementation, test, and validation of corresponding tests are comprehensively described in independent sections for each test.

The library supports the MKV1x, MKLxx, MKE0x, MKE1xZ, LPC84x, LPC82x, LPC80x, and MK32W0x/cm0+ families based on the Arm-M0+ core.

The core self-test library has two versions: source code version and object code version. Three object codes (one for each IDE) were 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 are the object files for the given IDEs:

- IAR: *IEC60730B\_IAR\_v4\_0.a*
- Keil: *IEC60730B\_Keil\_v4\_0.lib*
- MCUXpresso: *libIEC60730B\_MCUX\_v4\_0.a*

### 1.2 Core self-test library – source code version

The library name is IEC60730B\_CM0. The main header files are *iec60730b.h* and *iec60730b\_core.h*.

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

**Table 1. List of library items**

File Name	Test Type	Function Name
iec60730b.h	Library header file	-
iec60730b_core.h	Core depend library header file	-

*Table continues on the next page...*

**Table 1. List of library items (continued)**

File Name	Test Type	Function Name
asm_mac_common.h	Common assembler directives	-
iec60730b_aio.c	Analog I/O test	FS_AIO_InputInit()
	Analog I/O test	FS_AIO_InputInit_CYCLIC()
	Analog I/O test	FS_AIO_InputInit_LPC_ADC16()
	Analog I/O test	FS_AIO_InputInit_LPC_ADC12()
	Analog I/O test	FS_AIO_InputTrigger()
	Analog I/O test	FS_AIO_InputSet()
	Analog I/O test	FS_AIO_InputSet_K3S()
	Analog I/O test	FS_AIO_InputSet_CYCLIC()
	Analog I/O test	FS_AIO_InputSet_LPC8XX()
	Analog I/O test	FS_AIO_InputSet_LPC55SXX()
	Analog I/O test	FS_AIO_InputCheck()
	Analog I/O test	FS_AIO_InputCheck_KE()
	Analog I/O test	FS_AIO_InputCheck_K3S()
	Analog I/O test	FS_AIO_InputCheck_CYCLIC()
	Analog I/O test	FS_AIO_InputCheck_LPC8XX()
Analog I/O test	FS_AIO_InputCheck_LPC55SXX()	
iec60730b_clock.c	Clock test	FS_CLK_Check()
	Clock test	FS_CLK_Init()
	Clock test	FS_CLK_LPTMR()
	Clock test	FS_CLK_RTC()
	Clock test	FS_CLK_GPT()
	Clock test	FS_CLK_WKT_LPC()
	Clock test	FS_CLK_TIMER_LPC()
iec60730b_dio.c	Digital I/O test	FS_DIO_Input()
	Digital I/O test	FS_DIO_Output()
	Digital I/O test	FS_DIO_Output_RT()
	Digital I/O test	FS_DIO_Output_LPC()
iec60730b_dio_ext.c	Extended Digital I/O test	FS_DIO_InputExt()
	Extended Digital I/O test	FS_DIO_ShortToSupplySet()
	Extended Digital I/O test	FS_DIO_ShortToAdjSet()
	Extended Digital I/O test	FS_DIO_InputExt_RT()

*Table continues on the next page...*

**Table 1. List of library items (continued)**

File Name	Test Type	Function Name
	Extended Digital I/O test	FS_DIO_ShortToAdjSet_RT()
	Extended Digital I/O test	FS_DIO_InputExt_LPC()
	Extended Digital I/O test	FS_DIO_ShortToSupplySet_LPC()
	Extended Digital I/O test	FS_DIO_ShortToAdjSet_LPC()
iec60730b_flash.S	Invariable memory test (Flash)	FS_CM0_FLASH_HW16()
	Invariable memory test (Flash)	FS_CM0_FLASH_HW16_LPC
	Invariable memory test (Flash)	FS_CM0_FLASH_SW16()
iec60730b_pc.S	Program Counter test	FS_CM0_PC_Test()
iec60730b_pc_object.S	Program Counter test	FS_PC_Object()
iec60730b_ram.S	Variable memory test (RAM)	FS_CM0_RAM_AfterReset()
	Variable memory test (RAM)	FS_CM0_RAM_Runtime()
	Variable memory test (RAM)	FS_CM0_RAM_CopyToBackup()
	Variable memory test (RAM)	FS_CM0_RAM_CopyFromBackup()
	Variable memory test (RAM)	FS_CM0_RAM_SegmentMarchC()
	Variable memory test (RAM)	FS_CM0_RAM_SegmentMarchX()
iec60730b_reg.S	Register test	FS_CM0_CPU_Register()
	Register test	FS_CM0_CPU_NonStackedRegister()
	Register test	FS_CM0_CPU_Primask()
	Register test	FS_CM0_CPU_SPmain()
	Register test	FS_CM0_CPU_SPprocess()
	Register test	FS_CM0_CPU_Control()
iec60730b_stack.S	Stack test	FS_CM0_STACK_Init()
	Stack test	FS_CM0_STACK_Test()
iec60730b_tsi.c	Touch Sensing Interface test	FS_TSI_InputInit()
	Touch Sensing Interface test	FS_TSI_InputStimulate()
	Touch Sensing Interface test	FS_TSI_InputRelease()
	Touch Sensing Interface test	FS_TSI_InputCheckNONStimulated()
	Touch Sensing Interface test	FS_TSI_InputCheckStimulated()
iec60730b_wdg.c	Watchdog test	FS_WDOG_Setup()
	Watchdog test	FS_WDOG_Check()
	Watchdog test	FS_WDOG_Setup_KE0XZ()
	Watchdog test	FS_WDOG_Check_KE0XZ()

*Table continues on the next page...*

**Table 1. List of library items (continued)**

File Name	Test Type	Function Name
	Watchdog test	FS_WDOG_Setup_COP()
	Watchdog test	FS_WDOG_Setup_KE1XZ()
	Watchdog test	FS_WDOG_Check_KE1XZ()
	Watchdog test	FS_WDOG_Setup_KE1XF()
	Watchdog test	FS_WDOG_Check_KE1XF()
	Watchdog test	FS_CM0_WDOG_Setup_K32W()
	Watchdog test	FS_CM0_WDOG_Check_K32W()
	Watchdog test	FS_CM4_WDOG_Setup_K32W()
	Watchdog test	FS_CM4_WDOG_Check_K32W()
	Watchdog test	FS_WDOG_Setup_RT()
	Watchdog test	FS_WDOG_Check_RT()
	Watchdog test	FS_WDOG_Setup_WWDT_LPC()
	Watchdog test	FS_WDOG_Setup_WWDT_LPC_mrt()
	Watchdog test	FS_WDOG_Check_WWDT_LPC()
	Watchdog test	FS_WDOG_Check_WWDT_LPC55SXX()

## 1.2.1 LPC80x dedicated functions

The following table show the list of functions dedicated for the LPC80x device family.

**Table 2. LPC80x dedicated functions**

File	Suitable function	Functions size	Functions duration [µs]
iec60730b_aio.c	<a href="#">FS_AIO_InputInit_LPC_ADC12()</a>	20 bytes <sup>2</sup>	2.23 <sup>2</sup>
	<a href="#">FS_AIO_InputTrigger()</a>	10 bytes <sup>1</sup>	0.31 <sup>1</sup>
	<a href="#">FS_AIO_InputSet_LPC8XX()</a>	130 bytes <sup>2</sup>	7.1 <sup>2</sup>
	<a href="#">FS_AIO_InputCheck_LPC8XX()</a>	136 bytes <sup>2</sup>	1.77 <sup>2</sup>
iec60730b_clock.c	<a href="#">FS_CLK_Check()</a>	38 bytes <sup>1</sup>	0.60 <sup>1</sup>
	<a href="#">FS_CLK_Init()</a>	8 bytes <sup>1</sup>	0.24 <sup>1</sup>
	<a href="#">FS_CLK_WKT_LPC()</a>	60 bytes <sup>2</sup>	3.87 <sup>2</sup>
iec60730b_dio.c	<a href="#">FS_DIO_Output_LPC()</a>	120 bytes <sup>2</sup>	94.87 <sup>2</sup>
iec60730b_dio_ext.c	<a href="#">FS_DIO_InputExt_LPC()</a>	158 bytes <sup>2</sup>	7.93 <sup>2</sup>
	<a href="#">FS_DIO_ShortToSupplySet_LPC()</a>	114 bytes <sup>2</sup>	6.10 <sup>2</sup>
	<a href="#">FS_DIO_ShortToAdjSet_LPC()</a>	192 bytes <sup>2</sup>	9.23 <sup>2</sup>
iec60730b_wdg.c	<a href="#">FS_WDOG_Setup_WWDT_LPC()</a>	52 bytes <sup>2</sup>	-

*Table continues on the next page...*

Table 2. LPC80x dedicated functions (continued)

File	Suitable function	Functions size	Functions duration [μs]
	<a href="#">FS_WDOG_Setup_WWDT_LPC_mrt()</a>	52 bytes <sup>5</sup>	-
	<a href="#">FS_WDOG_Check_WWDT_LPC()</a>	70 bytes <sup>2</sup>	4.03 <sup>2</sup>
iec60730b_flash.S	Function are described in dedicated chapter	See the function dedicated chapter	
iec60730b_pc.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_ram.S	Function are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_reg.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_Stack.S	Function are common for all CM0+ devices	See the function dedicated chapter	

## 1.2.2 LPC82x dedicated functions

The following table shows the list of functions dedicated for the LPC82x device family.

Table 3. LPC82x dedicated functions

File	Suitable function	Functions size	Functions duration [μs]
iec60730b_aio.c	<a href="#">FS_AIO_InputInit_LPC_ADC12()</a>	20 bytes <sup>2</sup>	2.23 <sup>2</sup>
	<a href="#">FS_AIO_InputTrigger()</a>	10 bytes <sup>1</sup>	0.31 <sup>1</sup>
	<a href="#">FS_AIO_InputSet_LPC8XX()</a>	130 bytes <sup>2</sup>	7.1 <sup>2</sup>
	<a href="#">FS_AIO_InputCheck_LPC8XX()</a>	136 bytes <sup>2</sup>	1.77 <sup>2</sup>
iec60730b_clock.c	<a href="#">FS_CLK_Check()</a>	38 bytes <sup>1</sup>	0.60 <sup>1</sup>
	<a href="#">FS_CLK_Init()</a>	8 bytes <sup>1</sup>	0.24 <sup>1</sup>
	<a href="#">FS_CLK_WKT_LPC()</a>	60 bytes <sup>2</sup>	3.87 <sup>2</sup>
iec60730b_dio.c	<a href="#">FS_DIO_Output_LPC()</a>	120 bytes <sup>2</sup>	94.87 <sup>2</sup>
iec60730b_dio_ext.c	<a href="#">FS_DIO_InputExt_LPC()</a>	158 bytes <sup>2</sup>	7.93 <sup>2</sup>
	<a href="#">FS_DIO_ShortToSupplySet_LPC()</a>	114 bytes <sup>2</sup>	6.10 <sup>2</sup>
	<a href="#">FS_DIO_ShortToAdjSet_LPC()</a>	192 bytes <sup>2</sup>	9.23 <sup>2</sup>
iec60730b_wdg.c	<a href="#">FS_WDOG_Setup_WWDT_LPC_mrt()</a>	52 bytes <sup>5</sup>	-
	<a href="#">FS_WDOG_Check_WWDT_LPC()</a>	70 bytes <sup>2</sup>	4.03 <sup>2</sup>
iec60730b_flash.S	Function are described in dedicated chapter	See the function dedicated chapter	

*Table continues on the next page...*

Table 3. LPC82x dedicated functions (continued)

File	Suitable function	Functions size	Functions duration [μs]
iec60730b_pc.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_ram.S	Function are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_reg.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_Stack.S	Function are common for all CM0+ devices	See the function dedicated chapter	

### 1.2.3 LPC84x dedicated functions

The following table shows the list of functions dedicated for the LPC84x device family.

Table 4. LPC84x dedicated functions

File	Suitable function	Functions size	Functions duration [μs]
iec60730b_aio.c	FS_AIO_InputInit_LPC_ADC12()	20 bytes <sup>2</sup>	2.23 <sup>2</sup>
	FS_AIO_InputTrigger()	10 bytes <sup>1</sup>	0.31 <sup>1</sup>
	FS_AIO_InputSet_LPC8XX()	130 bytes <sup>2</sup>	7.1 <sup>2</sup>
	FS_AIO_InputCheck_LPC8XX()	136 bytes <sup>2</sup>	1.77 <sup>2</sup>
iec60730b_clock.c	FS_CLK_Check()	38 bytes <sup>1</sup>	0.60 <sup>1</sup>
	FS_CLK_Init()	8 bytes <sup>1</sup>	0.24 <sup>1</sup>
	FS_CLK_WKT_LPC()	60 bytes <sup>2</sup>	3.87 <sup>2</sup>
iec60730b_dio.c	FS_DIO_Output_LPC()	120 bytes <sup>2</sup>	94.87 <sup>2</sup>
iec60730b_dio_ext.c	FS_DIO_InputExt_LPC()	158 bytes <sup>2</sup>	7.93 <sup>2</sup>
	FS_DIO_ShortToSupplySet_LPC()	114 bytes <sup>2</sup>	6.10 <sup>2</sup>
	FS_DIO_ShortToAdjSet_LPC()	192 bytes <sup>2</sup>	9.23 <sup>2</sup>
iec60730b_wdg.c	FS_WDOG_Setup_WWDT_LPC()	52 bytes <sup>2</sup>	-
	FS_WDOG_Setup_WWDT_LPC_mrt()	52 bytes <sup>5</sup>	-
	FS_WDOG_Check_WWDT_LPC()	70 bytes <sup>2</sup>	4.03 <sup>2</sup>
iec60730b_flash.S	Function are described in dedicated chapter	See the function dedicated chapter	
iec60730b_pc.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_ram.S	Function are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_reg.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_Stack.S	Function are common for all CM0+ devices	See the function dedicated chapter	

### 1.2.4 MK32W dedicated functions

The following table shows the list of functions dedicated for the MK32W device family.



Table 5. MK32W dedicated functions

File	Suitable function	Functions size	Functions duration [μs]
iec60730b_aio.c	<a href="#">FS_AIO_InputInit()</a>	16 bytes <sup>1</sup>	0.44 <sup>1</sup>
	<a href="#">FS_AIO_InputTrigger()</a>	10 bytes <sup>1</sup>	0.31 <sup>1</sup>
	<a href="#">FS_AIO_InputSet_K3S()</a>	84 bytes <sup>7</sup>	1.21 <sup>7</sup>
	<a href="#">FS_AIO_InputCheck_K3S()</a>	138 bytes <sup>7</sup>	0.71 <sup>7</sup>
iec60730b_clock.c	<a href="#">FS_CLK_Check()</a>	38 bytes <sup>1</sup>	0.60 <sup>1</sup>
	<a href="#">FS_CLK_Init()</a>	8 bytes <sup>1</sup>	0.24 <sup>1</sup>
	<a href="#">FS_CLK_LPTMR()</a>	32 bytes <sup>1</sup>	0.55 <sup>1</sup>
iec60730b_dio.c	<a href="#">FS_DIO_Output()</a>	112 bytes <sup>1</sup>	TBD <sup>1</sup>
iec60730b_dio_ext.c	<a href="#">FS_DIO_InputExt()</a>	206 bytes <sup>1</sup>	TBD <sup>1</sup>
	<a href="#">FS_DIO_ShortToSupplySet()</a>	166 bytes <sup>1</sup>	TBD <sup>1</sup>
	<a href="#">FS_DIO_ShortToAdjSet()</a>	250 bytes <sup>1</sup>	TBD <sup>1</sup>
iec60730b_wdg.c	<a href="#">FS_CM0_WDOG_Setup_K3S()</a>	36 bytes <sup>7</sup>	-
	<a href="#">FS_CM0_WDOG_Check_K3S()</a>	72 bytes <sup>7</sup>	1.67 <sup>7</sup>
iec60730b_flash.S	Functions are described in dedicated chapter	See the function dedicated chapter	
iec60730b_pc.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_ram.S	Function are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_reg.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_stack.S	Function are common for all CM0+ devices	See the function dedicated chapter	

## 1.2.5 MKE0x dedicated functions

The following table shows the list of functions dedicated for the MKE0x device family.

Table 6. MKE0x dedicated functions

File	Suitable function	Functions size	Functions duration [μs]
iec60730b_aio.c	<a href="#">FS_AIO_InputInit()</a>	16 bytes <sup>1</sup>	0.44 <sup>1</sup>
	<a href="#">FS_AIO_InputTrigger()</a>	10 bytes <sup>1</sup>	0.31 <sup>1</sup>
	<a href="#">FS_AIO_InputSet()</a>	42 bytes <sup>1</sup>	0.43 <sup>1</sup>
	<a href="#">FS_AIO_InputCheck_KE()</a>	20 bytes <sup>1</sup>	0.49 <sup>1</sup>
iec60730b_clock.c	<a href="#">FS_CLK_Check()</a>	38 bytes <sup>1</sup>	0.60 <sup>1</sup>
	<a href="#">FS_CLK_Init()</a>	8 bytes <sup>1</sup>	0.24 <sup>1</sup>
	<a href="#">FS_CLK_RTC()</a>	20 bytes <sup>1</sup>	2.60 <sup>1</sup>
iec60730b_dio.c	<a href="#">FS_DIO_Output()</a>	112 bytes <sup>1</sup>	24,39 <sup>1</sup>
	<a href="#">FS_DIO_Input()</a>	60 bytes <sup>4</sup>	5.10 <sup>4</sup>

Table continues on the next page...

**Table 6. MKE0x dedicated functions  
(continued)**

File	Suitable function	Functions size	Functions duration [μs]
iec60730b_wdg.c	FS_WDOG_Setup_KE0XZ()	32 bytes <sup>4</sup>	-
	FS_WDOG_Check_KE0XZ() )	74 bytes <sup>4</sup>	5.80 <sup>4</sup>
iec60730b_flash.S	Function are described in dedicated chapter	See the function dedicated chapter	
iec60730b_pc.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_ram.S	Function are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_reg.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_Stack.S	Function are common for all CM0+ devices	See the function dedicated chapter	

## 1.2.6 MKE1xZ dedicated functions

The following table shows the list of functions dedicated for the MKE1xZ device family.

**Table 7. MKE1xZ dedicated functions**

File	Suitable function	Functions size	Functions duration [μs]
iec60730b_aio.c	FS_AIO_InputInit()	16 bytes <sup>1</sup>	0.44 <sup>1</sup>
	FS_AIO_InputTrigger()	10 bytes <sup>1</sup>	0.31 <sup>1</sup>
	FS_AIO_InputSet()	42 bytes <sup>1</sup>	0.43 <sup>1</sup>
	FS_AIO_InputCheck_KE()	106 bytes <sup>1</sup>	0.49 <sup>1</sup>
iec60730b_clock.c	FS_CLK_Check()	38 bytes <sup>1</sup>	0.60 <sup>1</sup>
	FS_CLK_Init()	8 bytes <sup>1</sup>	0.24 <sup>1</sup>
	FS_CLK_LPTMR()	32 bytes <sup>1</sup>	0.55 <sup>1</sup>
iec60730b_dio.c	FS_DIO_Output()	112 bytes <sup>1</sup>	24.39 <sup>1</sup>
iec60730b_dio_ext.c	FS_DIO_InputExt()	206 bytes <sup>1</sup>	2.97 <sup>1</sup>
	FS_DIO_ShortToSupplySet()	166 bytes <sup>1</sup>	2.94 <sup>1</sup>
	FS_DIO_ShortToAdjSet()	250 bytes <sup>1</sup>	4.14 <sup>1</sup>
iec60730b_wdg.c	FS_WDOG_Setup_KE1XZ()	34 bytes <sup>1</sup>	-
	FS_WDOG_Check_KE1XZ()	74 bytes <sup>1</sup>	1.35 <sup>1</sup>
iec60730b_tsi.c	FS_TSI_InputInit()	28 bytes <sup>1</sup>	0.61 <sup>1</sup>
	FS_TSI_InputStimulate()	182 bytes <sup>1</sup>	2.59 <sup>1</sup>
	FS_TSI_InputRelease()	116 bytes <sup>1</sup>	1.75 <sup>1</sup>

*Table continues on the next page...*

Table 7. MKE1xZ dedicated functions (continued)

File	Suitable function	Functions size	Functions duration [µs]
	FS_TSI_InputCheckNONStimulated()	196 bytes <sup>1</sup>	1.26 <sup>1</sup>
	FS_TSI_InputCheckStimulated()	294 bytes <sup>1</sup>	0.92 <sup>1</sup>
iec60730b_flash.S	Function are described in dedicated chapter	See the function dedicated chapter	
iec60730b_pc.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_ram.S	Function are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_reg.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_Stack.S	Function are common for all CM0+ devices	See the function dedicated chapter	

## 1.2.7 MKLxx dedicated functions

The following table shows the list of functions dedicated for the MKLxx device family.

Table 8. MKLxx dedicated functions

File	Suitable function	Functions size	Functions duration [µs]
iec60730b_aio.c	FS_AIO_InputInit()	16 bytes <sup>1</sup>	0.44 <sup>1</sup>
	FS_AIO_InputTrigger()	10 bytes <sup>1</sup>	0.31 <sup>1</sup>
	FS_AIO_InputSet()	42 bytes <sup>1</sup>	0.43 <sup>1</sup>
	FS_AIO_InputCheck()	20 bytes <sup>3</sup>	0.75 <sup>3</sup>
iec60730b_clock.c	FS_CLK_Check()	38 bytes <sup>1</sup>	0.60 <sup>1</sup>
	FS_CLK_Init()	8 bytes <sup>1</sup>	0.24 <sup>1</sup>
	FS_CLK_LPTMR()	32 bytes <sup>1</sup>	0.55 <sup>1</sup>
iec60730b_dio.c	FS_DIO_Output()	112 bytes <sup>1</sup>	24.39 <sup>1</sup>
	FS_DIO_Input()	60 bytes <sup>4</sup>	5.10 <sup>4</sup>
	FS_DIO_InputExt()	206 bytes <sup>1</sup>	2.97 <sup>1</sup>
iec60730b_dio_ext.c	FS_DIO_ShortToSupplySet()	166 bytes <sup>1</sup>	2.94 <sup>1</sup>
	FS_DIO_ShortToAdjSet()	250 bytes <sup>1</sup>	4.14 <sup>1</sup>
	FS_DIO_ShortToSupplySet()	166 bytes <sup>1</sup>	2.94 <sup>1</sup>
iec60730b_wdg.c	FS_WDOG_Setup_COP()	38 bytes <sup>6</sup>	-
	FS_WDOG_Check()	76 bytes <sup>3</sup>	1.29 <sup>3</sup>
iec60730b_flash.S	Function are described in dedicated chapter	See the function dedicated chapter	
iec60730b_pc.S	Functions are common for all CM0+ devices	See the function dedicated chapter	

*Table continues on the next page...*

Table 8. MKLxx dedicated functions (continued)

File	Suitable function	Functions size	Functions duration [ $\mu$ s]
iec60730b_ram.S	Function are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_reg.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_Stack.S	Function are common for all CM0+ devices	See the function dedicated chapter	

## 1.2.8 MKV1x dedicated functions

The following table shows the list of functions dedicated for the MKV1x device family.

Table 9. MKV1x dedicated functions

File	Suitable function	Functions size	Functions duration [ $\mu$ s]
iec60730b_aio.c	FS_AIO_InputInit()	16 bytes <sup>1</sup>	0.44 <sup>1</sup>
	FS_AIO_InputTrigger()	10 bytes <sup>1</sup>	0.31 <sup>1</sup>
	FS_AIO_InputSet()	42 bytes <sup>1</sup>	0.43 <sup>1</sup>
	FS_AIO_InputCheck()	20 bytes <sup>3</sup>	0.75 <sup>3</sup>
iec60730b_clock.c	FS_CLK_Check()	38 bytes <sup>1</sup>	0.60 <sup>1</sup>
	FS_CLK_Init()	8 bytes <sup>1</sup>	0.24 <sup>1</sup>
	FS_CLK_LPTMR()	32 bytes <sup>1</sup>	0.55 <sup>1</sup>
iec60730b_dio.c	FS_DIO_Output()	112 bytes <sup>1</sup>	24.4 <sup>1</sup>
	FS_DIO_Input()	60 bytes <sup>4</sup>	5.10 <sup>4</sup>
iec60730b_dio_ext.c	FS_DIO_InputExt()	206 bytes <sup>1</sup>	2.97 <sup>1</sup>
	FS_DIO_ShortToSupplySet()	166 bytes <sup>1</sup>	2.94 <sup>1</sup>
	FS_DIO_ShortToAdjSet()	250 bytes <sup>1</sup>	4.14 <sup>1</sup>
iec60730b_wdg.c	FS_WDOG_Setup()	38 bytes <sup>3</sup>	-
	FS_WDOG_Check()	76 bytes <sup>3</sup>	1.29 <sup>3</sup>
iec60730b_flash.S	Function are described in dedicated chapter	See the function dedicated chapter	
iec60730b_pc.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_ram.S	Function are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_reg.S	Functions are common for all CM0+ devices	See the function dedicated chapter	
iec60730b_Stack.S	Function are common for all CM0+ devices	See the function dedicated chapter	

## 1.2.9 Functions performance measurement

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

1. The function parameter was measured in the IAR 8.40.1. IDE on MKE15z with a clock frequency of 72 MHz.
2. The function parameter was measured in the IAR 8.40.1. IDE on LPC845 with a clock frequency of 30 MHz.
3. The function parameter was measured in the IAR 8.40.1. IDE on MKV10z with a clock frequency of 75 MHz.
4. The function parameter was measured in the IAR 8.40.1. IDE on MKE02z with a clock frequency of 10 MHz.
5. The function parameter was measured in the IAR 8.40.1. IDE on LPC824 with a clock frequency of 30 MHz.
6. The function parameter was measured in the IAR 8.40.1. IDE on MKL26z with a clock frequency of 48 MHz.
7. The function parameter was measured in the IAR 8.40.1. IDE on MK32W with a clock frequency of 48 MHz.

# Chapter 2

## Analog Input/Output (IO) test

The analog IO test procedure performs the 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 if an analog IO error occurs. 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 occurs. 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 three reference values: reference high, reference low, and bandgap voltage. See the device specification document to set up the correct values. The block diagram for the analog IO test is shown in the following figure:

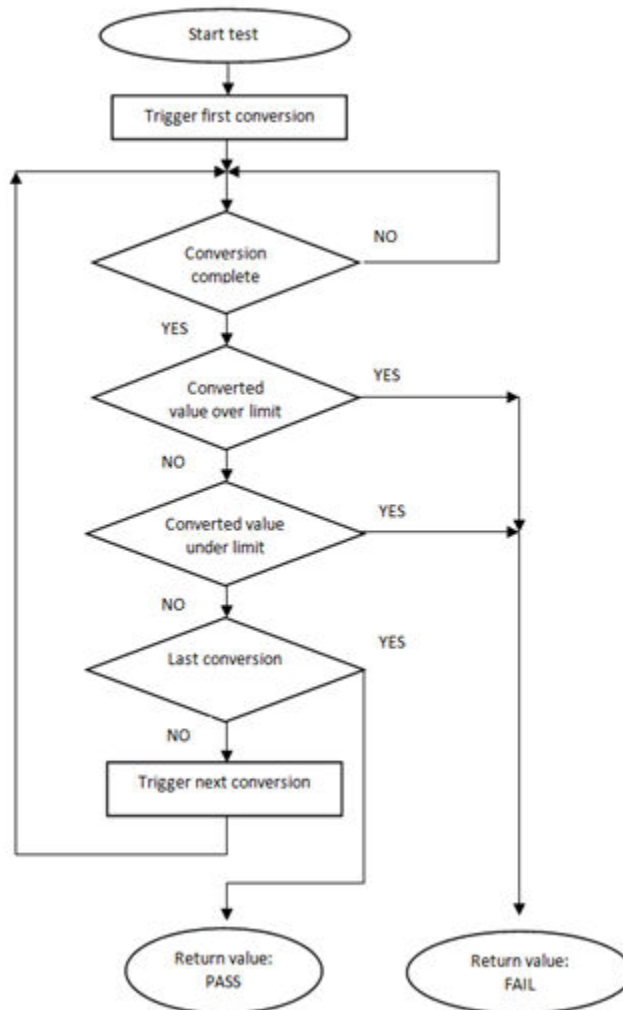


Figure 1. Block diagram for analog input test

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

The performed overload test fulfils 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. 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\_aio.c* file and written as "C" functions. The header file with the function prototypes is *iec60730b\_aio.h*. *iec60730b.h* is the common header file for the safety library.

The following functions are called to test the analog input:

- *FS\_AIO\_InputInit()* / *FS\_AIO\_InputInit\_K3SW()* / *FS\_AIO\_InputInit\_CYCLIC()* / *FS\_AIO\_InputInit\_LPC\_ADC16()* / *FS\_AIO\_InputInit\_LPC\_ADC12()* /
- *FS\_AIO\_InputTrigger()*
- *FS\_AIO\_InputSet()* / *FS\_AIO\_InputSet\_K3S()* / *FS\_AIO\_InputSet\_CYCLIC()* / *FS\_AIO\_InputSet\_LPC8XX()* / *FS\_AIO\_InputSet\_LPC55SXX()*
- *FS\_AIO\_InputCheck()* / *FS\_AIO\_InputCheck\_KE()* / *FS\_AIO\_InputCheck\_K3S()* / *FS\_AIO\_InputCheck\_CYCLIC()* / *FS\_AIO\_InputCheck\_LPC8XX()* / *FS\_AIO\_InputCheck\_LPC55SXX()*

The analog input test is based on a conversion of three analog inputs with known voltage values and checks if the converted values fit into the specified limits. Normally, the limits should be about 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.

Throughout all supported devices, the ADC module has a slightly different arrangement of the registers that are involved in the test. Therefore, a standalone check function is created for the ADC module. See [Core self-test library – source code version](#) for the function dedicated for your device.

The following is an example of a function call:

### Configuration of parameters

```
#define TESTED_ADC ADC0
#define ADC_RESOLUTION 12
#define ADC_MAX ((1<<(ADC_RESOLUTION))-1)
#define ADC_REFERENCE 3.06
#define ADC_BANDGAP_LEVEL 1.7
#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_INIT1
```

```

{
  {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, 27}

```

**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, (unsigned long*)TESTED_ADC);
  switch(psSafetyCommon->AIO_test_result)
  {
    case FS_AIO_START:
      FS_AIO_InputSet(&aio_Str, (unsigned long*)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 analog input test to start the execution of the test (sets state FS\_AIO\_START for pObj).

### **Function prototype:**

```
void FS_AIO_InputTrigger(fs_aio_test_t *pObj);
```

### **Function inputs:**

*\*pObj* - The input argument is the pointer to the analog test instance.

### **Function output:**

*void*

### **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 one 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 input argument is the pointer to the analog test instance.

*\*pLimits* - The input argument is the pointer to the array of limits used in the test.

*\*pInputs* - The input argument is the pointer to the array of input numbers used in the test.

*\*cntMax* - The input argument is the size of the input and the limits arrays.

### **Function output:**

*void*

### **Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 2.2.3 FS\_AIO\_InputInit\_CYCLIC()

This function initializes an 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 input argument is the pointer to the analog test instance.

*\*pLimits* - The input argument is the pointer to the array of limits used in the test.

*\*pInputs* - The input argument is the pointer to the array of input numbers used in the test.

*\*pSamples* - The input argument is the pointer to the array of sample numbers used in the test.

Analog Input/Output (IO) test

*cntMax* - The input argument is the size of the input and the limits arrays.

**Function output:**

*void*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 2.2.4 FS\_AIO\_InputInit\_LPC\_ADC16()

This function initializes an instance of the analog input test.

**Function prototype:**

```
void FS_AIO_InputInit_LPC_ADC16(fs_aio_test_t *pObj, fs_aio_limits_t *pLimits, uint8_t *pInputs, uint8_t cntMax, uint8_t sequence, uint8_t fifo);
```

**Function inputs:**

*\*pObj* - The input argument is the pointer to the analog test instance.

*\*pLimits* - The input argument is the pointer to the array of limits used in the test.

*\*pInputs* - The input argument is the pointer to the array of input numbers used in the test.

*cntMax* - The input argument is the size of the input and the limits arrays.

*sequence* - No effect, just because of compatibility.

*fifo* - The index of the used FIFO for result.

**Function output:**

*void*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 2.2.5 FS\_AIO\_InputInit\_LPC\_ADC12()

This function initializes an instance of the analog input test.

**Function prototype:**

```
void FS_AIO_InputInit_LPC_ADC12(fs_aio_test_t *pObj, fs_aio_limits_t *pLimits, uint8_t *pInputs, uint8_t cntMax, uint8_t sequence);
```

**Function inputs:**

*\*pObj* - The input argument is the pointer to the analog test instance.

*\*pLimits* - The input argument is the pointer to the array of limits used in the test.

*\*pInputs* - The input argument is a pointer to the array of input numbers used in the test.

*cntMax* - The input argument is the size of the input and the limits arrays.

*sequence* - The input argument is the index of the sequence used.

**Function output:**

*void*

**Function performance:**

The Information about the function performance is in [Core self-test library – source code version](#).

## 2.2.6 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. The state is changed to FS\_AIO\_PROGRESS. 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, uint32_t *pAdc);
```

### **Function inputs:**

\*pObj- The input argument is the pointer to the analog test instance.

\*pAdc- The input argument is the pointer to the analog converter.

### **Function output:**

```
typedef uint32_t FS_RESULT;
```

- FS\_AIO\_PROGRESS - A required return value. It means that the input is set.

If any other value is returned, the function has no effect.

### **Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 2.2.7 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. The state is changed to FS\_AIO\_PROGRESS. 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 input argument is the pointer to the analog test instance.

\*pAdc- The input argument is the pointer to the analog converter.

### **Function output:**

```
typedef uint32_t FS_RESULT;
```

- FS\_AIO\_PROGRESS - The required return value. It means that the input is set.

If any other value is returned, the function has no effect.

### **Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 2.2.8 FS\_AIO\_InputSet\_K3S()

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. The state is changed to FS\_AIO\_PROGRESS. This function can be called when the ADC module is idle and ready for the next conversion.

### **Function prototype:**

```
FS_RESULT FS_AIO_InputSet_K3S(fs_aio_test_t *pObj, uint32_t *pAdc);
```

### **Function inputs:**

\*pObj- The input argument is the pointer to the analog test instance.

Analog Input/Output (IO) test

*\*pAdc* - The input argument is the pointer to the analog converter.

**Function output:**

*typedef uint32\_t FS\_RESULT;*

- *FS\_AIO\_PROGRESS* - The required return value. It means that the input is set.

If any other value is returned, the function has no effect.

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 2.2.9 FS\_AIO\_InputSet\_LPC8XX()

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. The state is changed to *FS\_AIO\_PROGRESS*. This function can be called when the ADC module is idle and ready for the next conversion.

**Function prototype:**

*FS\_RESULT FS\_AIO\_InputSet\_LPC8XX(fs\_aio\_test\_t \*pObj, uint32\_t \*pAdc);*

**Function inputs:**

*\*pObj* - The input argument is the pointer to the analog test instance.

*\*pAdc* - The input argument is the pointer to the analog converter.

**Function output:**

*typedef uint32\_t FS\_RESULT;*

- *FS\_AIO\_PROGRESS* - The required return value. It means that the input is set.

If any other value is returned, the function has no effect.

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 2.2.10 FS\_AIO\_InputSet\_LPC55SXX()

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. The state is changed to *FS\_AIO\_PROGRESS*. This function can be called when the ADC module is idle and ready for the next conversion.

**Function prototype:**

*FS\_RESULT FS\_AIO\_InputSet\_LPC55SXX(fs\_aio\_test\_t \*pObj, uint32\_t \*pAdc);*

**Function inputs:**

*\*pObj* - The input argument is the pointer to the analog test instance.

*\*pAdc* - The input argument is the pointer to the analog converter.

**Function output:**

*typedef uint32\_t FS\_RESULT;*

- *FS\_AIO\_PROGRESS* - The required return value. It means that the input was set.

If any other value is returned, the function has no effect.

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 2.2.11 FS\_AIO\_InputCheck()

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. This function reads the converted value only if `pObj->state == FS_AIO_PROGRESS`. 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, uint32_t *pAdc);
```

### Function inputs:

*\*pObj*- The input argument is the pointer to the analog test instance.

*\*pAdc*- The input argument is the pointer to the analog converter.

### Function output:

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*- A successful execution of the test (all channels tested).
- *FS\_FAIL\_AIO*- The converted value does not fit into the limits.
- *FS\_AIO\_START*- A successful conversion and a setup input for the next conversion.
- *FS\_AIO\_PROGRESS*- The input is not converted yet.
- *FS\_AIO\_INIT*- The function has no effect.

### Function performance:

The information about the function performance is in [Core self-test library – source code version](#).

## 2.2.12 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. This function reads the converted value only if `pObj->state == FS_AIO_PROGRESS`. 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 input argument is the pointer to the analog test instance.

*\*pAdc*- The input argument is the pointer to the analog converter.

### Function output:

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*- A successful execution of the test (all channels tested).
- *FS\_FAIL\_AIO*- The converted value does not fit into the limits.
- *FS\_AIO\_START*- A successful conversion and a setup input for the next conversion.
- *FS\_AIO\_PROGRESS*- The input is not converted yet.
- *FS\_AIO\_INIT*- The function has no effect.

### Function performance:

The information about the function performance is in [Core self-test library – source code version](#).

## 2.2.13 FS\_AIO\_InputCheck\_K3S()

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. This function reads the converted value only if `pObj->state == FS_AIO_PROGRESS`. The test is finished when this function reports `FS_AIO_PASS` or `FS_AIO_FAIL`. It is dedicated for 12-bit low-power ADCs.

### **Function prototype:**

```
FS_RESULT FS_AIO_InputCheck_K3S(fs_aio_test_t *pObj, uint32_t *pAdc, uint32_t resFifo);
```

### **Function inputs:**

*\*pObj* - The input argument is the pointer to the analog test instance.

*\*pAdc* - The input argument is the pointer to the analog converter.

*resFifo* - A 32-bit value of the RESFIFO register.

### **Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS* - A successful execution of the test (all channels tested).
- *FS\_FAIL\_AIO* - The converted value does not fit into the limits.
- *FS\_AIO\_START* - A successful conversion and a setup input for the next conversion.
- *FS\_AIO\_PROGRESS* - The input is not converted yet.
- *FS\_AIO\_INIT* - The function has no effect.

### **Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 2.2.14 FS\_AIO\_InputCheck\_KE()

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. This function reads the converted value only if `pObj->state == FS_AIO_PROGRESS`. The test is finished, when this function reports `FS_AIO_PASS` or `FS_AIO_FAIL`.

### **Function prototype:**

```
FS_RESULT FS_AIO_InputCheck_KE(fs_aio_test_t *pObj, uint32_t *pAdc);
```

### **Function inputs:**

*\*pObj* - The input argument is the pointer to the analog test instance.

*\*pAdc* - The input argument is the pointer to the analog converter.

### **Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS* - A successful execution of the test (all channels tested).
- *FS\_FAIL\_AIO* - The converted value does not fit into the limits.
- *FS\_AIO\_START* - A successful conversion and a setup input for the next conversion.
- *FS\_AIO\_PROGRESS* - The input is not converted yet.
- *FS\_AIO\_INIT* - The function has no effect.

### **Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 2.2.15 FS\_AIO\_InputCheck\_LPC55SXX()

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. This function reads the converted value only if `pObj->state == FS_AIO_PROGRESS`. The test is finished when this function reports `FS_AIO_PASS` or `FS_AIO_FAIL`.

### Function prototype:

```
FS_RESULT FS_AIO_InputCheck_LPC55SXX(fs_aio_test_t* pObj, uint32_t *pAdc);
```

### Function inputs:

*\*pObj*- The input argument is the pointer to the analog test instance.

*\*pAdc*- The input argument is the pointer to the analog converter.

### Function output:

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS* - A successful execution of the test (all channels tested).
- *FS\_FAIL\_AIO* - The converted value does not fit into the limits.
- *FS\_AIO\_START* - A successful conversion and a setup input for the next conversion.
- *FS\_AIO\_PROGRESS* - The input is not converted yet.
- *FS\_AIO\_INIT* - The function has no effect.

### Function performance:

The Information about the function performance is in [Core self-test library – source code version](#).

## 2.2.16 FS\_AIO\_InputCheck\_LPC8XX()

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. This function reads the converted value only if `pObj->state == FS_AIO_PROGRESS`. The test is finished when this function reports `FS_AIO_PASS` or `FS_AIO_FAIL`.

### Function prototype:

```
FS_RESULT FS_AIO_InputCheck_LPC8XX(fs_aio_test_t* pObj, uint32_t *pAdc);
```

### Function inputs:

*\*pObj*- The input argument is the pointer to the analog test instance.

*\*pAdc*- The input argument is the pointer to the analog converter.

### Function output:

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS* - A successful execution of the test (all channels tested).
- *FS\_FAIL\_AIO* - The converted value does not fit into the limits.
- *FS\_AIO\_START* - A successful conversion and a setup input for the next conversion.
- *FS\_AIO\_PROGRESS* - The input is not converted yet.
- *FS\_AIO\_INIT* - The function has no effect.

### Function performance:

The information about the function performance is in [Core self-test library – source code version](#).

# Chapter 3

## Clock test

The clock test procedure tests the clock frequency of the processor for the "stuck-at" condition. The clock 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 case of a clock fault. Assess the return value of the test function. If it is equal to the FAIL return, then a jump into the safety error handling function should 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 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 device supported by the library has many timer/counter modules. See [Core self-test library – source code version](#) for a function suitable for your device.

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

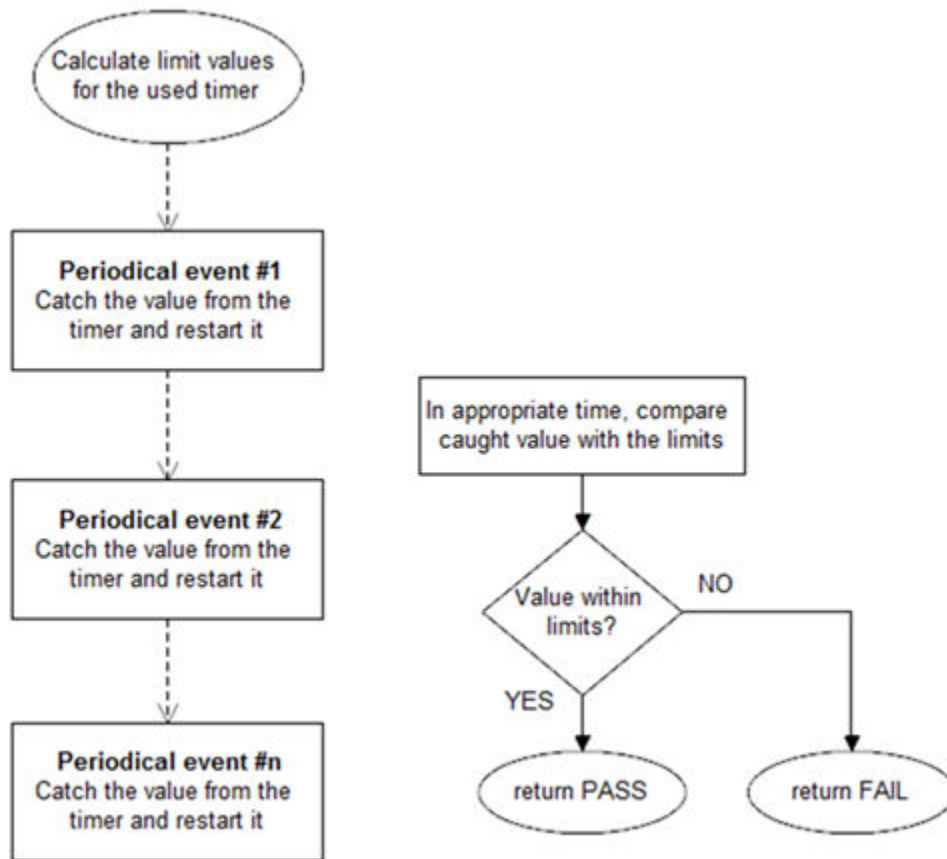


Figure 2. Block diagram for clock test

### 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 11. 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\_clock.c* file and written as "C" functions. The header file with the function prototypes is *iec60730b\_clock.h*. The common library header file is *iec60730b.h*.

The following functions are called to test the clock frequency:

- FS\_CLK\_Init()
- FS\_CLK\_LPTMR() / FS\_CLK\_RTC() / FS\_CLK\_GPT() / FS\_CLK\_WKT\_LPC() / FS\_CLK\_TIMER\_LPC()
- FS\_CLK\_Check()

Configure the reference timer, choose an appropriate periodical event and calculate the limit values. Declare the 32-bit global variable for storing the content of the timer counter register. 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, normally before the while() loop. The FS\_CLK\_LPTMR() (to choose the dedicated function for your device, see [Core self-test library – source code version](#)) function is then called within a periodic event. The FS\_CLK\_Check() function for evaluation can be called at any given time. When the test is in the initialization phase, the check function returns the "in progress" value. If the captured value from the 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 the test implementation is as follows:

```
#include "iec60730b.h"

FS_RESULT st;

unsigned long clockTestContext;

#define ISR_FREQUENCY (100)

#define CLOCK_TEST_TOLERANCE (10)

#define REF_TIMER_CLOCK_FREQUENCY (32e03l)

RTC_SC = RTC_SC_RTCLKS(2)|RTC_SC_RTCPS(1);

SysTick->VAL = 0x0;

SysTick->LOAD = 100e6*0.01;

SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_ENABLE_Msk|
SysTick_CTRL_TICKINT_Msk;

SysTick->VAL = 0x0;

FS_CLK_Init(&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((uint32_t*)RTC_BASE_PTR, &clockTestContext);
```

```
}

```

### 3.2.1 FS\_CLK\_Init()

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

**Function prototype:**

```
void FS_CLK_Init(uint32_t *pTestContext);
```

**Function inputs:**

*\*pTestContext* - The pointer to the variable that holds the captured timer value.

**Function output:**

*void*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

### 3.2.2 FS\_CLK\_Check()

This function handles the clock test. It evaluates the captured value stored in the testContext variable with predefined limits. Until the first execution of the respective Lsr function, the check function returns FS\_CLK\_PROGRESS.

**Function prototype:**

```
FS_RESULT FS_CLK_Check(uint32_t testContext, uint32_t limitLow, uint32_t limitHigh);
```

**Function inputs:**

*testContext* - The captured value of the timer.

*limitLow* - The low limit.

*limitHigh* - The high limit.

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS* - The testContext fits into the limits.
- *FS\_FAIL\_CLK* - The testContext value does not fit into the limits.
- *FS\_CLK\_PROGRESS* - The reference counter value is not read yet.

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

### 3.2.3 FS\_CLK\_LPTMR()

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

**Function prototype:**

```
void FS_CLK_LPTMR(uint32_t *pSafetyTmr, uint32_t *pTestContext);
```

**Function inputs:**

*\*pSafetyTmr* - The timer module address.

*\*pTestContext* - The pointer to the variable that holds the captured timer value.

**Function output:***void***Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

### 3.2.4 FS\_CLK\_CTIMER\_LPC()

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

**Function prototype:**

```
void FS_CLK_CTIMER_LPC(uint32_t *pSafetyTmr, uint32_t *pTestContext);
```

**Function inputs:**

*\*pSafetyTmr* - The timer module address.

*\*pTestContext* - The pointer to the variable that holds the captured timer value.

**Function output:***void***Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

### 3.2.5 FS\_CLK\_GPT()

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

**Function prototype:**

```
void FS_CLK_GPT(uint32_t *pSafetyTmr, uint32_t *pTestContext);
```

**Function inputs:**

*\*pSafetyTmr* - The timer module address.

*\*pTestContext* - The pointer to the variable that holds the captured timer value.

**Function output:***void***Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

### 3.2.6 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(uint32_t *pSafetyTmr, uint32_t *pTestContext);
```

**Function inputs:**

*\*pSafetyTmr* - The timer module address.

*\*pTestContext* - The pointer to the variable that holds the captured timer value.

Clock test

**Function output:**

*void*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 3.2.7 FS\_CLK\_WKT\_LPC()

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

**Function prototype:**

```
void FS_CLK_WKT_LPC(uint32_t *pSafetyTmr, uint32_t *pTestContext, uint32_t startValue);
```

**Function inputs:**

*\*pSafetyTmr* - The timer module address.

*\*pTestContext* - The pointer to the variable that holds the captured timer value.

*startValue* - The start value to decrease the WKT counter.

**Function output:**

*void*

**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 the plausibility check of the processor's digital IO interface.

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 and if it is equal to the FAIL return, the 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 are designed to check the digital input and output functionality and short circuit conditions between the tested pin and the supply voltage, ground, or optional adjacent pin. The execution of the DIO tests must be adapted to the final application. Be careful with the hardware connections and design. Be sure about which functions can be applied to a respective pin. In most of cases, the tested (and sometimes also auxiliary) pin must be reconfigured during the application run. When testing the digital output, reserve enough time between the test arrangement and the reading of results.

### 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 12. 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

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

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

- FS\_DIO\_Input()
- FS\_DIO\_Output() / FS\_DIO\_Output\_RT() / FS\_DIO\_Output\_LPC()
- FS\_DIO\_InputExt() / FS\_DIO\_InputExt\_RT() / FS\_DIO\_InputExt\_LPC()
- FS\_DIO\_ShortToSupplySet() / FS\_DIO\_ShortToSupplySet\_RT() / FS\_DIO\_ShortToSupplySet\_LPC()
- FS\_DIO\_ShortToAdjSet() / FS\_DIO\_ShortToAdjSet\_RT() / FS\_DIO\_ShortToAdjSet\_LPC()

The pointer to the *fs\_dio\_test\_t* structure type is a parameter of each function. The structure is defined in the *iec60730b\_dio.h* file.

```
typedef struct
{
    uint32_t pcr; /* Pin control register */
    uint32_t pddr; /* Port data direction register */
    uint32_t pdor; /* Port data output register */
} fs_dio_backup_t;
```

```

typedef struct
{
    uint32_t gpio;
    fs_dio_backup_t pcr;
    uint8_t pinNum;
    uint8_t pinDir;
    uint8_t pinMux;
    fs_dio_backup_t sTestedPinBackup;
} fs_dio_test_t;

```

These variables must be initialized before calling a test function. The following is an example of initialization:

```

fs_dio_test_t dio_safety_test_item_0 =
{
    .gpio = FS_DIO_PORTE,
    .pinNum = 24,
    .pinDir = PIN_DIRECTION_IN,
    .pinMux = PIN_MUX_GPIO,
};
fs_dio_test_t dio_safety_test_item_1 =
{
    .gpio = FS_DIO_PORTA,
    .pinNum = 2,
    .pinDir = PIN_DIRECTION_OUT,
    .pinMux = PIN_MUX_GPIO,
};
fs_dio_test_t *dio_safety_test_items[] = { &dio_safety_test_item_0, &dio_safety_test_item_1, 0 };

if (dio_safety_test_item_0 .gpio == FS_DIO_PORTE)
    dio_safety_test_item_0 .pcr = FS_DIO_PORTE_PCR;

if (dio_safety_test_item_1 .gpio == FS_DIO_PORTA)
    dio_safety_test_item_1 .pcr = FS_DIO_PORTA_PCR;

```

## 4.2.1 FS\_DIO\_Input()

This function executes the digital input test. The test tests one digital pin. The pin is tested according to the block diagram in the following figure:

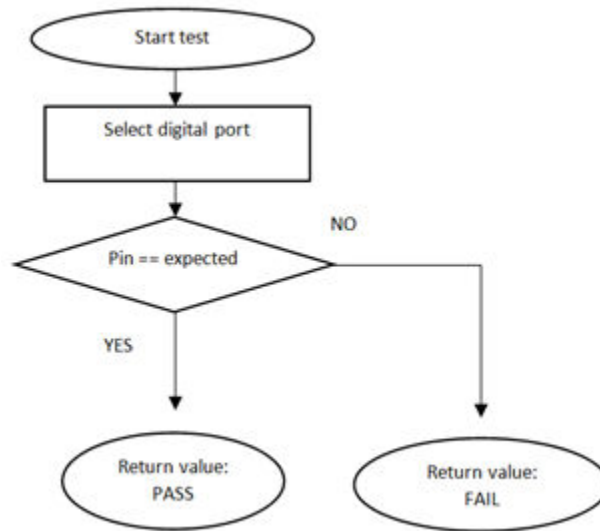


Figure 3. Block diagram for digital input test

**Function prototype:**

*FS\_RESULT FS\_DIO\_Input(fs\_dio\_test\_t \*pTestedPin, bool\_t expectedValue);*

**Function inputs:**

*\*pTestedPin* - The pointer to the tested pin struct.

*expectedValue* - The expected input value. Adjust this parameter correctly.

**Function output:**

*typedef uint32\_t FS\_RESULT;*

- *FS\_PASS*
- *FS\_FAIL\_DIO*

**Example of function call:**

*fs\_dio\_input\_test\_result = FS\_DIO\_Input(&dio\_safety\_test\_items[0], DIO\_EXPECTED\_VALUE);*

**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 a GPIO with input direction.

## 4.2.2 FS\_DIO\_InputExt()

This is a modified version of the previously mentioned digital input test. It cannot be used with MKE0x devices. This version is 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 at the time of the test. The logical level can result from the actual configuration in the application or it can be initialized for the test (if possible). The block diagram of the *FS\_DIO\_InputExt()* function is shown in the following figure. Two function input parameters are related to an adjacent pin. For a simple input test functionality, these parameters are not important. Enter the same inputs as for the tested pin (recommended). See the example code.

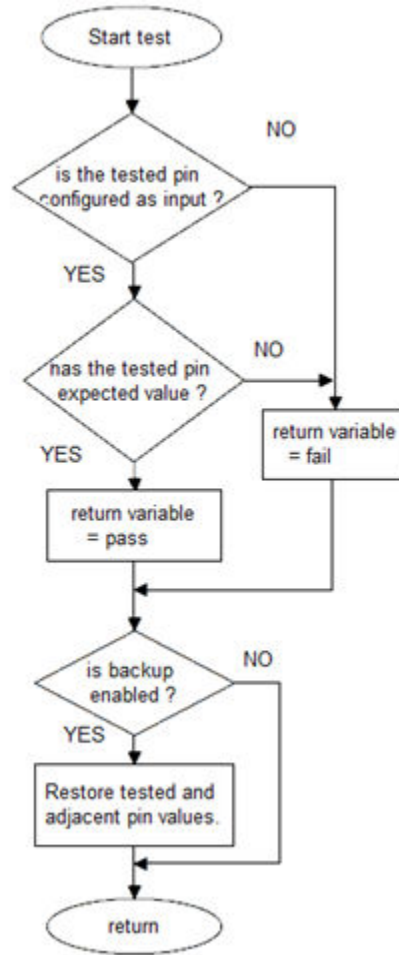


Figure 4. Extended digital input test

**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 the tested pin struct.

*\*pAdjPin* - The pointer to the adjacent pin struct.

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

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

**Function output:**

*typedef uint32\_t FS\_RESULT;*

- *FS\_PASS*
- *FS\_FAIL\_DIO*

**Example of function call:**

*fs\_dio\_input\_test\_result = FS\_DIO\_InputExt(&dio\_safety\_test\_item\_0, &dio\_safety\_test\_item\_0, DIO\_EXPECTED\_VALUE, BACKUP\_ENABLE);*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).



**Calling restrictions:**

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

**4.2.3 FS\_DIO\_InputExt\_RT()**

This is a modified version of the previously mentioned digital input test. Use this version as a get function for the "short-to" tests. Apply the function to the pin that is already configured as a GPIO input and you know what logical level is expected at the time of the test. The logical level results from the actual configuration in the application or it is initialized for the test (if possible). The block diagram of the *FS\_DIO\_InputExt\_RT()* function is shown in the following figure. Two function input parameters are related to an adjacent pin. For simple input test functionality, these parameters are not important. Enter the same inputs as for the tested pin (recommended). See the example code.

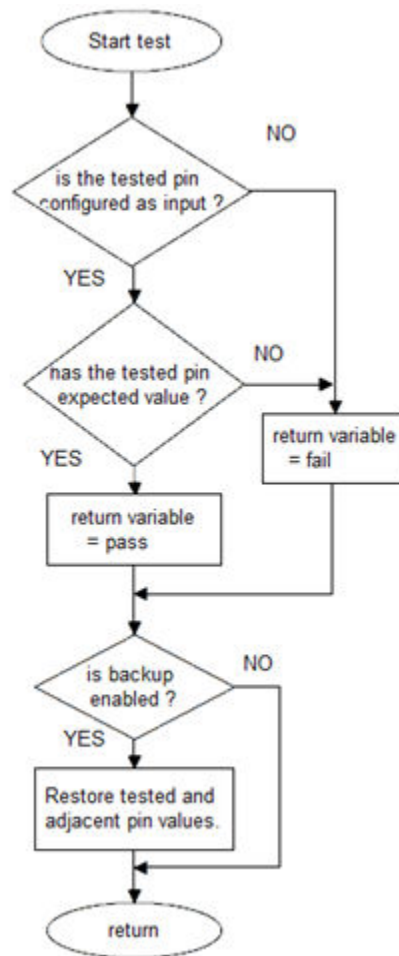


Figure 5. Extended digital input test for RT

**Function prototype:**

```
FS_RESULT FS_DIO_InputExt_RT(fs_dio_test_rt_t *pTestedPin, fs_dio_test_rt_t *pAdjPin, bool_t testedPinValue, bool_t backupEnable);
```

**Function inputs:**

\*pTestedPin - The pointer to the tested pin struct.

\*pAdjPin - The pointer to the adjacent pin struct.

Digital input/output test

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

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

**Function output:**

*typedef uint32\_t FS\_RESULT;*

- *FS\_PASS*
- *FS\_FAIL\_DIO*

**Example of function call:**

*fs\_dio\_input\_test\_result = FS\_DIO\_InputExt\_RT(&dio\_safety\_test\_item\_0, &dio\_safety\_test\_item\_0, DIO\_EXPECTED\_VALUE, BACKUP\_ENABLE);*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The function can be used only for the i.MXRT devices. Configure the tested pin as a GPIO input before calling the function. Even if no adjacent pins are involved in the test, specify the *AdjacentPin* parameter. It is recommended is enter the same input as for *TestedPin*.

## 4.2.4 FS\_DIO\_InputExt\_LPC()

This is a modified version of the previously mentioned digital input test. This version is used as a get function for the "short-to" tests. Apply the function to the pin that is already configured as a 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 of the *FS\_DIO\_InputExt\_LPC()* function is shown in the following figure. Two function input parameters are related to an adjacent pin. For simple input test functionality, these parameters are not important. Enter the same inputs as for the tested pin (recommended). See the example code.

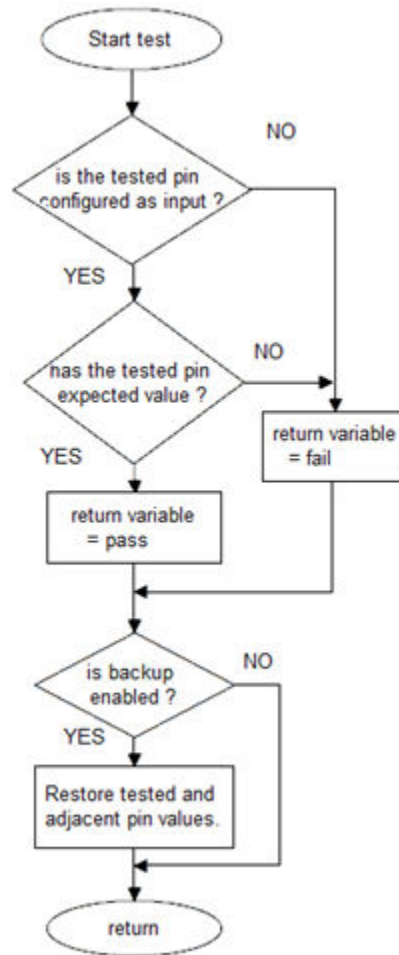


Figure 6. Extended digital input test for LPC devices

**Function prototype:**

```
FS_RESULT FS_DIO_InputExt_LPC(fs_dio_test_lpc_t *pTestedPin, fs_dio_test_lpc_t *pAdjPin, bool_t testedPinValue, bool_t backupEnable);
```

**Function inputs:**

*\*pTestedPin* - The pointer to the tested pin struct.

*\*pAdjPin* - The pointer to the adjacent pin struct.

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

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

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*
- *FS\_FAIL\_DIO*

**Example of function call:**

```
fs_dio_input_test_result = FS_DIO_InputExt_LPC(&dio_safety_test_item_0, &dio_safety_test_item_0, DIO_EXPECTED_VALUE, BACKUP_ENABLE);
```

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

Configure the tested pin as a GPIO input before the function call. Even if no adjacent pins are involved in the test, specify the AdjacentPin parameter. It is recommended to enter the same input as for the TestedPin.

### 4.2.5 FS\_DIO\_Output()

The digital output test 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.

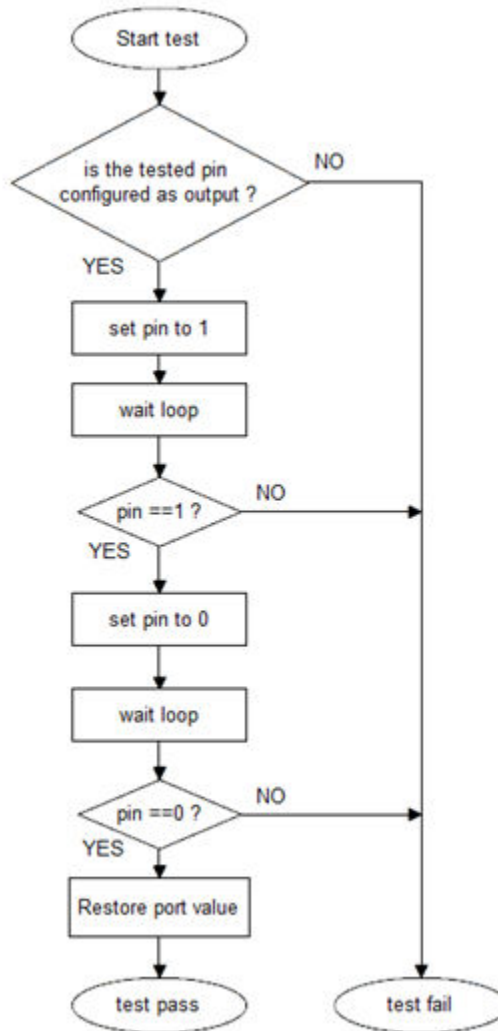


Figure 7. Block diagram for digital output test

**Function prototype:**

*FS\_RESULT FS\_DIO\_Output(fs\_dio\_test\_t \*pTestedPin, uint32\_t delay);*

**Function inputs:**

*\*pTestedPin* - The pointer to the tested pin struct.

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

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- `FS_PASS`
- `FS_FAIL_DIO`

**Example of function call:**

```
fs_dio_output_test_result = FS_DIO_Output(&dio_safety_test_items[1], DIO_WAIT_CYCLE);
```

**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. Define an appropriate delay for proper functionality.

## 4.2.6 FS\_DIO\_Output\_RT()

This test tests the digital output functionality of the pin. The principle of this 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.

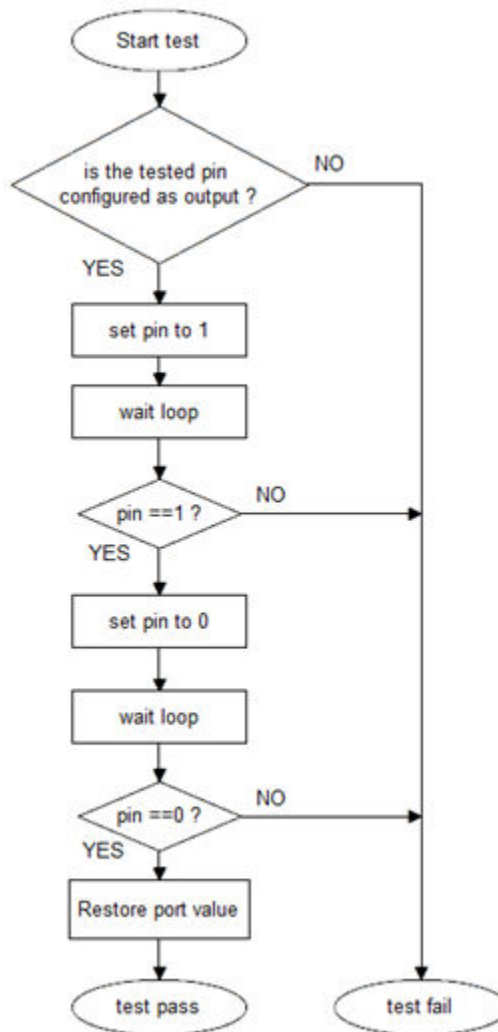


Figure 8. Block diagram for digital output test

Digital input/output test

**Function prototype:**

```
FS_RESULT FS_DIO_Output_RT(fs_dio_test_rt_t *pTestedPin, uint32_t delay);
```

**Function inputs:**

*\*pTestedPin* - The pointer to the tested pin struct.

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

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*
- *FS\_FAIL\_DIO*

**Example of function call:**

```
fs_dio_output_test_result = FS_DIO_Output_RT(&dio_safety_test_items[1], DIO_WAIT_CYCLE);
```

**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 a digital output. Define an appropriate delay for proper functionality.

## 4.2.7 FS\_DIO\_Output\_LPC()

This test 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. A suitable delay parameter must be entered. 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.

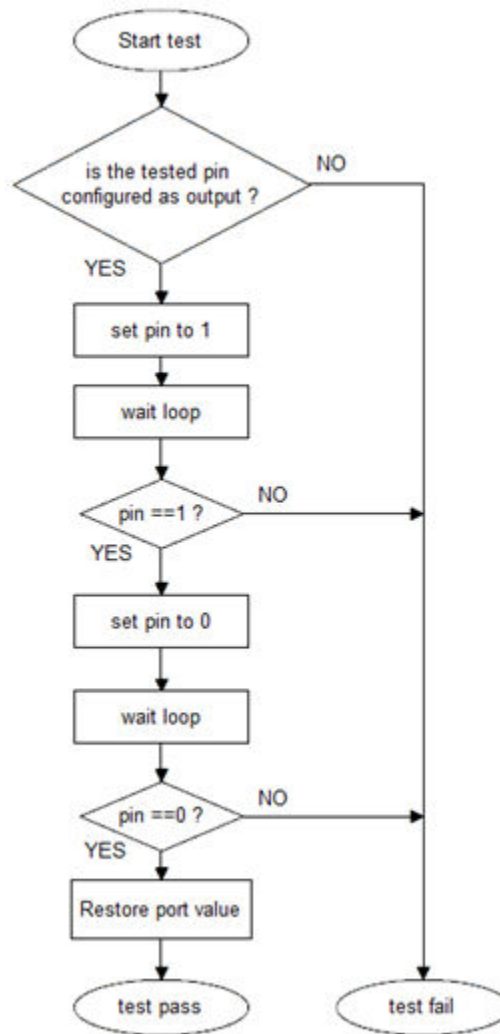


Figure 9. Block diagram for digital output test

**Function prototype:**

```
FS_RESULT FS_DIO_Output_LPC(fs_dio_test_ipc_t *pTestedPin, uint32_t delay);
```

**Function inputs:**

*\*pTestedPin* - The pointer to the tested pin struct.

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

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*
- *FS\_FAIL\_DIO*

**Example of function call:**

```
fs_dio_output_test_result = FS_DIO_Output_LPC(&dio_safety_test_items[1], DIO_WAIT_CYCLE);
```

**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 a digital output. Define an appropriate delay for proper functionality.

## 4.2.8 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-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 section. Specify the tested pin and the adjacent pin for the input test function.

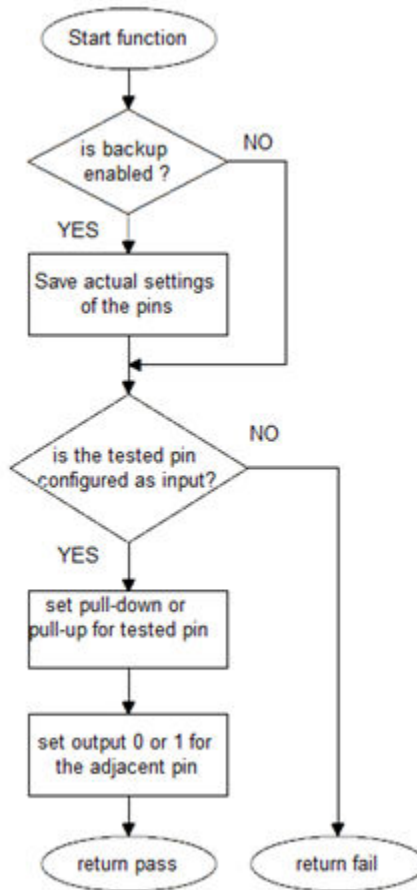


Figure 10. Block diagram of FS\_DIO\_ShortToAdjSet() function

### 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 the tested pin struct.

*\*pAdjPin* - The pointer to the adjacent pin struct.

*testedPinValue* - The value to be set on the tested pin.

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

### Function output:

```
typedef uint32_t FS_RESULT;
```



- *FS\_PASS*
- *FS\_FAIL\_DIO*

**Example of function call:**

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

```
#define BACKUP_ENABLE 1
#define LOGICAL_ONE 1
#define LOGICAL_ZERO 0

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

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

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The function cannot be used with MKE0x devices. The tested must be configured as GPIO input and adjacent pins must be configured as a GPIO output before calling the function. If the backup functionality is enabled, the function sets directions for both pins. If not, configure the directions (the tested pin as the input, the adjacent pin as the output). 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.

## 4.2.9 FS\_DIO\_ShortToAdjSet\_RT()

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-supply test, this test requires the use of two functions. The second (get) function evaluates the test result. The *FS\_DIO\_InputExt\_RT()* function is described in the respective chapter. Specify the tested pin and the adjacent pin for the input test function.

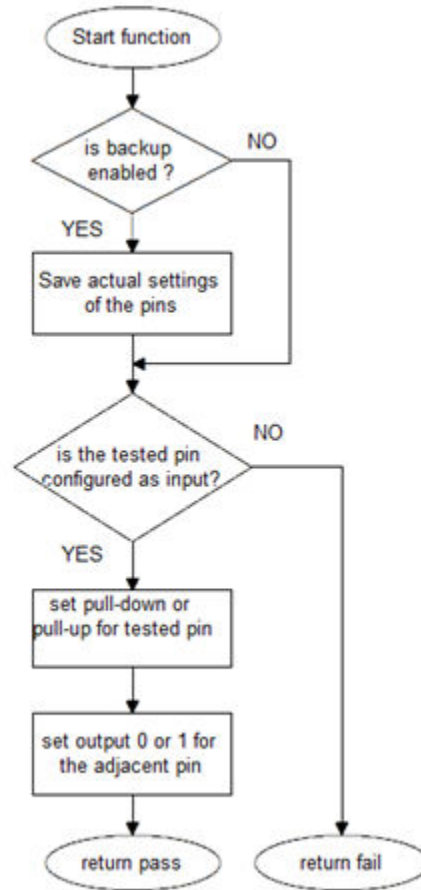


Figure 11. Block diagram of FS\_DIO\_ShortToAdjSet\_RT() function

**Function prototype:**

*FS\_RESULT FS\_DIO\_ShortToAdjSet\_RT(fs\_dio\_test\_rt\_t \*pTestedPin, fs\_dio\_test\_rt\_t \*pAdjPin, bool\_t testedPinValue, bool\_t backupEnable);*

**Function inputs:**

- \*pTestedPin* - The pointer to the tested pin struct.
- \*pAdjPin* - The pointer to the adjacent pin struct.
- testedPinValue* - The value that is set on the tested pin.
- backupEnable* - The flag. If it is non-zero, the backup functionality is enable/active.

**Function output:**

*typedef uint32\_t FS\_RESULT;*

- *FS\_PASS*
- *FS\_FAIL\_DIO*

**Example of function call:**

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

```

#define BACKUP_ENABLE 1
#define LOGICAL_ONE 1
    
```

```

#define LOGICAL_ZERO 0

dio_short_to_adj_test_result = FS_DIO_ShortToAdjSet_RT(&dio_safety_test_items[0],
&dio_safety_test_items[1], LOGICAL_ONE, BACKUP_ENABLE);

dio_short_to_adj_test_result =FS_DIO_InputExt_RT(&dio_safety_test_items[0], &dio_safety_test_items[1],
LOGICAL_ONE, 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 configure as GPIO input and the adjacent pin must be configured as a 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 pin nor the adjacent pin until the *FS\_DIO\_InputExt\_RT()* function for these pins is called.

**4.2.9.1 FS\_DIO\_ShortToAdjSet\_LPC()**

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-supply test, this test requires the use of two functions. The second (get) function evaluates the test result. The *FS\_DIO\_InputExt\_LPC()* function is described in the respective section. Specify the tested pin and the adjacent pin for the input test function.

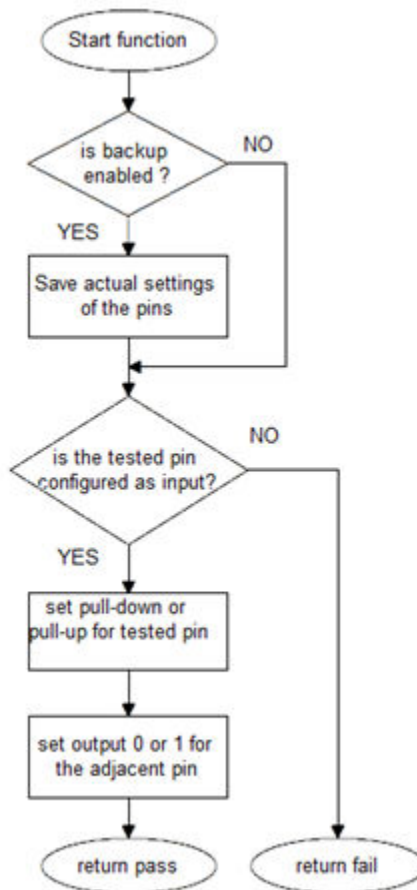


Figure 12. Block diagram of FS\_DIO\_ShortToAdjSet\_LPC() function

Digital input/output test

**Function prototype:**

```
FS_RESULT FS_DIO_ShortToAdjSet_LPC(fs_dio_test_lpc_t *pTestedPin, fs_dio_test_lpc_t *pAdjPin, bool_t testedPinValue, bool_t backupEnable);
```

**Function inputs:**

*\*pTestedPin* - The pointer to the tested pin struct.

*\*pAdjPin* - The pointer to the adjacent pin struct.

*testedPinValue* - The value that is set on the tested pin.

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

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*
- *FS\_FAIL\_DIO*

**Example of function call:**

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

```
#define BACKUP_ENABLE 1
#define LOGICAL_ONE 1
#define LOGICAL_ZERO 0

dio_short_to_adj_test_result = FS_DIO_ShortToAdjSet_LPC(&dio_safety_test_items[0],
&dio_safety_test_items[1], LOGICAL_ONE, BACKUP_ENABLE);

dio_short_to_adj_test_result = FS_DIO_InputExt_LPC(&dio_safety_test_items[0], &dio_safety_test_items[1],
LOGICAL_ONE, BACKUP_ENABLE);
```

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The tested must be configured as GPIO input and adjacent pins must be configured as a 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 pins, until the *FS\_DIO\_InputExt\_LPC()* function for these pins is called.

## 4.2.10 FS\_DIO\_ShortToAdjSet\_LPC()

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-supply test, this test requires the use of two functions. The second (get) function evaluates the test result. The *FS\_DIO\_InputExt\_LPC()* function is described in the respective section. Specify the tested pin and the adjacent pin for the input test function.

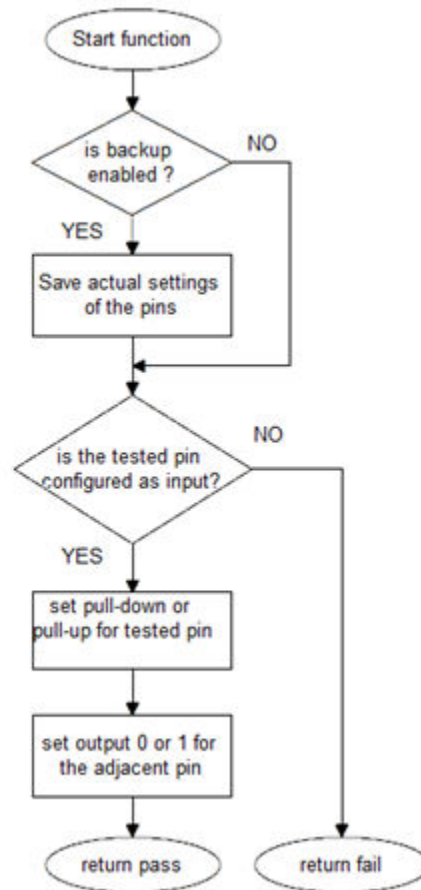


Figure 13. Block diagram of FS\_DIO\_ShortToAdjSet\_LPC() function

**Function prototype:**

```
FS_RESULT FS_DIO_ShortToAdjSet_LPC(fs_dio_test_lpc_t *pTestedPin, fs_dio_test_lpc_t *pAdjPin, bool_t testedPinValue, bool_t backupEnable);
```

**Function inputs:**

*\*pTestedPin* - The pointer to the tested pin struct.

*\*pAdjPin* - The pointer to the adjacent pin struct.

*testedPinValue* - The value that is set on the tested pin.

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

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- FS\_PASS
- FS\_FAIL\_DIO

**Example of function call:**

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

```
#define BACKUP_ENABLE 1
#define LOGICAL_ONE 1
```

```
#define LOGICAL_ZERO 0

dio_short_to_adj_test_result = FS_DIO_ShortToAdjSet_LPC(&dio_safety_test_items[0],
&dio_safety_test_items[1], LOGICAL_ONE, BACKUP_ENABLE);

dio_short_to_adj_test_result =FS_DIO_InputExt_LPC(&dio_safety_test_items[0], &dio_safety_test_items[1],
LOGICAL_ONE, BACKUP_ENABLE);
```

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The tested must be configured as GPIO input and adjacent pins must be configured as a 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 pins, until the *FS\_DIO\_InputExt\_LPC()* function for these pins is called.

### 4.2.11 FS\_DIO\_ShortToSupplySet()

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 supply voltage (Vcc, Vdd) or 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 that is described in the respective section. The main purpose of the *FS\_DIO\_InputExt()* function is to set the pull-up (or pull-down) resistor connection on the tested pin. It also ensures whether the pin is correctly configured and backs up its settings (if needed).

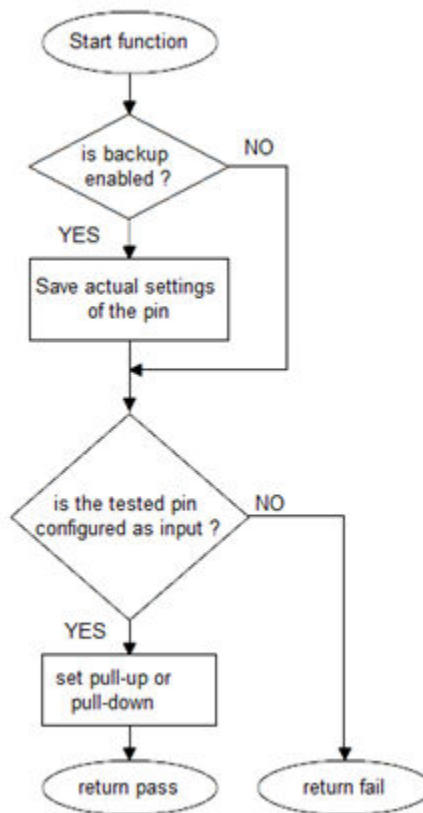


Figure 14. Block diagram of FS\_DIO\_ShortToSupplySet function

**Function prototype:**

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

**Function inputs:**

*\*pTestedPin* - The pointer to the tested pin struct.

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

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

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*
- *FS\_FAIL\_DIO*

**Example of function call:**

The following is a code example of the test for both the short-to-GND and short-to-VDD cases. Note that the implementation difference is only in one parameter. If the short to GND is tested, the parameter must have non-zero value and vice versa.

```
#define DIO_SHORT_TO_GND_TEST 1
#define DIO_SHORT_TO_VDD_TEST 0

dio_short_to_vcc_test_result = FS_DIO_ShortToSupplySet(&dio_safety_test_items[0],
DIO_SHORT_TO_GND_TEST, BACKUP_ENABLE);

dio_short_to_vcc_test_result = FS_DIO_InputExt(&dio_safety_test_items[0], &dio_safety_test_items[0],
DIO_SHORT_TO_GND_TEST, BACKUP_ENABLE);

dio_short_to_vcc_test_result = FS_DIO_ShortToSupplySet(&dio_safety_test_items[0],
DIO_SHORT_TO_VDD_TEST, BACKUP_ENABLE);

dio_short_to_vcc_test_result = FS_DIO_InputExt(&dio_safety_test_items[0], &dio_safety_test_items[0],
DIO_SHORT_TO_VDD_TEST, BACKUP_ENABLE);
```

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The function cannot be used with MKE0x devices. The tested pin must be configured as a 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 for the tested pin is called.

## 4.2.12 FS\_DIO\_ShortToSupplySet\_RT()

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 supply voltage (VCC, VDD) or 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\_RT()* function described in the respective section. The main purpose of the *FS\_DIO\_InputExt\_RT()* function is to set the pull-up or pull-down resistor connections on the tested pin. It also ensures whether the pin is correctly configured and makes a backup of its settings (if needed).

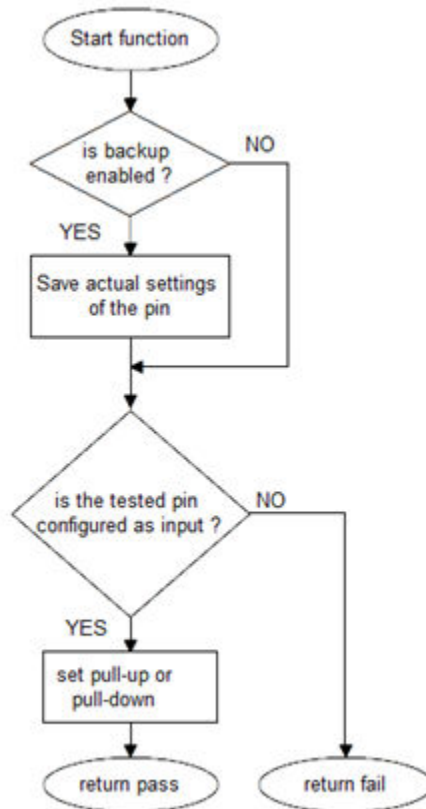


Figure 15. Block diagram of FS\_DIO\_ShortToSupplySet\_RT function

**Function prototype:**

*FS\_RESULT FS\_DIO\_ShortToSupplySet\_RT(fs\_dio\_test\_rt\_t \*pTestedPin, bool\_t shortToVoltage, bool\_t backupEnable);*

**Function inputs:**

*\*pTestedPin* - The pointer to the tested pin struct.

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

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

**Function output:**

*typedef uint32\_t FS\_RESULT;*

- *FS\_PASS*
- *FS\_FAIL\_DIO*

**Example of function call:**

The following is a code example of the test for both the short-to-GND and short-to-VDD cases. Note that the implementation difference is only in one parameter. If the short to the GND is tested, the parameter must have a non-zero value (vice versa).

```

#define DIO_SHORT_TO_GND_TEST 1
#define DIO_SHORT_TO_VDD_TEST 0

dio_short_to_vcc_test_result = FS_DIO_ShortToSupplySet_RT(&dio_safety_test_items[0],
DIO_SHORT_TO_GND_TEST, BACKUP_ENABLE);
    
```



```

dio_short_to_vcc_test_result = FS_DIO_InputExt_RT(&dio_safety_test_items[0], &dio_safety_test_items[0],
DIO_SHORT_TO_GND_TEST, BACKUP_ENABLE);

```

```

dio_short_to_vcc_test_result = FS_DIO_ShortToSupplySet_RT(&dio_safety_test_items[0],
DIO_SHORT_TO_VDD_TEST, BACKUP_ENABLE);

```

```

dio_short_to_vcc_test_result = FS_DIO_InputExt_RT(&dio_safety_test_items[0], &dio_safety_test_items[0],
DIO_SHORT_TO_VDD_TEST, 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\_RT()* function is called for the tested pin.

### 4.2.13 FS\_DIO\_ShortToSupplySet\_LPC()

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 supply voltage (Vcc, Vdd) or 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\_LPC()* function described in the respective section. The main purpose of the *FS\_DIO\_InputExt\_LPC()* function is to set the pull-up or pull-down resistor connections on the tested pin. It also tests whether the pin is correctly configured and makes a backup of its settings (if needed).

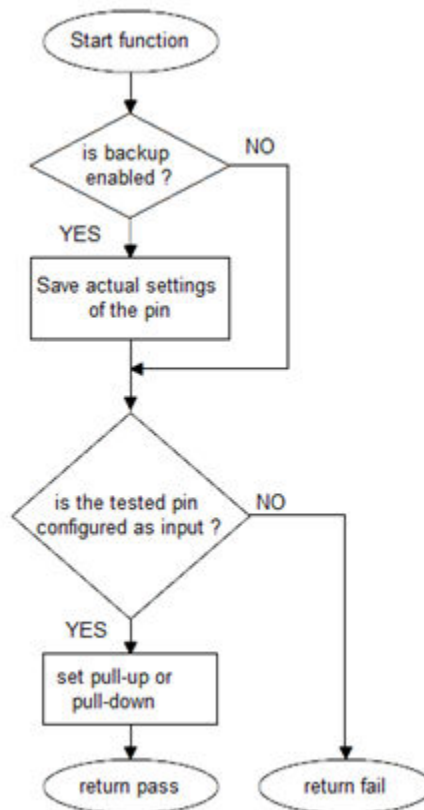


Figure 16. Block diagram of FS\_DIO\_ShortToSupplySet\_LPC function

Digital input/output test

**Function prototype:**

*FS\_RESULT FS\_DIO\_ShortToSupplySet\_LPC(fs\_dio\_test\_lpc\_t \*pTestedPin, bool\_t shortToVoltage, bool\_t backupEnable);*

**Function inputs:**

*\*pTestedPin* - The pointer to the tested pin struct.

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

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

**Function output:**

*typedef uint32\_t FS\_RESULT;*

- *FS\_PASS*
- *FS\_FAIL\_DIO*

**Example of function call:**

The following is a code example of the test for both the short-to-GND and short-to-VDD cases. Note that the implementation difference is only in one parameter. If the short to GND is tested, the parameter must have non-zero value (and vice versa).

```
#define DIO_SHORT_TO_GND_TEST 1
#define DIO_SHORT_TO_VDD_TEST 0

dio_short_to_vcc_test_result = FS_DIO_ShortToSupplySet_LPC(&dio_safety_test_items[0],
DIO_SHORT_TO_GND_TEST, BACKUP_ENABLE);

dio_short_to_vcc_test_result = FS_DIO_InputExt_LPC(&dio_safety_test_items[0], &dio_safety_test_items[0],
DIO_SHORT_TO_GND_TEST, BACKUP_ENABLE);

dio_short_to_vcc_test_result = FS_DIO_ShortToSupplySet_LPC(&dio_safety_test_items[0],
DIO_SHORT_TO_VDD_TEST, BACKUP_ENABLE);

dio_short_to_vcc_test_result = FS_DIO_InputExt_LPC(&dio_safety_test_items[0], &dio_safety_test_items[0],
DIO_SHORT_TO_VDD_TEST, 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 a 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\_LPC()* function for the tested pin is called.

# Chapter 5

## Invariable memory test

The invariable memory on the supported CM0 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 into the invariable memory, but in a different area than the one that the checksum is calculated for. In runtime and after the 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) 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 hardware CRC module. In IAR IDE, you can calculate the CRC by the linker. In other IDEs, you can use an external tool. For the Keil uVision IDE, see *Calculating Post-Build CRC in Arm® Keil®* (document [AN12520](#)).

Some 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. The software version of the test must be used on devices without a CRC hardware module.

### 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 13. 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 *iec60730b\_cm0\_flash.S* and written as assembler functions. The header file with the definitions and function prototypes is *iec60730b\_flash.h*. The *iec60730b.h* and *asm\_mac\_common.h* files must be placed in the application as well. These functions are located in *iec60730b\_flash.S*:

- FS\_CM0\_FLASH\_HW16() / FS\_CM0\_FLASH\_HW16\_LPC()
- FS\_CM0\_FLASH\_SW16()

The functions have the same input and output parameters and the same functionality. The HW16 function uses the hardware CRC module that is included in the supported MCU. The SW16 function calculates the CRC value without hardware support, so it has longer execution time.

#### 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. A checksum calculation is best done with a linker. However, this is not possible in all compilers. The following example is valid only for the IAR IDE. For further details, refer to the IAR documentation. For using external tool in Keil-uVision IDE, see *Calculating Post-Build CRC in Arm® Keil®* (document [AN12520](#)).

## Invariable memory test

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 is stored. To do this, the linker configuration file must be modified. The path to the linker configuration file can be found in: Project > Options > Linker > Config. The file name extension is \*.icf. For this example, the "CHECKSUM" block with the ".checksum" section is defined.

```
define symbol __FlashCRC_start__ = 0x6FF0;
define symbol __FlashCRC_end__ = 0x6FFF;
define region CRC_region = mem:[from __FlashCRC_start__ to __FlashCRC_end__];
define block CHECKSUM { section .checksum };
place in CRC_region { block CHECKSUM };
```

The input parameters for the CRC calculation must be set up in the linker option tabs: Project > Options > Linker. There are two options for setting up the calculation parameters. The first option is used to calculate the checksum for one block of memory in your application. The parameters are filled in the "Checksum" subtab. For this example, the start and end addresses are 0x510 and 0x3000. The unused memory is filled with 0xFF. The checksum is stored with 16 bits. The checksum algorithm is CRC16 with the standard 0x1021 polynomial. The initial seed is zero. The block size for a particular calculation is 8 bits. The variable for the result is `__checksum`.

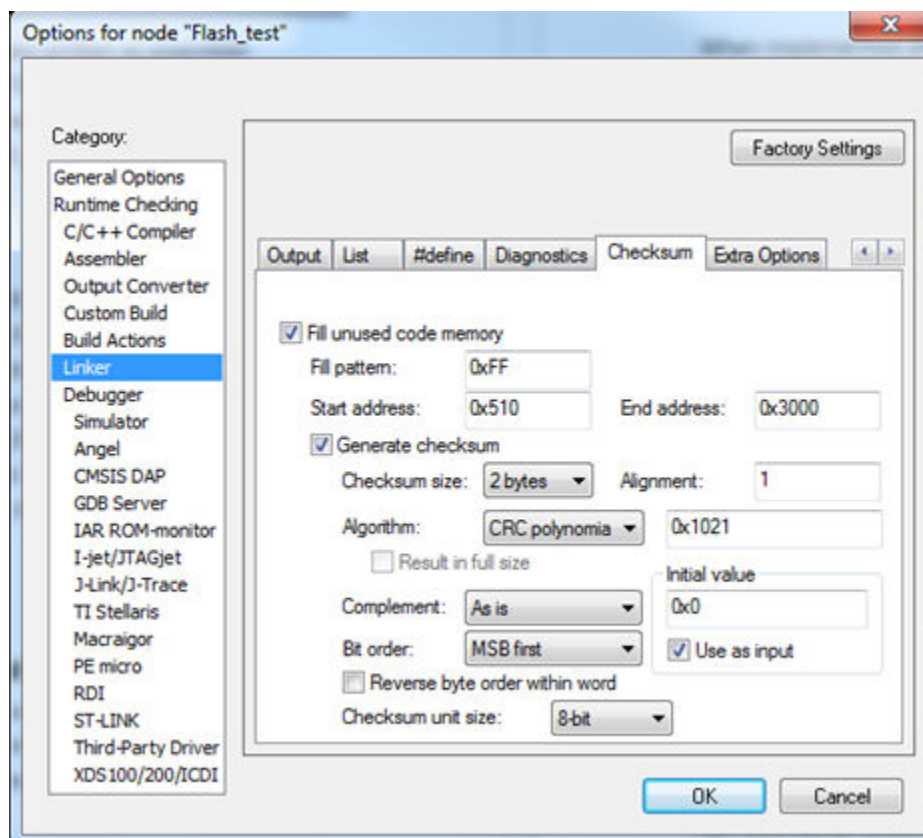


Figure 17. Checksum settings for linker

The constant variable name (`__checksum`) must be written into Project > Options > Linker > Input > Keep symbols.

The following lines must be placed into the source code, to have the `__checksum` variable available in the application.

```
#pragma section = ".checksum"
#pragma location = ".checksum"
```

```
extern unsigned short const __checksum;
```

If you need a CRC calculation for more memory blocks, use the following approach. There must be enough space in the block defined in the linker configuration file. For this example, the parameters for the calculations are the same as in the previous example and the addresses of blocks are: (0x510 – 0x610, 0x620 – 0x720, 0x730 – 0x830). The variables are as follows: (*\_\_checksum\_first*, *\_\_checksum\_second*, *\_\_checksum\_third*). In this case, the linker command line directives are used: Project > Options > Linker > Extra Options. Use the command line options and enter the following lines there. Uncheck the options in the "Checksum" subtab.

```
—fill 0xFF;0x510-0x610
—checksum __checksum_first:2,crc16,0x0;0x510-0x610
—place_holder __checksum_first,2,.checksum,4

—fill 0xFF;0x620-0x720
—checksum __checksum_second:2,crc16,0x0;0x620-0x720
—place_holder __checksum_second,2,.checksum,4

—fill 0xFF;0x730-0x830
—checksum __checksum_third:2,crc16,0x0;0x730-0x830
—place_holder __checksum_third,2,.checksum,4
```

Project > Options > Linker > Input

Write this to the "Keep symbols" block:

```
__checksum_first
__checksum_second
__checksum_third
```

Add the following lines to the source code, so that the *\_\_checksum\_first*, *\_\_checksum\_second*, and *\_\_checksum\_third* variables are available in the application.

```
#pragma section = ".checksum"
#pragma location = ".checksum"
extern unsigned short const __checksum_first;
extern unsigned short const __checksum_second;
extern unsigned short const __checksum_third;
```

## 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"
#pragma section = ".checksum"
#pragma location = ".checksum"
```

Invariable memory test

```
extern uint16_t const __checksum;  
if((uint16_t) __checksum != FS_CM0_FLASH_HW16(start_address, size, CRC_BASE, start_seed))  
SafetyError();
```

Where:

- \_\_checksum - The constant variable with the CRC value computed in the linking phase of the 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.
- start\_seed - The start condition seed. It must be "0" for the algorithm used.

### 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"  
#pragma section = ".checksum"  
#pragma location = ".checksum"  
extern unsigned short const __checksum;  
  
flash_crc.part_crc = FS_CM0_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 constant variable with the CRC value computed in the post-build phase of the application.
- flash\_crc.part\_crc - The particular CRC result and 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\_CM0\_FLASH\_SW16()

This function generates the 16-bit CRC value using software.

**Function prototype:**

```
uint16_t FS_CM0_FLASH_SW16(uint32_t startAddress, uint32_t size, uint32_t moduleAddress, uint16_t crcVal);
```

**Function inputs:**

*startAddress* - The first address of the tested memory.

*size* - The size of the tested memory.

*moduleAddress* - This input has no effect due to the compatibility with the hardware function.

*crcVal* - The start condition seed. For the first iteration, it is typically a user-defined value. For the next iterations, it is the result from the previous function call.

**Function output:**

*uint16\_t* - The 16-bit CRC value of the memory range (CRC-16-CCITT - normal 0x1021).

**Function performance:**

The function size is 76 B.<sup>1</sup>

The function duration depends on the defined block size. Several examples are shown in this table:<sup>1</sup>

**Table 14. Duration of FS\_CM0\_FLASH\_SW16() in dependence of tested block size**

Block size (Bytes)	Clock cycles	Execution time (approximately)
0x10	1845	25.62 µs
0x20	3843	53.375 µs
0x50	9724	135.06 µs

**Calling restrictions:**

None.

## 5.2.5 FS\_CM0\_FLASH\_HW16()

This function generates the 16-bit CRC value using the hardware CRC module.

**Function prototype:**

*uint16\_t* FS\_CM0\_FLASH\_HW16(*uint32\_t* startAddress, *uint32\_t* size, *uint32\_t* moduleAddress, *uint16\_t* crcVal);

**Function inputs:**

*startAddress* - The first address of the tested memory.

*size* - The size of the tested memory.

*moduleAddress* - The address of the CRC module.

*crcVal* - The start condition seed. For the first iteration, it is typically a user-defined value. For the next iterations, it is the result from the previous function call).

**Function output:**

*uint16\_t* - The 16-bit CRC value of the memory range (CRC-16-CCITT - normal 0x1021).

**Function performance:**

The function size is 44 B.<sup>1</sup>

The function duration depends on the defined block size. Several examples are shown in this table:<sup>1</sup>

**Table 15. Duration of FS\_CM0\_FLASH\_HW16() in dependence of tested block size**

Block size (Bytes)	Clock cycles	Execution time (approximately)
0x10	267	3.7 µs
0x20	465	6.5 µs
0x50	1062	14.7 µs

Invariable memory test

**Calling restrictions:**

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

## 5.2.6 FS\_CM0\_FLASH\_HW16\_LPC()

This function generates the 16-bit CRC value using the hardware CRC module.

**Function prototype:**

```
uint16_t FS_CM0_FLASH_HW16_LPC(uint32_t startAddress, uint32_t size, uint32_t moduleAddress, uint16_t crcVal);
```

**Function inputs:**

*startAddress* - The first address of the tested memory.

*size* - The size of the tested memory.

*moduleAddress* - The address of the CRC module.

*crcVal* - The start condition seed. For the first iteration, it is typically a user-defined value. For the next iterations, it is the result from the previous function call.

**Function output:**

*uint16\_t* - The 16-bit CRC value of the memory range (CRC-16-CCITT - normal 0x1021).

**Function performance:**

The function size is 40 B.<sup>2</sup>

The function duration depends on the defined block size. Several examples are shown in this table:<sup>2</sup>

**Table 16. Duration of FS\_CM0\_FLASH\_HW16\_LPC() in dependence of tested block size**

Block size (Bytes)	Clock cycles	Execution time (approximately)
0x10	360	12.00 µs
0x20	664	22.13 µs
0x50	1576	52.53 µs

**Calling restrictions:**

The function cannot be interrupted with a 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.

The identification of the safety error is ensured by the specific FAIL return if the CPU program counter register does not work correctly. Assess the return value of the test function. If it is equal to the FAIL return, then the jump into the safety error handling function occurs. 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.

Contrary 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 that is testing the pattern to verify the program counter functionality.

The program counter test works without an initialization function. The short function (another object) is written in a separate file. Place this object 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 the appropriate address in the RAM memory to test the program counter.

The block diagrams for the program counter register tests are shown in this figure:

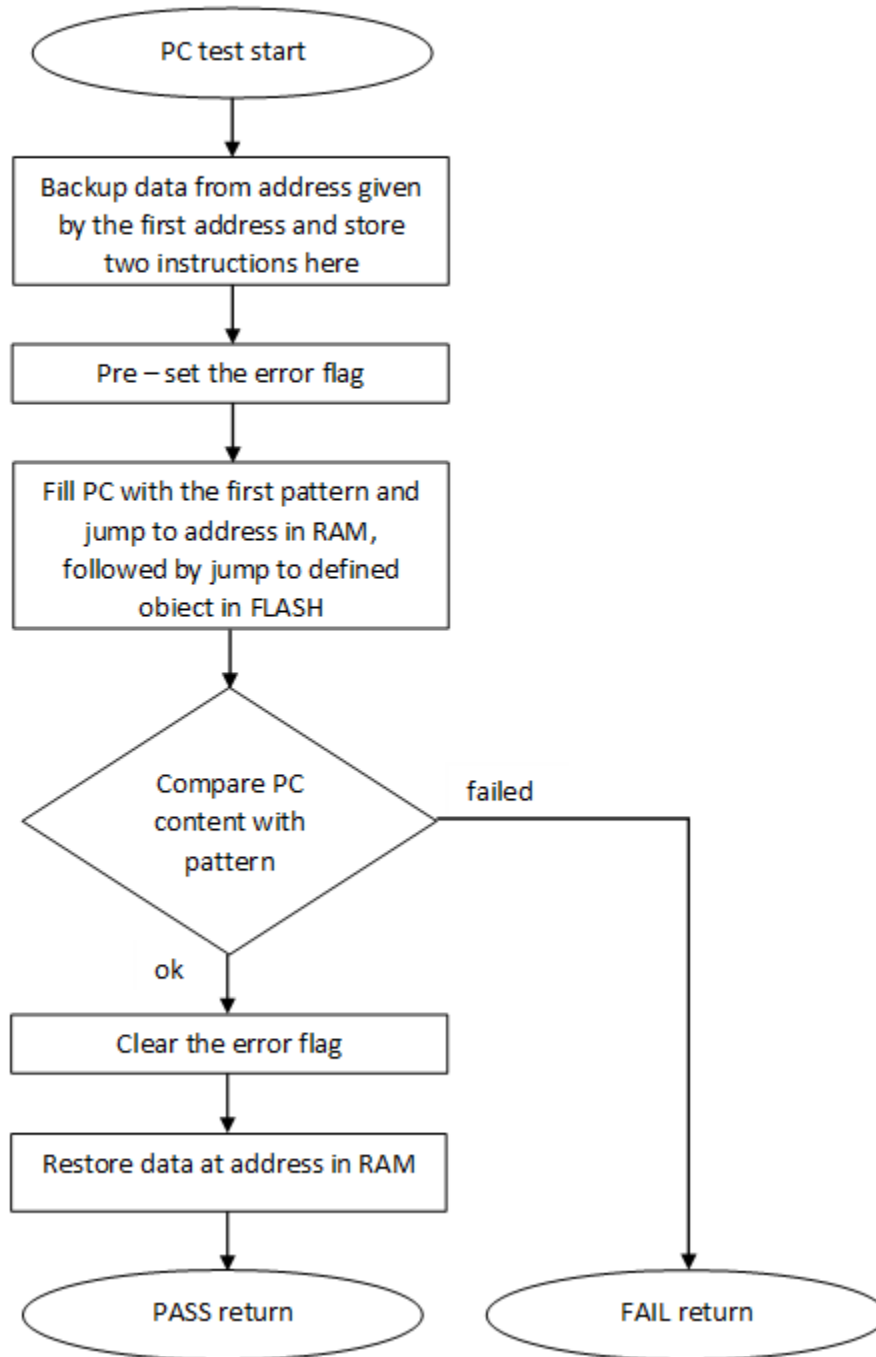


Figure 18. Block diagram for PC\_Test

## 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 this table:

Table 17. 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 registers are placed in the *iec60730b\_cm0\_pc.S* file and written as assembler functions. The header file with the test patterns and the function prototypes is *iec60730b\_cm0\_pc.h*. The *iec60730b.h* and *asm\_mac\_common.h* files are included in the *iec60730b\_cm0\_pc.S* file. Therefore, they must be also placed in the application. For the second test type, the *iec60730b\_cm0\_pc\_object.S* file must be placed to an appropriate address in the flash memory.

### Implementation example of PC test:

The only function that is handled in the application is:

*FS\_CM0\_PC\_Test()*

Place an appropriate pattern as the first input. If needed, call the function more times in a sequence with different patterns. Note that the test pattern must be a real address in the RAM and it must be even-numbered. Place the *iec60730b\_cm0\_pc\_object.S* file to an appropriate address in the flash memory.

This is an example of the function call:

```
#include "iec60730b.h"
extern unsigned long PC_test_flag; /* from Linker configuration file */
const unsigned long Program_Counter_test_flag = (unsigned long)&PC_test_flag;
#define PC_TEST_FLAG ((unsigned long *) Program_Counter_test_flag)

fs_pc_test_result = FS_CM0_PC_Test(0x20000013, FS_PC_object, PC_TEST_FLAG);
if(FS_PC_FAIL == fs_pc_test_result)
    SafetyError();
```

### 6.2.1 FS\_CM0\_PC\_Test()

The program counter register is tested according to block diagram in [Figure 18](#).

#### Function prototype:

```
FS_RESULT FS_CM0_PC_Test(uint32_t pattern1, tFcn_pc pObjectFunction, uint32_t *pFlag);
```

#### Function inputs:

*pattern1* - The address from the RAM memory, adequate as a pattern for the program counter.

*pObjectFunction* - The address of the *FS\_PC\_Object()* function.

*\*pFlag* - The address of the variable/place in memory used as a flag. If the flag is "0", the test is successful ("1" if it failed).

#### Function output:

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*
- *FS\_FAIL\_PC* - In the case of incorrect test execution, PC\_flag has a value of "1".

CPU program counter test

**Function performance:**

The function takes approximately 95 cycles (1.32 µs).<sup>1</sup>

The function size is 64 B.<sup>1</sup>

**Calling restrictions:**

This function cannot be interrupted.

## 6.2.2 FS\_PC\_Object()

This function is internally used by the *FS\_CM0\_PC\_Test()* function. The function is used to perform the program counter test. It should be called only by the *FS\_CM0\_PC\_Test()* function. It should be placed at a reliable address (by editing the linker file).

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

```
define symbol __PC_test_start__ = 0x00008FE0;
define symbol __PC_test_end__ = 0x00008FFF;
define region PC_region = mem:[from __PC_test_start__ to __PC_test_end__];
define block PC_TEST { section .text object iec60730b_cm0_pc_object.o};
place in PC_region { block PC_TEST};
```

**Function prototype:**

*void FS\_PC\_Object(void);*

**Function inputs:**

*void*

**Function output:**

*void*

**Function performance:**

The function duration is included in the duration of the *FS\_CM0\_PC\_Test()* function. Its size is 32 B.<sup>1</sup>

**Calling restrictions:**

This function is used to perform the program counter test. It should be called only by the *FS\_CM0\_PC\_Test()* function.

# Chapter 7

## Variable memory test

The variable memory test for supported devices checks the on-chip RAM for DC faults. The application stack area can also be tested. The March C and March X schemes are used as control mechanisms. Choose whether to use the March C or March X scheme. The handling functions are different for the after-reset test and for the runtime test. Both functions must have a backup area defined in the RAM and reserved by the developer. The size of this area must be at least the same as the size of the tested block. The RAM test is considered destructive. This is because the data from the memory area with the variables, the stack area, and the functions placed in the RAM is moved away, rewritten multiple times (with test patterns 0x55555555 and 0xAAAAAAAA), 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:

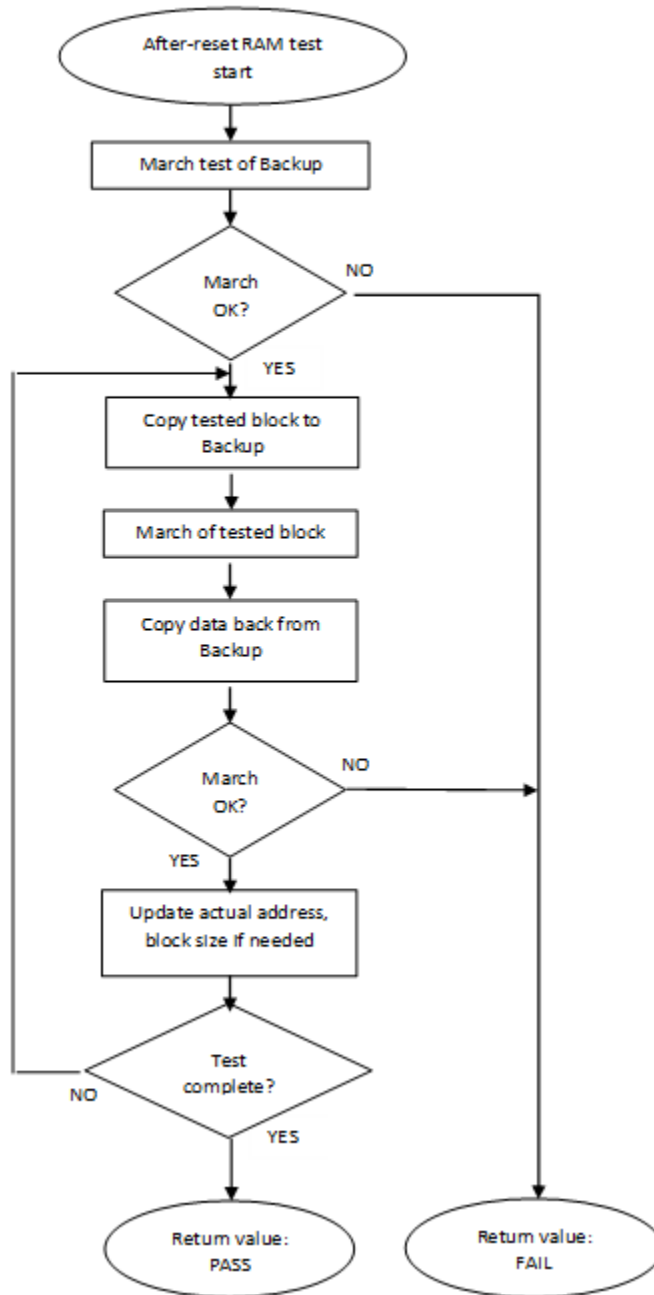


Figure 19. Block diagram for after-reset test of RAM

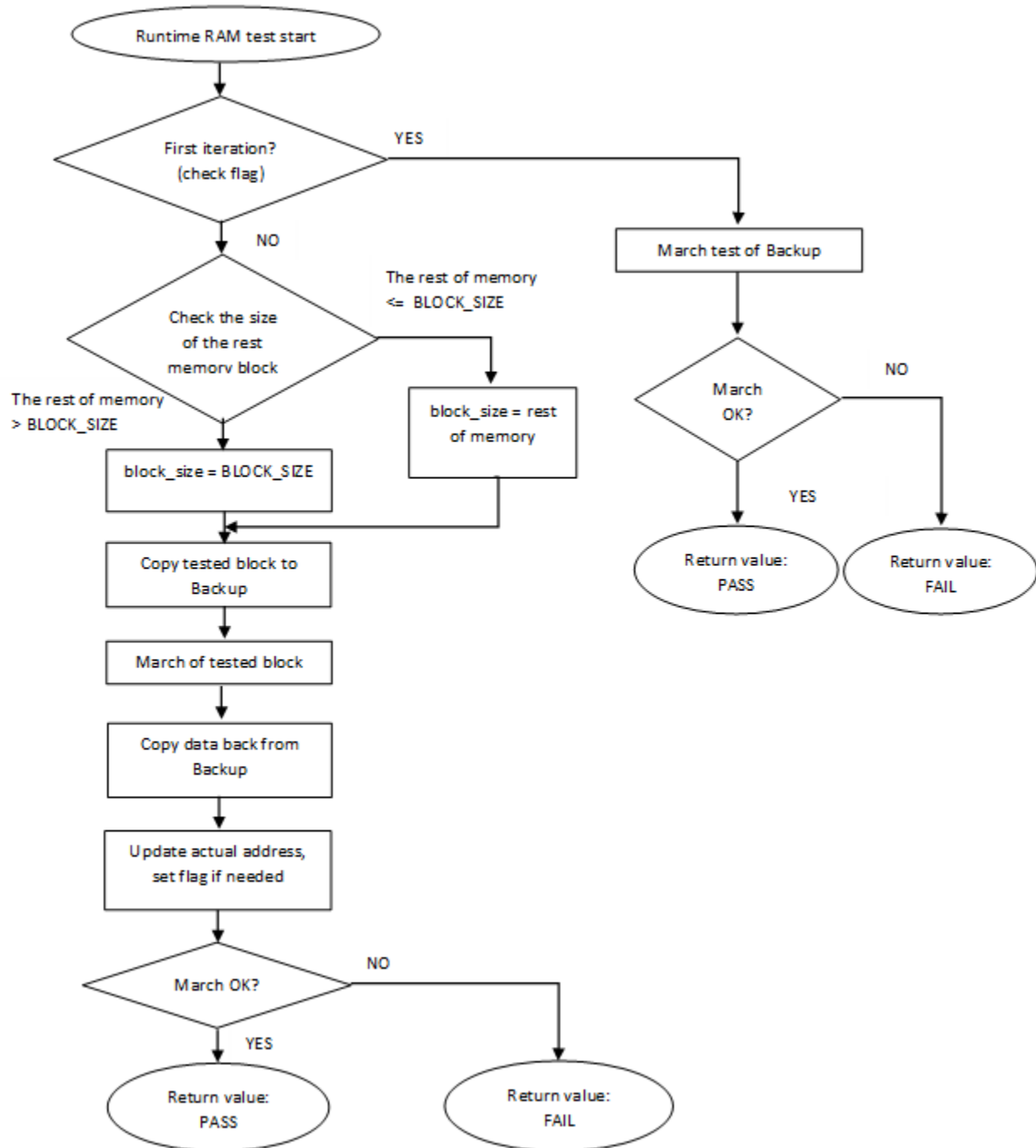


Figure 20. Block diagram for runtime test of RAM

## 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 18. 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\_ram.S* file and written as assembler functions. The header file with return values and function prototypes is *iec60730b\_ram.h*. *iec60730b.h* and *asm\_mac\_common.h* are files that are included in the *iec60730b\_ram.S* file. Therefore, they must be also placed in the application.

The RAM test consists of these public functions:

- FS\_CM0\_RAM\_AfterReset()
- FS\_CM0\_RAM\_Runtime()
- FS\_CM0\_RAM\_CopyToBackup()
- FS\_CM0\_RAM\_CopyFromBackup()
- FS\_CM0\_RAM\_SegmentMarchC()
- FS\_CM0\_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\_CM0\_RAM\_AfterReset()

The after-reset test is done by the *FS\_CM0\_RAM\_AfterReset()* function. This function is called once after reset, when the execution time is not critical. Reserve 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 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. The block diagram is shown in [Figure 19](#).

Here is an example of the function call:

```
#include "iec60730b.h"

if(FS_RAM_FAIL == FS_CM0_RAM_AfterReset(start_address, end_address, block_size, backup_address,
FS_CM0_RAM_SegmentMarchC))
    SafetyError();
```

#### Function prototype:

```
FS_RESULT FS_CM0_RAM_AfterReset(uint32_t startAddress, uint32_t endAddress, uint32_t blockSize, uint32_t
backupAddress, tFcn pMarchType);
```

#### Function inputs:

*startAddress* - The first address of the tested RAM area.

*endAddress* - The address of the first byte after the tested RAM area.

*blockSize* - The tested block size.

*backupAddress* - The address of the backup area.

*\*pMarchType* - The address of the March function (March X or March C).



**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*
- *FS\_FAIL\_RAM*

**Function performance:**

The function size is 132 B.<sup>1</sup>

The execution time depends on the memory size. It also varies with different block sizes and the March method used.<sup>1</sup>

**Table 19. FS\_CM0\_RAM\_AfterReset duration**

Memory Size (B)	Block Size (B)	Cycles - March X	Cycles - March C
0x100	0x20	4686	6182
0x100	0x40	4504	6130
0x100	0x80	4806	6752
0x200	0x20	8942	11757
0x200	0x40	8286	11221
0x200	0x80	8366	11601
0x400	0x20	17454	22919
0x400	0x40	15880	21415
0x400	0x80	15486	21301

**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 of at least the same size as the tested block size defined by the "block\_size" parameter.

## 7.2.2 FS\_CM0\_RAM\_Runtime()

The runtime test is done by the *FS\_CM0\_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. During the first call, the function checks the backup area. After the call, blocks of memory are processed in a sequence. They 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 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 above. The example of a function call is as follows:

```
#include "iec60730b.h"

if(FS_RAM_FAIL == FS_RESULT FS_CM0_RAM_Runtime(start_address, end_address, &actual_address,
block_size, backup_address, IEC60730B_RAM_SegmentMarchX))
    SafetyError();
```

**Function prototype:**

```
FS_RESULT FS_CM0_RAM_Runtime(uint32_t startAddress, uint32_t endAddress, uint32_t *pActualAddress, uint32_t
blockSize, uint32_t backupAddress, tFcn pMarchType);
```

**Function inputs:**

*startAddress* - The first address of the tested RAM area.

Variable memory test

*endAddress* - The address of the first byte after the tested RAM area.

*\*pActualAddress* - The address of the variable that holds the actual address value.

*blockSize* - The tested block size.

*backupAddress* - The address of the backup area.

*\*pMarchType* - The address of the March function (March X or March C).

**Function output:**

*typedef uint32\_t FS\_RESULT;*

- *FS\_PASS*
- *FS\_FAIL\_RAM*

**Function performance:**

The function size is 150 B.<sup>1</sup>

The execution time depends on the block size and it is different for the March C and March X methods.<sup>1</sup>

**Table 20. FS\_CM0\_RAM\_Runtime duration**

Block Size (B)	Cycles - March X	Cycles - March C
0x4	215	239
0x8	249	293
0x20	453	617
0x40	725	1049

**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\_CM0\_RAM\_CopyToBackup()

This function copies a block of memory to the dedicated backup area.

**Function prototype:**

*void FS\_CM0\_RAM\_CopyToBackup(uint32\_t startAddress, uint32\_t blockSize, uint32\_t backupAddress);*

**Function inputs:**

*startAddress* - The first address of the source.

*blockSize* - The size of the memory block.

*backupAddress* - The address of the backup area.

**Function output:**

*void*

**Function performance:**

The function size is 20 B.<sup>1</sup>

## 7.2.4 FS\_CM0\_RAM\_CopyFromBackup()

This function copies a block of memory from the backup area to the dedicated place.

**Function prototype:**

```
void FS_CM0_RAM_CopyFromBackup(uint32_t startAddress, uint32_t blockSize, uint32_t backupAddress);
```

**Function inputs:**

*startAddress* - The first address of the destination.

*blockSize* - The size of the memory block.

*backupAddress* - The address of the backup area.

**Function output:**

*void*

**Function performance:**

The function size is 20 B.<sup>1</sup>

## 7.2.5 FS\_CM0\_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.

**Function prototype:**

```
FS_RESULT FS_CM0_RAM_SegmentMarchC(uint32_t startAddress, uint32_t blockSize);
```

**Function inputs:**

*startAddress* - The first address of the tested memory block.

*blockSize* - The size of the tested memory block.

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*
- *FS\_FAIL\_RAM*

**Function performance:**

The function size is 124 B.<sup>1</sup>

## 7.2.6 FS\_CM0\_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_CM0_RAM_SegmentMarchX(uint32_t startAddress, uint32_t blockSize);
```

**Function inputs:**

*startAddress* - The first address of the tested memory block.

*blockSize* - The size of the tested memory block.

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*
- *FS\_FAIL\_RAM*

**Function performance:**

Variable memory test

The function size is 114 B.<sup>1</sup>

# Chapter 8

## CPU register test

The CPU register test procedure tests all of the CM0+ CPU registers for the stuck-at condition (except for the program counter register). 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 test of the following registers:

General-purpose registers:

- R0
- R1
- R2
- R3
- R4
- R5
- R6
- R7
- R8
- R9
- R10
- R11
- R12

Status register:

- APSR

Stack pointer registers:

- SP main
- SP process

Mask register:

- PRIMASK

Control register:

- CONTROL

Link register:

- LR

The identification of safety errors is ensured by the specific FAIL return if some registers have the stuck-at fault. Assess the return value of every function. If the value equals 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 safe state.

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 for reset in an endless loop.

The principle of the 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 or with the value written into another register that was tested before. Most of the

CPU register test

time, R0, R1, and R2 are used as auxiliary registers. Patterns are defined to check the logical one and logical zero values in all register bits.

For the PRIMASK and CONTROL tests, the original content must be backed up. For the SP\_main and SP\_process tests, the CONTROL register content must be backed up. The block diagrams for the respective registers are shown in the following figures:

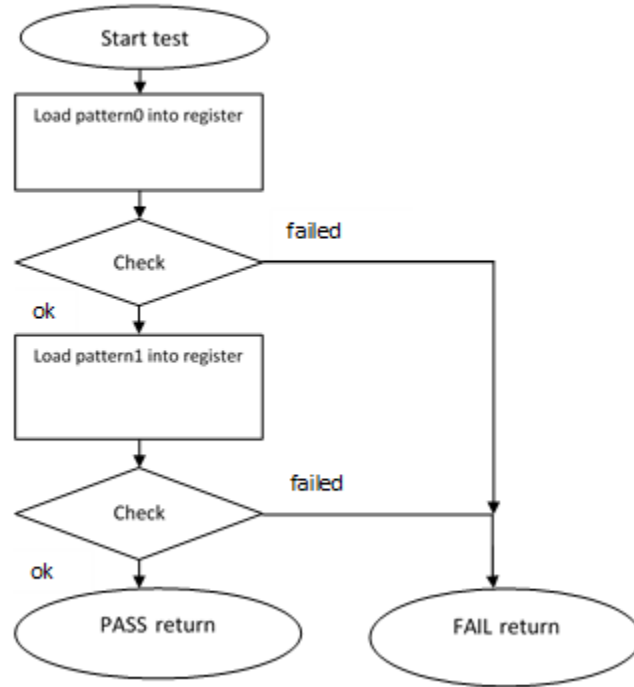


Figure 21. Block diagram for R2 – R12 registers test

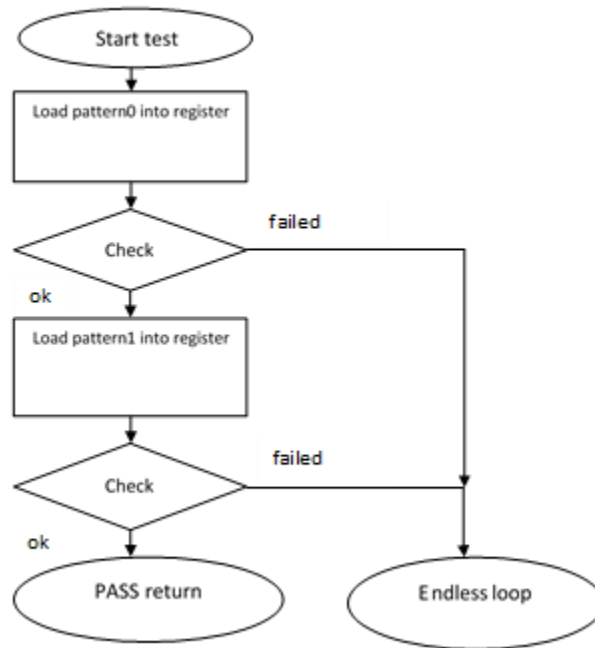


Figure 22. Block diagram for R0, R1, LR, APSR registers test

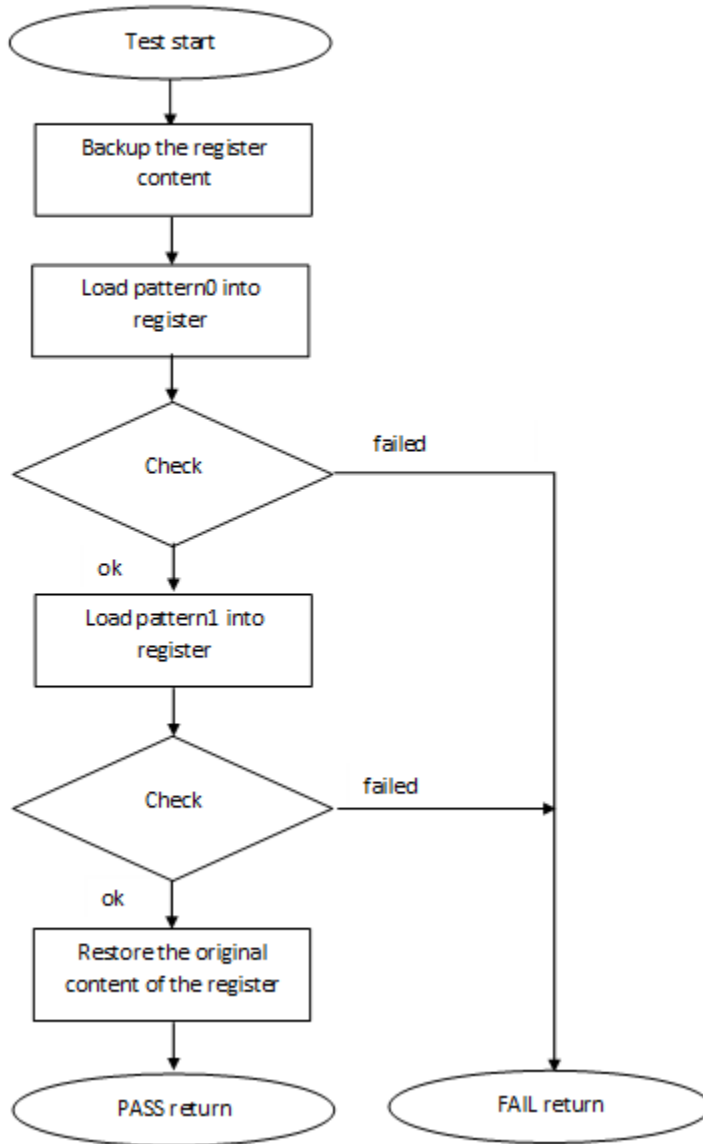


Figure 23. Block diagram for PRIMASK and CONTROL registers test



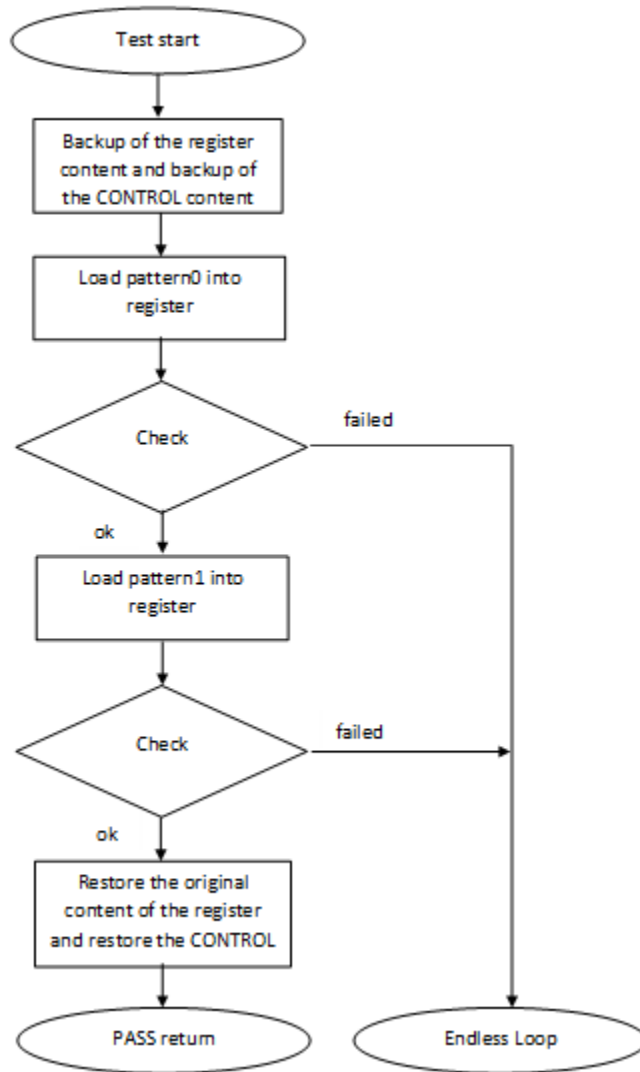


Figure 24. Block diagram for SP\_main and SP\_process registers test

## 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 the following table:

Table 21. 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 in the *iec60730b\_reg.S* file and written as assembler functions. The header file with the return values and function prototypes is *iec60730b\_reg.h*.

The *iec60730b.h*, and *asm\_mac\_common.h* files are included in the *iec60730b\_reg.S* file. Therefore, they must be also placed in the application.

The following functions are called to test the corresponding registers:

- FS\_CM0\_CPU\_Register()
- FS\_CM0\_CPU\_NonStackedRegister()
- FS\_CM0\_CPU\_Primask()
- FS\_CM0\_CPU\_SPmain()
- FS\_CM0\_CPU\_SPprocess()
- FS\_CM0\_CPU\_Control()

The error detection is recognized by the specific return value, as described in the following sections. There are several exceptions. If some of the R0, R1, LR, APSR, and SP registers are corrupted, the application is in an endless loop instead of returning an error value. If some of these registers are corrupted, the application is not able to 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. Pay attention to the functions during runtime, as described in the following chapters.

The following is an example of a function call:

```
#include "iec60730b.h"
if(FS_CPU_REGISTER_FAIL == FS_CM0_CPU_Register())
    SafetyError();
```

## 8.2.1 FS\_CM0\_CPU\_Register()

This function tests the R0-R7, R12, LR, and APSR CPU registers in a sequence. Each register is tested according to [this block diagram](#).

### **Function prototype:**

```
FS_RESULT FS_CM0_CPU_Register(void);
```

### **Test patterns for respective registers:**

R0 – R7, R12, LR: 0x55555555, 0xA0000000

APSR: 0x50000000, 0xA0000000

### **Function inputs:**

void

### **Function output:**

```
typedef uint32_t FS_RESULT;
```

- FS\_PASS
- FS\_FAIL\_CPU\_REGISTER

If the R0, R1, LR, or APSR registers are corrupted, the function is in an endless loop with the interrupts disabled. This state must be observed by another safety mechanism (for example, watchdog).

### **Function performance:**

The function takes approximately 163 cycles, including the result comparison (2.26  $\mu$ s).<sup>1</sup>

The function size is 188 B.<sup>1</sup>

### **Calling restrictions:**

None.

## 8.2.2 FS\_CM0\_CPU\_SPmain()

This function tests the SP\_main register according to [this block diagram](#).

### **Function prototype:**

```
FS_RESULT FS_CM0_CPU_SPmain(void);
```

### **Test pattern:**

```
SP_main: 0x55555554, 0xAAAAAAAA8
```

### **Function inputs:**

void

### **Function output:**

```
typedef uint32_t FS_RESULT;
```

- FS\_PASS

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

### **Function performance:**

The function takes approximately 63 cycles, including the result comparison (0.86  $\mu$ s) <sup>1</sup>

The function size is 60 B. <sup>1</sup>

### **Calling restrictions:**

This function cannot be interrupted.

## 8.2.3 FS\_CM0\_CPU\_SPprocess()

This function checks SP\_process according to [this block diagram](#).

### **Function prototype:**

```
FS_RESULT FS_CM0_CPU_SPprocess(void);
```

### **Test pattern:**

```
SP_process: 0x55555554, 0xAAAAAAAA8
```

### **Function inputs:**

void

### **Function output:**

```
typedef uint32_t FS_RESULT;
```

- FS\_PASS

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

### **Function performance:**

The function takes approximately 58 cycles, including the result comparison (0.8  $\mu$ s) <sup>1</sup>

The function size is 84 B. <sup>1</sup>

### **Calling restrictions:**

This function cannot be interrupted.

## 8.2.4 FS\_CM0\_CPU\_NonStackedRegister()

This function tests the R8, R9, R10, and R11 CPU registers. Each register is tested according to [this block diagram](#).

### **Function prototype:**

```
FS_RESULT FS_CM0_CPU_NonStackedRegister(void);
```

### **Test patterns for respective registers:**

R8 – R11: 0x55555555, 0xAAAAAAAA

### **Function inputs:**

void

### **Function output:**

```
typedef uint32_t FS_RESULT;
```

- FS\_PASS
- FS\_FAIL\_CPU\_NONSTACKED\_REGISTER

### **Function performance:**

The function takes approximately 98 cycles, including the result comparison (1.36  $\mu$ s).<sup>1</sup>

The function size is 120 B.<sup>1</sup>

### **Calling restrictions:**

None.

## 8.2.5 FS\_CM0\_CPU\_Primask()

This function tests the PRIMASK register according to [this block diagram](#).

### **Function prototype:**

```
FS_RESULT FS_CM0_CPU_Primask(void);
```

### **Test pattern:**

PRIMASK: 0x00000001, 0x00000000

### **Function inputs:**

void

### **Function output:**

```
typedef uint32_t FS_RESULT;
```

- FS\_PASS
- FS\_FAIL\_CPU\_PRIMASK

### **Function performance:**

The function takes approximately 44 cycles, including the result comparison (0.61  $\mu$ s).<sup>1</sup>

The function size is 44 B.<sup>1</sup>

### **Calling restrictions:**

This function cannot be interrupted by an interrupt where the global interrupts are disabled.

## 8.2.6 FS\_CM0\_CPU\_Control()

This function tests the R0-R7, R12, LR, and APSR CPU registers according to [this block diagram](#).

**Function prototype:**

```
FS_RESULT FS_CM0_CPU_Control(void);
```

**Test pattern:**

```
CONTROL: 0x00000000, 0x00000002
```

**Function inputs:**

```
void
```

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*
- *FS\_FAIL\_CPU\_CONTROL*

**Function performance:**

The function takes approximately 45 cycles, including the result comparison (0.63  $\mu$ s). <sup>1</sup>

The function size is 48 B. <sup>1</sup>

**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 the stuck-at faults in the memory area occupied by the stack is covered by the variable memory test. The overflow or underflow of the stack can occur if the stack is incorrectly controlled or by defining the "too low" stack area for the given application.

The principle of the test is to fill the area below and above the stack with a known pattern. These areas must be defined in the linker configuration file, together with the stack. The initialization function then fills these areas with your pattern. 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 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 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 other methods to achieve this, only one example is shown here. The size of the areas must be a multiple of 0x4. The minimum size is 0x4.

```

define symbol __ICFEDIT_region_RAM_start__ = 0x1FFFFC10;
define symbol __ICFEDIT_region_RAM_end__ = 0x20000000;
define symbol __region_RAM2_start__ = 0x20000000;
define symbol __region_RAM2_end__ = 0x200017FF;
define symbol __ICFEDIT_size_cstack__ = 512;
define exported symbol STACK_TEST_BLOCK_SIZE = 0x10;
define exported symbol STACK_TEST_P_4 = __region_RAM2_end__ - 0x3;
define exported symbol STACK_TEST_P_3 = STACK_TEST_P_4 - STACK_TEST_BLOCK_SIZE + 0x4;
define exported symbol __BOOT_STACK_ADDRESS = STACK_TEST_P_3 - 0x4;
define exported symbol STACK_TEST_P_2 = __BOOT_STACK_ADDRESS - __ICFEDIT_size_cstack__
- 0x4;
define exported symbol STACK_TEST_P_1 = STACK_TEST_P_2 - STACK_TEST_BLOCK_SIZE;
define region RAM_region = mem:[from __ICFEDIT_region_RAM_start__ to __region_RAM2_end__] - mem:
[from STACK_TEST_P_1 size 0x10] - mem:[from STACK_TEST_P_3 size 0x10];

// _____
// |_____| --> STACK_TEST_P_1 ....ADR
// |_____| ....ADR + 0x4
// |_____| ....ADR + 0x8
// |_____| --> STACK_TEST_P_2 ....ADR + 0xC
// | |
// | |

```

```

||| |
||| STACK |
|||
|||
|||
|||
|||
||| _____ | --> __BOOT_STACK_ADDRESS
||| _____ | --> STACK_TEST_P_3
||| _____ |
||| _____ |
||| _____ | --> STACK_TEST_P_4

```

In the example, the size is set to 0x10. The `STACK_TEST_P_2` and `STACK_TEST_P_3` symbols define the first addresses under and above the stack and are defined as exported symbols. This means that they are also visible in the application. The areas are not included in the RAM region, so the compiler cannot reserve this place for any variables or other parameters.

## 9.3 Stack test implementation

The test function for the stack and the initialization function are placed in the `iec60730b_stack.S` file and written as assembler functions. The header file with the return values and the function prototypes is `iec60730b_stack.h`. The `iec60730b.h` and `asm_mac_common.h` files are included in the `iec60730b_stack.S` stack. Therefore, they must be also placed in the application. The following sections show the example of the linker setup, process of initialization, and implementation.

### 9.3.1 FS\_CM0\_STACK\_Init()

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"

extern unsigned long STACK_TEST_P_2;
extern unsigned long STACK_TEST_P_3;

const unsigned long stack_test_first_address = (unsigned long)&STACK_TEST_P_2;
const unsigned long stack_test_second_address = (unsigned long)&STACK_TEST_P_3;

const unsigned long stack_test_pattern = 0x77777777;
const unsigned long stack_test_block_size = 0x10;

```

#### **Function prototype:**

```
void FS_CM0_STACK_Init(uint32_t stackTestPattern, uint32_t firstAddress, uint32_t secondAddress, uint32_t blockSize);
```

#### **Function inputs:**

*stackTestPattern* - The pattern to be written into the areas (for example, 0x77777777).

*firstAddress* - The first address of the block under the stack area.

*secondAddress* - The first address of the block above the stack area.

Stack test

*blockSize* - The size of areas under and above the stack.

**Function output:**

*void*

**Function performance:**

The function takes approximately 86 cycles for a block size of 0x10. (1.19 µs)<sup>1</sup>

The function size is 26 B.<sup>1</sup>

**Calling restrictions:**

None.

## 9.3.2 FS\_CM0\_STACK\_Test()

The testing procedure is the same after reset and during runtime. The 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 for the initialization functions.

**Function prototype:**

```
FS_RESULT FS_CM0_STACK_Test(uint32_t stackTestPattern, uint32_t firstAddress, uint32_t secondAddress, uint32_t  
blockSize);
```

**Function inputs:**

*stackTestPattern* - The pattern to be checked in the areas (for example, 0x77777777).

*firstAddress* - The first address of the block under the stack area.

*secondAddress* - The first address of the block above the stack area.

*blockSize* - The size of areas under and above the stack.

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*
- *FS\_FAIL\_STACK*

**Function performance:**

The function for block size 0x10 takes approximately 111 cycles (1.54 µs).<sup>1</sup>

The function size is 42 B.<sup>1</sup>

**Calling restrictions:**

None.



# Chapter 10

## TSI tests

The Touch Sensing Interface (TSI) provides touch sensing detection on capacitive touch sensors. The external capacitive touch sensor is typically formed on PCB and the sensor's electrodes are connected to the TSI input channels through the I/O pins in the device.

The following is a simplified block diagram of the I/O on the KE15z device:

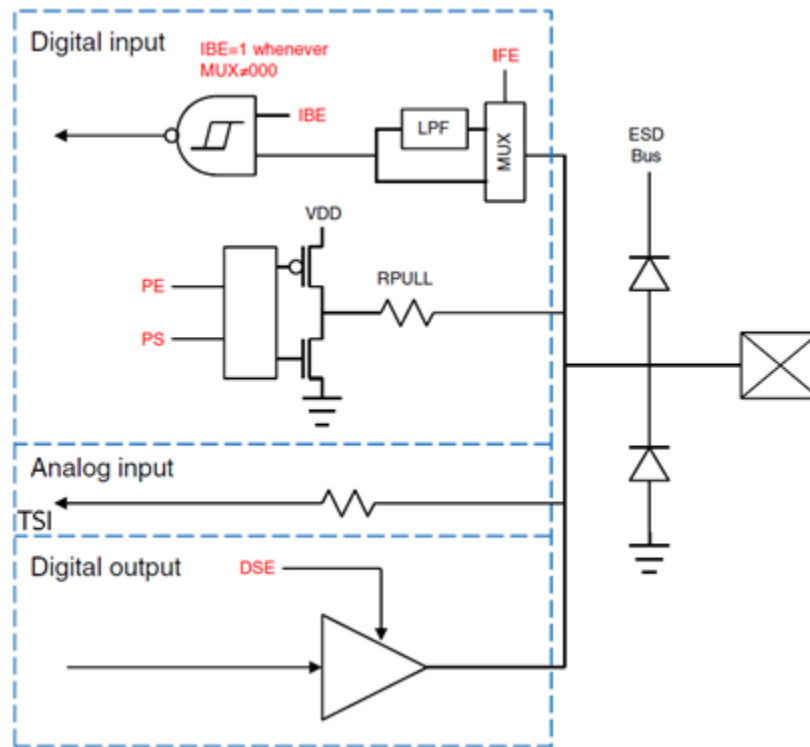


Figure 25. I/O simplified block diagram

### 10.1 TSI signal shorts tests

Because the analog TSI channels are shared with digital I/O pins and the analog or digital feature can be easily selected or switched by the software writing to the appropriate pin MUX control bits located in the Pin Control Register (PCR), the test procedure can periodically switch the pin MUX between the TSI (analog) mode and the GPIO (digital) mode. It means that switching to the GPIO mode can be helpful for testing the TSI signal trace shorts.

To test the TSI signal shorts, the following IEC60730 DIO short tests can be reused (see [Digital input/output test](#)):

- FS\_DIO\_ShortToSupplySet() / FS\_DIO\_InputExt() - to test the TSI trace short to the power supply VDD or GND.
- FS\_DIO\_ShortToAdjSet() / FS\_DIO\_InputExt() - to test the TSI trace short to the adjacent pins or traces.

### 10.2 TSI input test

This test is responsible for checking the typical conversion results of the individual TSI channels. When the touch-sensing electrode is released (not touched), the typical conversion result is given by the intrinsic (parasitic) capacitance load connected externally to the TSI channel. The intrinsic capacitance is given by physical aspects of the PCB board, such as the touch-sensing electrodes and their type, size, shape, and signal trace length. When the electrode is touched, the total external capacitive load

increases, which changes the conversion result. When the electrode is expected as released, you get the typical TSI counter value for the electrode.

### 10.2.1 TSI input electrode disconnected (open pin) tests

The TSI input test covers also issues caused by wrong (cold) soldering, corrosion, or improper PCB component placement during the manufacturing, such as wrong SMD part values or a mismatch between the SMD components.

The detection method is based on tracking the typical signal (TSI counter) value. All of the sensor electrodes have their typical signal baseline level stored in the internal flash memory (in a secure flash location, managed by the CRC) as constants that are calibrated and stored during the production of the device. In the application, the actual (measured) TSI counter value is then compared with the typical value for the individual sensors. If the actual value is lesser or much higher than the stored typical value, a fault is detected. The thresholds must be properly tuned to avoid false fault indications, because of environmental drifts and aging.

For example, two thresholds (high-watermark and low-watermark) can be selected, while expecting that the signal stays within the tolerances in normal operation conditions, where the tolerance range can be selected like a +/- 25 % deviation from the stored values.

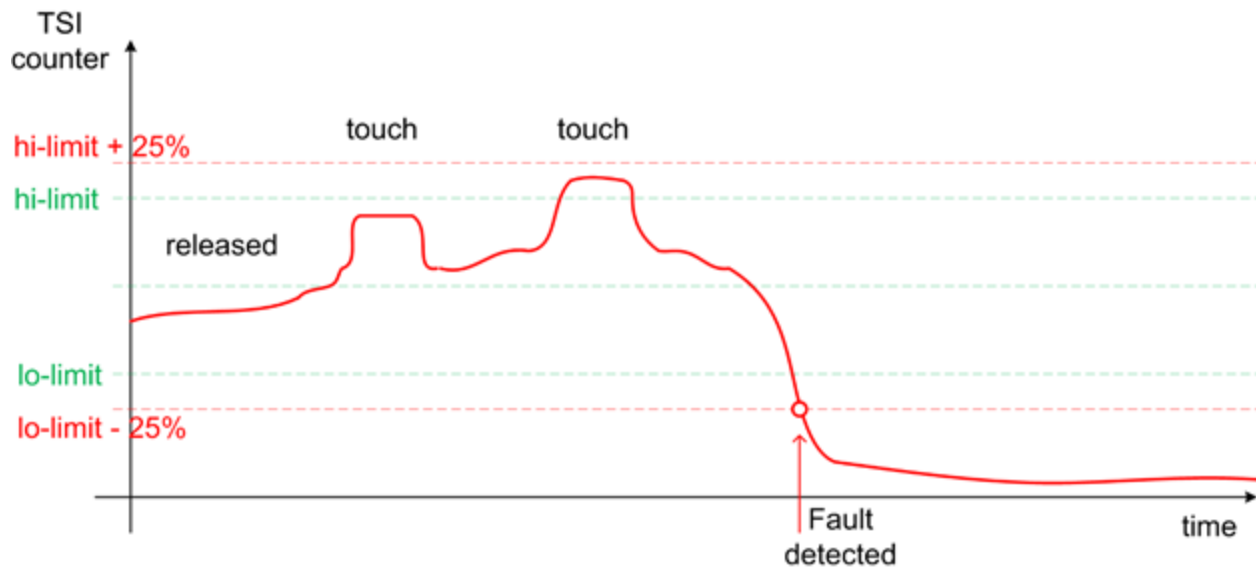


Figure 26. TSI input test fault detection

NOTE: A fault occurs when the signal drops below the low watermark or rises above the high watermark.

If the abnormal signal level is measured during the production or factory calibration, it means that there may be something wrong in the PCB manufacturing or assembly, like soldering, component placement, or mechanical assembling (shorted or bended spring electrodes, and so on).

The signal suddenly drops below the normal level when the electrode connection is lost or the signal track is terminated between the MCU pin and the electrode. It happens mostly because of cold electrode soldering or cold serial resistor soldering. The signal may suddenly rise above the normal level because of the additional loading, which may indicate a short cut or stray conductance because of long term oxidation.

### 10.3 Shorts or disconnection on guard sensors or shield electrode

The guard sensor is typically a hidden electrode connected to the dedicated TSI channel and physically surrounding the other electrodes on the PCB. It is commonly used to detect the water flood on the touch control panel and to disable the other electrodes when this issue happens. It can be used for the software offset compensation, increasing the robustness and safety. The guard electrode signal path can be tested using all the methods described above.

The shield electrode is a copper plane actively driven (buffered) by a dedicated TSI channel to compensate the parasitic capacitance and increase the sensitivity and immunity against the environmental changes (drift). The similar methods described above can be used to test the shield electrode.

## 10.4 TSI input test architecture

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

The identification of a safety error is ensured by the specific FAIL return in the case of an TSI IO error. The application developer must compare the return value of the test function with the expected value. If this is equal to the FAIL return, then the jump into a 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.

### 10.4.1 TSI input check with non-stimulated inputs

The TSI IO test is based on sequence execution, where a certain external capacity level is connected to a defined TSI input. The test function checks whether the converted value is within the tolerance. The test covers the check of the TSI input interface and checks the defined TSI input channel values.

The block diagram for the TSI IO test with non-stimulated input is shown in the following figure:

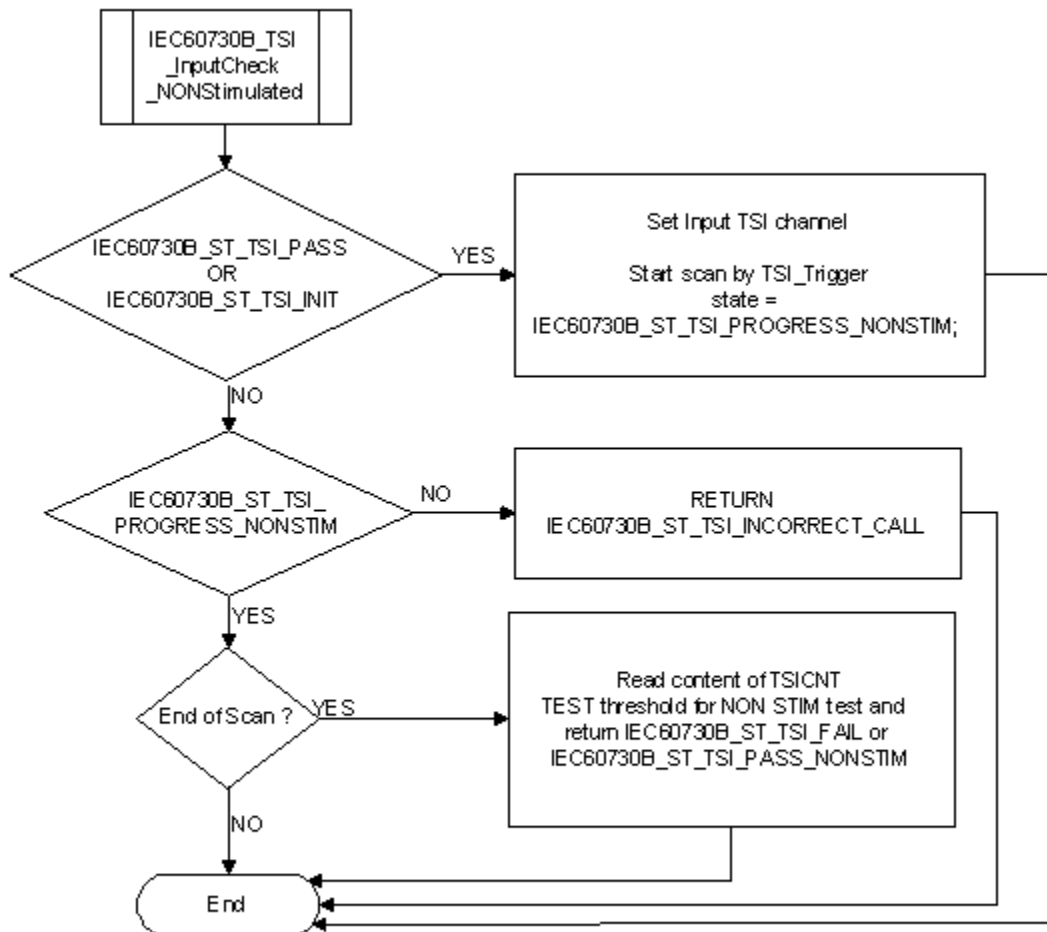


Figure 27. Software program flow for TSI input test with non-stimulated inputs

### 10.4.2 TSI input check with stimulated inputs (signal delta check)

The GPIO pull-up/down device can be enabled on an individual TSI channel pin, while the TSI channel is actively scanned to affect the analog conversion result by additional loading caused by the pull-resistor. This can be used for the stimulation of the pin. This channel stimulation is used to emulate the TSI signal (counter value) change on the desired channel pin by software, without the external touch event. By enabling of the internal pull-down or pull-up resistors on the appropriate DIO pin while the TSI measurement is active, you add the load to the charging signal, resulting in a changed accumulated TSI counter number (signal delta). Using this method, you can check the entire measurement chain from the TSI input pin to the TSI conversion counter, including the internal analog multiplexer. You can stimulate the individual TSI channel inputs, check the individual conversion results, and compare them with their typical signal delta values valid for the stimulated state. When disabling the pull device, the TSI counter value must return to the typical level valid for the idle state.

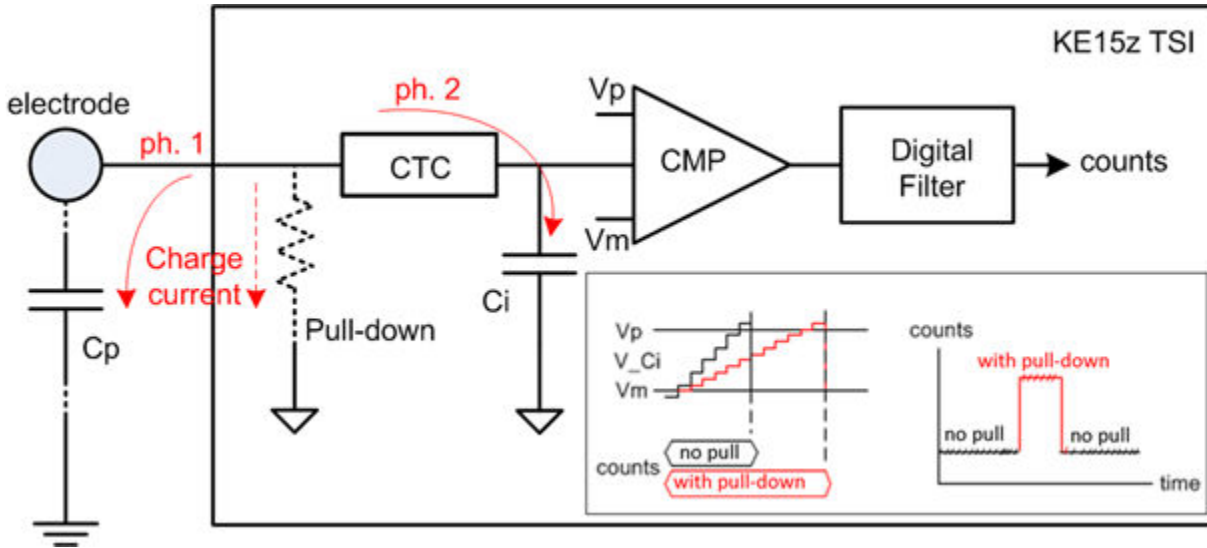


Figure 28. TSI channel stimulation principles

### 10.4.2.1 TSI input channel stimulation

In a normal state, during every external charging cycle (ph. 1), the charging current is completely used to charge the  $C_p$  up to a certain level. When the pull-down resistor is enabled, it creates an additional signal path for the charging current, where a part of the current leaks through the resistor to the GND. The  $C_p$  is charged to a smaller level (and the charge accumulated by the  $C_p$  is smaller) when compared to the normal state with the pull-down resistor disabled.

During the internal charging cycle (ph. 2), the charge accumulated by the  $C_p$  is transferred to the reference internal capacitor  $C_i$ . When the internal pull-up resistor is enabled, the charge steps are smaller. You need more charging steps to charge the  $C_i$  to the appropriate level. More charging steps result in longer time and higher count accumulated in the TSI result counter.

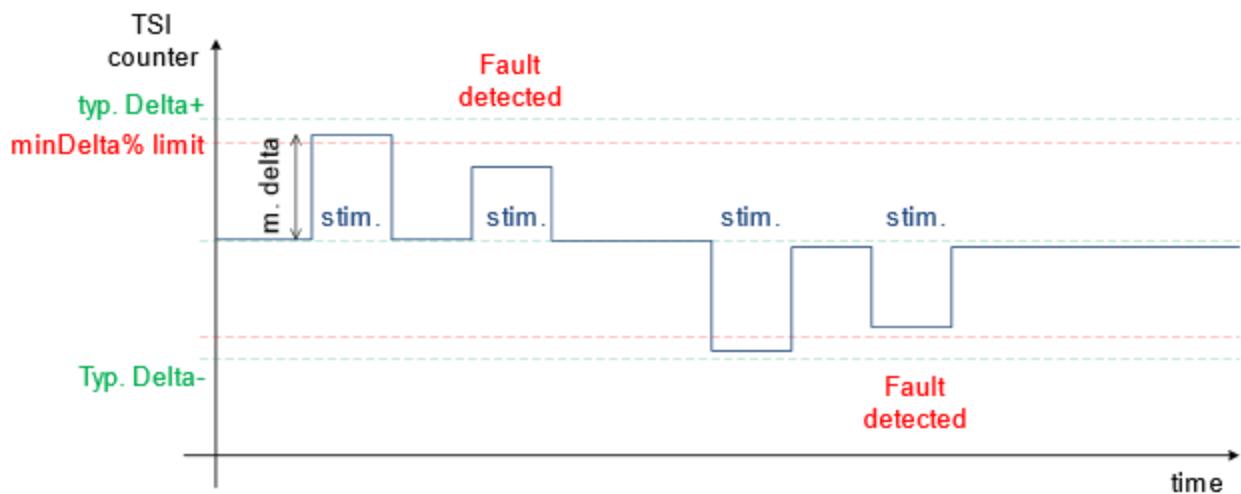


Figure 29. TSI stimulated channel Delta check

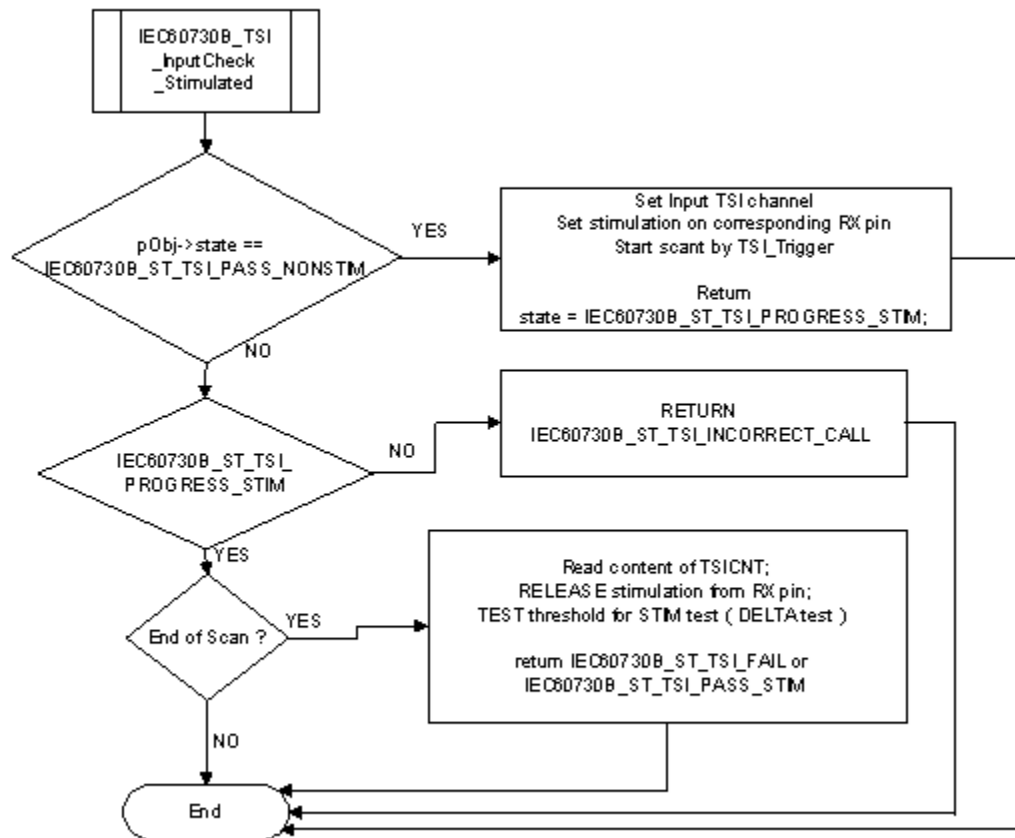


Figure 30. Software program flow for TSI input test with stimulated inputs

## 10.5 TSI test implementation

The test functions for the TSI IO test are in the *iec60730b\_tsi.c* file and written as C functions. The header file with the function prototypes is *iec60730b\_tsi.h*. The *iec60730b.h* file is the common header file for the safety library.

The following functions are called to test the TSI input:

- *FS\_TSI\_InputInit()*
- *FS\_TSI\_InputCheckNONStimulated()*
- *FS\_TSI\_InputCheckStimulated()*
- *FS\_TSI\_InputStimulate()*
- *FS\_TSI\_InputRelease()*

### 10.5.1 TSI input test principles

The principle of the TSI input test is based on checking whether the converted analog value has the expected value. This test uses the TSI inputs with known converted values and checks whether the converted values fit within the defined limits. It should normally be about +/- 25 % around the desired reference values.

The test is triggered by the first call of the `FS_TSI_InputCheckNONStimulated()` function. The test is divided into three parts (the initialization, test execution, and end of test). This test also gathers TSI counter data in the normal (non-stimulated) state, which are used as reference data for the TSI stimulated input test.

See [TSI input test](#) for more details about the test.

## 10.5.2 TSI stimulated input test principles

This test is responsible for a periodical check of the TSI counter delta change on the input stimulated by an internal pull-up. The test is triggered by the `FS_TSI_InputCheckStimulated()` function call. When the channel measurement completes, the appropriate pull resistor is disabled on the current input. The TSI counter value measured with the stimulated input is compared with the value gathered previously without stimulation. This difference is called the TSI delta signal. The TSI input channel is working properly when the delta signal is non-zero. It means that a significant counter change is measured while the input is stimulated. Depending on the TSI sensing mode and the polarity of stimulation, the delta value may have positive or negative signs. This delta value is then compared with the typical delta value experimentally measured and predefined in the configuration file. It means that the typical delta values must be measured in advance during the calibration of a known and good device. See [TSI input test](#) for more details about the test.

### NOTE

This test requires that the non-stimulated input test precedes the stimulated input test. The `FS_TSI_InputCheckNONStimulated()` and `FS_TSI_InputCheckStimulated()` functions must be called sequentially for the current TSI input channel. If the calling sequence is invalid, the function returns the `FS_TSI_INCORRECT_CALL` fail code.

## 10.5.3 TSI test input function call example

```
uint32_t SafetyTsiChanelTest(safety_common_t *psSafetyCommon, fs_tsi_t* pObj)
{
    if(pObj->state == FS_TSI_PROGRESS_NONSTIM )
    {
        FS_TSI_InputCheckNONStimulated(pObj, (uint32_t *)TSI); /*Periodically call for result check */
    }
    if (( pObj->state == FS_TSI_PASS_NONSTIM) || (pObj->state == FS_TSI_PROGRESS_STIM ) )
    { /*NON stimulated input check OK */
        FS_TSI_InputCheckStimulated(pObj, (uint32_t *)TSI);
    }
    if((pObj->state == FS_TSI_PASS ) || (pObj->state == FS_TSI_INIT ))
    { /*First call for this channel occur */
        if (pObj->input.tx_ch == SAFETY_SELFCAP_MODE) /*SET HW */
        { /* We want to test SELF CAP input*/
            Tsi0SetupSelfCap(); /* TSI HW init in Self mode */
        } else
        { /*HW to mutual cap*/
            Tsi0SetupMutualCap(); /* TSI HW init in Mutual mode */
        }
    }
}
```

```

FS_TSI_InputCheckNONStimulated(pObj, (uint32_t *)TSI);
psSafetyCommon->TSI_test_result = FS_TSI_INPROGRESS;
}
if (pObj->state == FS_TSI_PASS_STIM) /*Second part of test done => set PASS to all */
{
psSafetyCommon->TSI_test_result = FS_TSI_PASS;
}
if (pObj->state == FS_TSI_FAIL )
{ /*TEST FAIL */
psSafetyCommon->TSI_test_result = FS_TSI_FAIL;
SafetyErrorHandling(psSafetyCommon);
}
return 0;
}

```

## 10.5.4 FS\_TSI\_InputInit()

This function initializes the respective items in the defined fs\_tsi\_t structure and sets state to FS\_TSI\_INIT. It should be called before the non-stimulated input test.

### **Function prototype:**

```
void FS_TSI_InputInit(fs_tsi_t *pObj);
```

### **Function inputs:**

\*pObj- The input argument is the pointer to the TSI test instance.

### **Function output:**

void

### **Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 10.5.5 FS\_TSI\_InputCheckNONStimulated()

This function executes the first part of the TSI test sequence with a non-stimulated input. It reads the TSI counter value and checks whether the value fits into the predefined limits. It also gathers the TSI counter data for the normal (non-stimulated) state, which are required for the further stimulated input test.

The test is finished when the function reports FS\_TSI\_PASS\_NONSTIM or FS\_FAIL\_TSI.

### **Function prototype:**

```
FS_RESULT FS_TSI_InputCheckNONStimulated(fs_tsi_t *pObj, uint32_t *pTsi);
```

### **Function inputs:**

\*pObj- The input argument is the pointer to the TSI test instance.

\*pTsi- The input argument is the pointer to the TSI module.

### **Function output:**



```
typedef uint32_t FS_RESULT;
```

- *FS\_TSI\_PASS\_NONSTIM*
- *FS\_TSI\_INCORRECT\_CALL*
- *FS\_FAIL\_TSI*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 10.5.6 FS\_TSI\_InputCheckStimulated()

This function executes the second part of the TSI test sequence with a stimulated input. It checks whether the TSI input stimulated counter delta is in the expected range. The test function can be called only after passing the non-stimulated test. Otherwise, *FS\_TSI\_INCORRECT\_CALL* is returned.

**NOTE**

Normally, the *FS\_TSI\_InputCheckNONStimulated()* call precedes the *FS\_TSI\_InputCheckStimulated()* call. It is recommended to call both test functions in a close sequence.

The test is finished, when this function reports *FS\_TSI\_PASS\_STIM* or *FS\_FAIL\_TSI*.

**Function prototype:**

```
FS_RESULT FS_TSI_InputCheckStimulated(fs_tsi_t *pObj, uint32_t *pTsi);
```

**Function inputs:**

*\*pObj* - The input argument is the pointer to the TSI test instance.

*\*pTsi* - The input argument is the pointer to the TSI module.

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_TSI\_PASS\_STIM*
- *FS\_TSI\_INCORRECT\_CALL*
- *FS\_FAIL\_TSI*

**Function performance:**

The Information about the function performance is in [Core self-test library – source code version](#).

## 10.5.7 FS\_TSI\_InputStimulate()

The function stimulates the appropriate TSI pin by the pull-resistor on the current TSI channel when the TSI input stimulation is required. The pull-up/down polarity is given by the *stim\_polarity* parameter in the *fs\_tsi\_t* structure.

**Function prototype:**

```
FS_RESULT FS_TSI_InputStimulate(fs_tsi_t *pObj);
```

**Function inputs:**

*\*pObj* - The input argument is the pointer to the TSI test instance.

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*
- *FS\_FAIL\_TSI*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

## 10.5.8 FS\_TSI\_InputRelease()

This function disables the pull-resistor stimulation on the appropriate TSI channel. It is also called internally by the *FS\_TSI\_InputStimulate()* function as soon as the stimulated input check completes.

**Function prototype:**

```
FS_RESULT FS_TSI_InputRelease(fs_tsi_t *pObj);
```

**Function inputs:**

*\*pObj* - The input argument is the pointer to the TSI test instance.

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*
- *FS\_FAIL\_TSI*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

# Chapter 11

## Watchdog test

The watchdog test provides the testing of 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 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.

### 11.1 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 IEC 60730-1, IEC 60335, UL 60730, and UL 1998 standards, as described in the following table:

**Table 22. 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

### 11.2 Watchdog test implementation

The test functions for the watchdog are placed in the *iec60730b\_wdg.c* file. The header file is *iec60730b\_wdg.h*. The *iec60730b.h* and *asm\_mac\_common.h* files must be placed in the application as well.

You must have available space in the RAM memory, which is not corrupted after the non-POR.

This memory is used for your variable of the *fs\_wdog\_test* type, which is a structure with three members. It is defined in the *iec60730b\_wdg.h* file.

It is important to configure the watchdog module and the device timer before starting the watchdog test.

The watchdog timer module is different for the supported devices. For a correct function for the corresponding device, see the device implementation chapter.

Ensure the handling of the functions. To identify the source of the reset, use the reset control module. 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.

The following is an example of the watchdog test implementation (MKV1x):

```
#include "iec60730b.h"
#define WATCHDOG_ON
```

```

#define Watchdog_refresh WDOG_REFRESH = 0xA602;WDOG_REFRESH = 0xB480

extern unsigned long WD_TEST_BACKUP; /* from Linker configuration file */
const unsigned long WD_backup_address = (unsigned long)&WD_TEST_BACKUP;

#define WATCHDOG_TEST_VARIABLES ((WD_Test_Str *) WD_backup_address)

#define WD_TEST_LIMIT_HIGH 3400
#define WD_TEST_LIMIT_LOW 3000
#define ENDLESS_LOOP_ENABLE 1 /* set 1 or 0 */
#define WATCHDOG_RESETS_LIMIT 1000
#define WATCHDOG_TIMEOUT_VALUE 100

MCG_C1 |= MCG_C1_IRCLKEN_MASK; /* MCGIRCLK active */
MCG_C2 &= (~MCG_C2_IRCS_MASK); /* slow reference clock selected */
SIM_SCGC5 |= SIM_SCGC5_LPTMR_MASK; /* enable clock gate to LPTMR */
LPTMR0_CSR = 0; /* time counter mode */
LPTMR0_CSR = LPTMR_CSR_TCF_MASK|LPTMR_CSR_TFC_MASK; /* CNR reset on overflow */
LPTMR0_PSR |= LPTMR_PSR_PBYP_MASK; /* prescaler bypassed, */
LPTMR0_PSR &= (~LPTMR_PSR_PCS_MASK); /* clear prescaler clock */
LPTMR0_PSR |= LPTMR_PSR_PCS(0); /* select the clock input */
LPTMR0_CMR = 0; /* clear the compare register */
LPTMR0_CSR |= LPTMR_CSR_TEN_MASK; /* enable timer */

WatchdogEnable();

if(RCM_SRS0_POR_MASK==(RCM_SRS0_POR_MASK &RCM_SRS0)) /* if POR reset */
{
FS_WDOG_Setup(WATCHDOG_TEST_VARIABLES);
}

if(RCM_SRS0_POR_MASK!=(RCM_SRS0_POR_MASK &RCM_SRS0)) /* if non-POR reset */
{
FS_WDOG_Check(WD_TEST_LIMIT_HIGH, WD_TEST_LIMIT_LOW, WATCHDOG_RESETS_LIMIT,
ENDLESS_LOOP_ENABLE, WATCHDOG_TEST_VARIABLES);
}

```

## 11.2.1 FS\_WDOG\_Setup()

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

### Function prototype:

```
void FS_WDOG_Setup(fs_wdog_test_t *pWatchdogBackup);
```

**Function inputs:**

*\*pWatchdogBackup* - The pointer to the structure with `fs_wdog_test_t` variables.

**Function output:**

*void*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The watchdog timer and the LPTMR must be configured correctly. A variable of the `fs_wdog_test_t` type must be declared and placed into a reliable place. Interrupts should be disabled.

## 11.2.2 FS\_WDOG\_Setup\_COP()

This function can be used for the MKLxx devices. This function clears the reset counter, which is a member of the `fs_wdog_test_t` structure. It refreshes the watchdog to start counting from zero. It starts the LPTMR, which must be configured before the function call occurs. Within the waiting endless loop, the value from the LPTMR is periodically stored in the reserved area in the RAM.

**Function prototype:**

```
void FS_WDOG_Setup_COP(fs_wdog_test_t *pWatchdogBackup);
```

**Function inputs:**

*\*pWatchdogBackup* - The pointer to a structure with the `fs_wdog_test_t` variables.

**Function output:**

*void*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The watchdog timer and the LPTMR must be configured correctly. A variable of the `fs_wdog_test_t` type must be declared and placed into a reliable place. Interrupts should be disabled.

## 11.2.3 FS\_WDOG\_Setup\_KE0XZ()

Can be used for the KE0xZ devices. This function clears the reset counter, which is a member of the `fs_wdog_test_t` structure. It refreshes the watchdog to start counting from zero. It starts the RTC, which must be configured before the function call occurs. Within the waiting endless loop, the value from the RTC is periodically stored in the reserved area in the RAM.

**Function prototype:**

```
void FS_WDOG_Setup_KE0XZ(fs_wdog_test_t *pWatchdogBackup);
```

**Function inputs:**

*\*pWatchdogBackup* - The pointer to the structure with `fs_wdog_test_t` variables.

**Function output:**

*void*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The watchdog timer and the RTC 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.

## 11.2.4 FS\_WDOG\_Setup\_KE1XF()

Can be used for the MKE1xF devices. This function clears the reset counter, which is a member of the `fs_wdog_test_t` structure. It refreshes the watchdog to start counting from zero. It starts the LPTMR, which must be configured before the function call occurs. Within the waiting endless loop, the value from the LPTMR is periodically stored in the reserved area in the RAM.

**Function prototype:**

```
void FS_WDOG_Setup_KE1XF(fs_wdog_test_t *pWatchdogBackup);
```

**Function inputs:**

*\*pWatchdogBackup* - The pointer to the structure with `fs_wdog_test_t` variables.

**Function output:**

*void*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The watchdog timer and the LPTMR 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.

## 11.2.5 FS\_WDOG\_Setup\_KE1XZ()

This function can be used for the KE1xZ devices. This function clears the reset counter, which is a member of the `fs_wdog_test_t` structure. It refreshes the watchdog to start counting from zero. It starts the LPTMR, which must be configured before the function call occurs. Within the waiting endless loop, the value from the LPTMR is periodically stored in the reserved area in the RAM.

**Function prototype:**

```
void FS_WDOG_Setup_KE1XZ(fs_wdog_test_t *pWatchdogBackup);
```

**Function inputs:**

*\*pWatchdogBackup* - The pointer to the structure with `fs_wdog_test_t` variables.

**Function output:**

*void*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The watchdog timer and the LPTMR must be configured correctly. A variable of the `fs_wdog_test` type must be declared and placed into the RAM area that is not overwritten during the application startup. Interrupts should be disabled.

## 11.2.6 FS\_WDOG\_Setup\_WWDT\_LPC\_mrt()

This function can be used for the LPC devices with WWDT and MRT. This function clears the reset counter, which is a member of the `fs_wdog_test_t` structure. It refreshes the watchdog to start counting from zero. It starts the MRT, which must be configured before the function call occurs. Within the waiting endless loop, the value from the MRT is periodically stored in the reserved area in the RAM.

**Function prototype:**

```
void FS_WDOG_Setup_WWDT_LPC_mrt(fs_wdog_test_t *pWatchdogBackup, uint32_t *pMRTimerBase, uint8_t channel);
```

**Function inputs:**

*\*pWatchdogBackup* - The pointer to the structure with fs\_wdog\_test\_t variables.

*\*pMRTimerBase* - The pointer to the Ctimer used.

*channel* - The channel index of the MRT timer.

**Function output:**

*void*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The watchdog timer and the MRT must be configured correctly. A variable of the fs\_wdog\_test type must be declared and placed into the RAM area that is not overwritten during application startup. Interrupts should be disabled.

## 11.2.7 FS\_WDOG\_Setup\_WWDT\_LPC()

This function can be used for the LPC devices with WWDT. This function clears the reset counter, which is a member of the fs\_wdog\_test\_t structure. It refreshes the watchdog to start counting from zero. It starts the CTimer, which must be configured before the function call occurs. Within the waiting endless loop, the value from the CTimer is periodically stored in the reserved area in the RAM.

**Function prototype:**

```
void FS_WDOG_Setup_WWDT_LPC(fs_wdog_test_t *pWatchdogBackup, uint32_t *pCtimerBase);
```

**Function inputs:**

*\*pWatchdogBackup* - The pointer to the structure with fs\_wdog\_test\_t variables.

*\*pCtimerBase* - The pointer to the CTIMER base address.

**Function output:**

*void*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The watchdog timer and the Ctimer must be configured correctly. A variable of the fs\_wdog\_test type must be declared and placed into the RAM area that is not overwritten during application startup. Interrupts should be disabled.

## 11.2.8 FS\_WDOG\_Setup\_RT()

This function can be used for the MIMXRT10xx devices. This function clears the reset counter, which is a member of the fs\_wdog\_test\_t structure. It refreshes the watchdog to start counting from zero. It starts the GPT, which must be configured before the function call occurs. Within the waiting endless loop, the value from the GPT is periodically stored in the reserved area in the RAM.

**Function prototype:**

```
void FS_WDOG_Setup_RT(fs_wdog_test_t *pWatchdogBackup, uint32_t *pGPT);
```

**Function inputs:**

*\*pWatchdogBackup* - The pointer to the structure with fs\_wdog\_test\_t variables.

*\*pGPT* - The pointer to the GPT base address.

Watchdog test

**Function output:**

*void*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The watchdog timer and the GPT 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.

## 11.2.9 FS\_CM0\_WDOG\_Setup\_K32W()

This function can be used for the CM0+ core on MK32W0x devices. This function clears the reset counter, which is a member of the `fs_wdog_test_t` structure. It refreshes the watchdog to start counting from zero. It starts the LPTMR, which must be configured before the function call occurs. Within the waiting endless loop, the value from the LPTMR is periodically stored in the reserved area in the RAM.

**Function prototype:**

```
void FS_CM0_WDOG_Setup_K32W(fs_wdog_test_t *pWatchdogBackup, uint32_t *pWDOG, uint32_t *pLPTMR);
```

**Function inputs:**

*\*pWatchdogBackup* - The pointer to the structure with `fs_wdog_test_t` variables.

*\*pWDOG* - The pointer to the WDOG base address.

*\*pLPTMR* - The pointer to the LPTMR base address.

**Function output:**

*void*

**Function performance:**

The information about the function performance is in [Core self-test library – source code version](#).

**Calling restrictions:**

The watchdog timer and the LPTMR 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.

## 11.2.10 FS\_CM4\_WDOG\_Setup\_K32W()

This function can be used for the CM4 core of MK32W0x devices. This function compares the captured value of the LPTMR counter with precalculated limit values and checks whether the watchdog reset counter overflows. If the function is called after a non-watchdog reset, `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 1 or 0). If enabled, the function ends up in an endless loop under these conditions:

- 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_CM4_WDOG_Setup_K32W(fs_wdog_test_t *pWatchdogBackup, uint32_t *pWDOG, uint32_t *pLPTMR);
```

**Function inputs:**

*\*pWatchdogBackup* - The pointer to the structure with `fs_wdog_test_t` variables.

*\*pWDOG* - The pointer to the WDOG base address.

*\*pLPTMR* - The pointer to the LPTMR base address.



**Function output:***void***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.

## 11.2.11 FS\_WDOG\_Check()

This function can be used for the MKV1x and MKLxx devices. This function compares the captured value of the LPTMR counter with precalculated limit values and checks whether the watchdog reset counter overflows. If the function is called after a non-watchdog reset, `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 1 or 0). If enabled, the function ends up in an endless loop under these conditions:

- 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_WDOG_Check(uint32_t limitHigh, uint32_t limitLow, uint32_t limitResets, bool_t endlessLoopEnable, fs_wdog_test_t *pWatchdogBackup);
```

**Function inputs:**

*limitHigh* - The precalculated limit value for the reference counter.

*limitLow* - The precalculated limit value for the reference counter.

*limitResets* - The limit value for watchdog resets.

*endlessLoopEnable* - Enables or disables the endless loop within the function.

*\*pWatchdogBackup* - The pointer to the structure with `fs_wdog_test_t` variables.

**Function output:***void***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.

## 11.2.12 FS\_WDOG\_Check\_KE0XZ()

This function can be used for MKE0XZ devices. This function compares the captured value of the RTC counter with precalculated limit values and checks whether the watchdog reset counter overflows. If the function is called after a non-watchdog reset, `wd_test_uncomplete_flag` is set. The endless loop within the function is enabled or disabled with the `endless_loop_enable` parameter (by setting it to 1 or 0). If enabled, the function ends up in an endless loop under these conditions:

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

**Function prototype:**

Watchdog test

```
void FS_WDOG_Check_KE0XZ(uint32_t limitHigh, uint32_t limitLow, uint32_t limitResets, bool_t endlessLoopEnable, fs_wdog_test_t *pWatchdogBackup);
```

**Function inputs:**

*limitHigh* - The precalculated limit value for the reference counter.

*limitLow* - The precalculated limit value for the reference counter.

*limitResets* - The limit value for the watchdog resets.

*endlessLoopEnable* - Enable or disable the endless loop within the function.

*\*pWatchdogBackup* - The pointer to the structure with fs\_wdog\_test\_t variables.

**Function output:**

*void*

**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.

## 11.2.13 FS\_WDOG\_Check\_KE1XF()

This function can be used for the KE1xF devices. This function compares the captured value of the LPTMR counter with precalculated limit values and checks whether the watchdog reset counter overflows. If the function is called after a non-watchdog reset, wd\_test\_uncomplete\_flag is set. The endless loop within the function is enabled or disabled with the endless\_loop\_enable parameter (by setting it to 1 or 0). If enabled, the function ends up in an endless loop under these conditions:

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

**Function prototype:**

```
void FS_WDOG_Check_KE1XF(uint32_t limitHigh, uint32_t limitLow, uint32_t limitResets, bool_t endlessLoopEnable, fs_wdog_test_t *pWatchdogBackup);
```

**Function inputs:**

*limitHigh* - The precalculated limit value for the reference counter.

*limitLow* - The precalculated limit value for the reference counter.

*limitResets* - The limit value for the watchdog resets.

*endlessLoopEnable* - Enable or disable the endless loop within the function.

*\*pWatchdogBackup* - The pointer to the structure with fs\_wdog\_test\_t variables.

**Function output:**

*void*

**Function performance:**

The information about the function performance is mentioned in [Core self-test library – source code version](#).

**Calling restrictions:**

The respective setup function must be executed first.

## 11.2.14 FS\_WDOG\_Check\_KE1XZ()

This function can be used for MKE1xZ devices. This function compares the captured value of the LPTMR counter with precalculated limit values and checks whether the watchdog reset counter overflows. If the function is called after a non-watchdog reset, `wd_test_uncomplete_flag` is set. The endless loop within the function is enabled or disabled with the `endless_loop_enable` parameter (by setting it to 1 or 0). If enabled, the function ends up in an endless loop under these conditions:

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

### Function prototype:

```
void FS_WDOG_Check_KE1XZ(uint32_t limitHigh, uint32_t limitLow, uint32_t limitResets, bool_t endlessLoopEnable,
fs_wdog_test_t *pWatchdogBackup);
```

### Function inputs:

*limitHigh* - The precalculated limit value for the reference counter.

*limitLow* - The precalculated limit value for the reference counter.

*limitResets* - The limit value for watchdog resets.

*endlessLoopEnable* - Enable or disable the endless loop within the function.

*\*pWatchdogBackup* - The pointer to the structure with `fs_wdog_test_t` variables.

### Function output:

*void*

### 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.

## 11.2.15 FS\_WDOG\_Check\_WWDT\_LPC()

This function can be used for the LPC8xx devices. This function compares the captured value of the target counter with precalculated limit values and checks whether the watchdog reset counter overflows. If the function is called after a non-watchdog reset, `wd_test_uncomplete_flag` is set. The endless loop within the function is enabled or disabled with the `endless_loop_enable` parameter (by setting it to 1 or 0). If enabled, the function ends up in an endless loop under these conditions:

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

### Function prototype:

```
void FS_WDOG_Check_WWDT_LPC(uint32_t limitHigh, uint32_t limitLow, uint32_t limitResets, bool_t endlessLoopEnable,
fs_wdog_test_t *pWatchdogBackup);
```

### Function inputs:

*limitHigh* - The precalculated limit value for the reference counter.

*limitLow* - The precalculated limit value for the reference counter.

*limitResets* - The limit value for watchdog resets.

*endlessLoopEnable* - Enable or disable endless loop within the function.

Watchdog test

*\*pWatchdogBackup* - The pointer to the structure with *fs\_wdog\_test\_t* variables.

**Function output:**

*void*

**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.

## 11.2.16 FS\_WDOG\_Check\_RT()

This function can be used for MIMXRT10xx devices. This function compares the captured value of the GPT counter with precalculated limit values and checks whether the watchdog reset counter overflows. If the function is called after a non-watchdog reset, *wd\_test\_uncomplete\_flag* is set. The endless loop within the function is enabled or disabled with the *endless\_loop\_enable* parameter (by setting it to 1 or 0). If enabled, the function ends up in an endless loop under these conditions:

- 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\_WDOG\_Check\_RT(uint32\_t limitHigh, uint32\_t limitLow, uint32\_t limitResets, bool\_t endlessLoopEnable, fs\_wdog\_test\_t \*pWatchdogBackup);*

**Function inputs:**

*limitHigh* - The precalculated limit value for the reference counter.

*limitLow* - The precalculated limit value for the reference counter.

*limitResets* - The limit value for the watchdog resets.

*endlessLoopEnable* - Enable or disable the endless loop within the function.

*\*pWatchdogBackup* - The pointer to the structure with *fs\_wdog\_test\_t* variables.

**Function output:**

*void*

**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.

## 11.2.17 FS\_CM0\_WDOG\_Check\_K32W()

This function can be used for the CM0+ core on MK32W0x devices. This function compares the captured value of the LPTMR 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 1 or 0). If it is enabled, the function ends up in an endless loop under these conditions:

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

**Function prototype:**

```
void FS_CM0_WDOG_Check_K32W(uint32_t limitHigh, uint32_t limitLow, uint32_t limitResets, bool_t endlessLoopEnable,
fs_wdog_test_t *pWatchdogBackup);
```

**Function inputs:**

*limitHigh* - The precalculated limit value for the reference counter.

*limitLow* - The precalculated limit value for the reference counter.

*limitResets* - The limit value for the watchdog resets.

*endlessLoopEnable* - Enables or disables the endless loop within the function.

*\*pWatchdogBackup* - The pointer to a structure with the fs\_wdog\_test\_t variables.

**Function output:**

*void*

**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.

## 11.2.18 FS\_CM4\_WDOG\_Check\_K32W()

This function can be used for the CM4 core of MK32W0x devices. This function compares the captured value of the LPTMR 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 1 or 0). If enabled, the function ends up in an endless loop under these conditions:

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

**Function prototype:**

```
void FS_CM4_WDOG_Check_K32W(uint32_t limitHigh, uint32_t limitLow, uint32_t limitResets, bool_t endlessLoopEnable,
fs_wdog_test_t *pWatchdogBackup);
```

**Function inputs:**

*limitHigh* - The precalculated limit value for the reference counter.

*limitLow* - The precalculated limit value for the reference counter.

*limitResets* - The limit value for the watchdog resets.

*endlessLoopEnable* - Enable or disable the endless loop within the function.

*\*pWatchdogBackup* - The pointer to the structure with fs\_wdog\_test\_t variables.

**Function output:**

*void*

**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.

## 11.2.19 FS\_WDOG\_Check\_WWDT\_LPC55SXX()

This function can be used for the LPC55Sxx devices. This function compares the captured value of the target counter with precalculated limit values and checks whether the watchdog reset counter overflows. If the function is called after a non-watchdog reset, `wd_test_uncomplete_flag` is set. The endless loop within the function is enabled or disabled with the `endless_loop_enable` parameter (by setting it to 1 or 0). If enabled, the function ends up in an endless loop under these conditions:

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

### **Function prototype:**

```
void FS_WDOG_Check_WWDT_LPC55SXX(uint32_t limitHigh, uint32_t limitLow, uint32_t limitResets, bool_t  
endlessLoopEnable, fs_wdog_test_t *pWatchdogBackup);
```

### **Function inputs:**

*limitHigh* - The precalculated limit value for the reference counter.

*limitLow* - The precalculated limit value for the reference counter.

*limitResets* - The limit value for watchdog resets.

*endlessLoopEnable* - Enable or disable the endless loop within the function.

*\*pWatchdogBackup* - The pointer to the structure with `fs_wdog_test_t` variables.

### **Function output:**

*void*

### **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](http://nxp.com)

### Web Support:

[nxp.com/support](http://nxp.com/support)

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: [nxp.com/SalesTermsandConditions](http://nxp.com/SalesTermsandConditions).

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, Altivec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, UMEMS, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro,  $\mu$ Vision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© NXP B.V. 2020.

All rights reserved.

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

For sales office addresses, please send an email to: [salesaddresses@nxp.com](mailto:salesaddresses@nxp.com)

Date of release: 01/2020

Document identifier: IEC60730BCM0LUG40

