Document identifier: IEC60730BCM33L41UG
Rev. 0, 12/2020

IEC60730_B_CM33_Library_UG_v4_1

IEC60730B Library User's Guide

Contents

Chapter 1 Core self-test library....................3

Chapter 2 Analog Input/Output (IO) test.................... 10

Chapter 3 Clock test.................... 22

Chapter 4 Digital input/output test....................27

Chapter 5 Invariable memory test....................57

Chapter 6 CPU program counter test.................... 64

Chapter 7 Variable memory test.................... 68

Chapter 8 CPU register test....................75

Chapter 9 Stack test.................... 93

Chapter 10 TSI tests....................96

Chapter 11 Watchdog test 106

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:

Core dependent part

- CPU registers test
- CPU program counter test
- Variable memory test
- Invariable memory test
- Stack test

Peripheral dependent part

- Clock test
- Digital input/output test
- Analog input/output test
- Watchdog test
- Touch-sensing interface test (only for TSIv5 peripheral)

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

The library supports the LPC55Sxx and LPC55xx families based on the Arm-M33 core.

The core self-test library is disributed as an object code version. For the source code, contact an NXP representative.

1.1 Core self-test library – object code

The object code of the library is divided to two parts: the core-dependent part and the peripheral dependent part with the corresponding header file.

The following are the object files for the given IDEs:

Table 1. Library object code

IDE	Part	Object file
IAR	Core	*IEC60730B_M33_IAR_v4_1.a*
	Peripheral	*IEC60730B_M33_COM_IAR_v4_1.a*
Keil	Core	*IEC60730B_M33_Keil_v4_1.lib*
	Peripheral	*IEC60730B_M33_COM_Keil_v4_1.lib*
MCUX	Core	*libIEC60730B_M33_MCUX_v4_1.a*
	Peripheral	*libIEC60730B_M33_COM_MCUX_v4_1.a*

1.2 Core self-test library – source code

The library name is IEC60730B_CM33 . The main header files are *iec60730b.h* and *iec60730b_core.h*. All necessary data types for the library are defined in *iec60730b_types.h*.

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

Table 2. List of library items

File Name	Test Type	Function Name	Functions size [bytes]	Functions duration [µs]
iec60730b.h	Library header file	-		
iec60730b_core.h	Core depend library header file	-		
iec60730b_types.h	Data types for library	-		
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()	29[1]	0.73[1]
	Analog I/O test	FS_AIO_InputInit_LPC_ADC12()	-	-
	Analog I/O test	FS_AIO_InputInit_IMXRT117X()	-	-
	Analog I/O test	FS_AIO_InputTrigger()	12[1]	0.16[1]
	Analog I/O test	FS_AIO_InputSet()	-	-
	Analog I/O test	FS_AIO_InputSet_KE()	-	-
	Analog I/O test	FS_AIO_InputSet_CYCLIC()	-	-
	Analog I/O test	FS_AIO_InputSet_LPC8XX()	-	-
	Analog I/O test	FS_AIO_InputSet_LPC55SXX()	36[1]	0.68[1]
	Analog I/O test	FS_AIO_InputSet_IMXRT10XX_SWTRIG()	-	-
	Analog I/O test	FS_AIO_InputSet_IMXRT117X_SWTRIG()	-	-
	Analog I/O test	FS_AIO_InputCheck()	-	-
	Analog I/O test	FS_AIO_InputCheck_KE()	-	-
	Analog I/O test	FS_AIO_InputCheck_CYCLIC()	-	-
	Analog I/O test	FS_AIO_InputCheck_LPC8XX()	-	-
	Analog I/O test	FS_AIO_InputCheck_LPC55SXX()	120[1]	0.53[1]
	Analog I/O test	FS_AIO_InputCheck_IMXRT10XX_SWTRIG()	-	-
	Analog I/O test	FS_AIO_InputCheck_IMXRT117X()	-	-

Table continues on the next page...

Table 2. List of library items (continued)

File Name	Test Type	Function Name	Functions size [bytes]	Functions duration [µs]
iec60730b_clock.c	Clock test	FS_CLK_Check()	38[1]	0.33[1]
	Clock test	FS_CLK_Init()	8[1]	0.14[1]
	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_CTIMER_LPC()	24[1]	12.04[1]
iec60730b_dio.c	Digital I/O test	FS_DIO_Input()	-	-
	Digital I/O test	FS_DIO_Output()	-	-
	Digital I/O test	FS_DIO_Output_IMXRT()	-	-
	Digital I/O test	FS_DIO_Output_IMX8M()	-	-
	Digital I/O test	FS_DIO_Output_LPC()	128[1]	15.59[1]
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_IMXRT()	-	-
	Extended Digital I/O test	FS_DIO_ShortToSupplySet_IMXRT()	-	-
	Extended Digital I/O test	FS_DIO_ShortToAdjSet_IMXRT()	-	-
	Extended Digital I/O test	FS_DIO_InputExt_IMX8M()	-	-
	Extended Digital I/O test	FS_DIO_ShortToSupplySet_IMX8M()	-	-
	Extended Digital I/O test	FS_DIO_ShortToAdjSet_IMX8M()	-	-
	Extended Digital I/O test	FS_DIO_InputExt_LPC()	150[1]	1.62[1]
	Extended Digital I/O test	FS_DIO_ShortToSupplySet_LPC()	110[1]	1.17[1]
	Extended Digital I/O test	FS_DIO_ShortToAdjSet_LPC()	184[1]	1.83[1]

Table continues on the next page...

Table 2. List of library items (continued)

File Name	Test Type	Function Name	Functions size [bytes]	Functions duration [µs]
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_cm33_flash.S	Invariable memory test (Flash)	FS_CM33_FLASH_HW16()	See the function dedicated chapter	
	Invariable memory test (Flash)	FS_CM33_FLASH_HW32()	See the function dedicated chapter	
	Invariable memory test (Flash)	FS_CM33_FLASH_SW16()	See the function dedicated chapter	
	Invariable memory test (Flash)	FS_CM33_FLASH_SW32()	See the function dedicated chapter	
iec60730b_cm33_pc.S	Program Counter test	FS_CM33_PC_Test()	See the function dedicated chapter	
iec60730b_cm33_pc_object.S	Program Counter test	FS_PC_Object()	See the function dedicated chapter	
iec60730b_cm33_ram.S	Variable memory test (RAM)	FS_CM33_RAM_AfterReset()	See the function dedicated chapter	
	Variable memory test (RAM)	FS_CM33_RAM_Runtime()	See the function dedicated chapter	
	Variable memory test (RAM)	FS_CM33_RAM_CopyToBackup()	See the function dedicated chapter	
	Variable memory test (RAM)	FS_CM33_RAM_CopyFromBackup()	See the function dedicated chapter	
	Variable memory test (RAM)	FS_CM33_RAM_SegmentMarchC()	See the function dedicated chapter	
	Variable memory test (RAM)	FS_CM33_RAM_SegmentMarchX()	See the function dedicated chapter	
iec60730b_cm33_reg.S	Register test	FS_CM33_CPU_Register()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_NonStackedRegister()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_Primask_S()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_Primask_NS()	See the function dedicated chapter	

Table continues on the next page...

Table 2. List of library items (continued)

File Name	Test Type	Function Name	Functions size [bytes]	Functions duration [µs]
	Register test	FS_CM33_CPU_SPmain_S()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_SPmain_NS()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_SPmain_Limit_S()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_SPmain_Limit_NS()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_SPprocess_S()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_SPprocess_NS()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_SPprocess_Limit_S()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_SPprocess_Limit_NS()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_Control_S()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_Control_NS()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_Control()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_Special8PriorityLevels_S()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_Special8PriorityLevels_NS()	See the function dedicated chapter	
iec60730b_cm33_reg_fpu.S	Register test	FS_CM33_CPU_Float1()	See the function dedicated chapter	
	Register test	FS_CM33_CPU_Float2()	See the function dedicated chapter	
iec60730b_cm33_stack.S	Stack test	FS_CM33_STACK_Init()	See the function dedicated chapter	
	Stack test	FS_CM33_STACK_Test()	See the function dedicated chapter	
iec60730b_wdog.c	Watchdog test	FS_WDOG_Setup_LPTMR()	-	Duration time depends on the WDOG timeout
	Watchdog test	FS_WDOG_Setup_KE0XZ()	-	Duration time depends on the WDOG timeout
	Watchdog test	FS_WDOG_Setup_IMX_GPT()	-	Duration time depends on the WDOG timeout
	Watchdog test	FS_WDOG_Setup_WWDT_LPC()	52[1]	Duration time depends on the WDOG timeout
	Watchdog test	FS_WDOG_Setup_WWDT_LPC_mrt()	-	Duration time depends on the WDOG timeout
	Watchdog test	FS_WDOG_Check()	-	-

Table continues on the next page...

Table 2. List of library items (continued)

File Name	Test Type	Function Name	Functions size [bytes]	Functions duration [μs]
	Watchdog test	FS_WDOG_Check_WWDT_LPC()	-	-
	Watchdog test	FS_WDOG_Check_WWDT_LPC55SXX()	72[1]	1.35[1]

1.2.1 LPC55Sxx dedicated functions

Table 3 shows the list of functions dedicated for the LPC55Sxx device family.

Table 3. LPC55Sxx dedicated functions

File	Suitable function
iec60730b_aio.c	FS_AIO_InputInit_LPC_ADC16()
	FS_AIO_InputTrigger()
	FS_AIO_InputSet_LPC55SXX()
	FS_AIO_InputCheck_LPC55SXX()
iec60730b_clock.c	FS_CLK_Check()
	FS_CLK_Init()
	FS_CLK_CTIMER_LPC()
iec60730b_dio.c	FS_DIO_Output_LPC()
iec60730b_dio_ext.c	FS_DIO_InputExt_LPC()
	FS_DIO_ShortToSupplySet_LPC()
	FS_DIO_ShortToAdjSet_LPC()
iec60730b_wdog.c	FS_WDOG_Setup_WWDT_LPC()
	FS_WDOG_Check_WWDT_LPC55SXX()
iec60730b_cm33_flash.S	FS_CM33_FLASH_HW16()
	FS_CM33_FLASH_HW32()
	FS_CM33_FLASH_SW16()
	FS_CM33_FLASH_SW32()
iec60730b_cm33_pc.S	Common for all CM33 devices
iec60730b_cm33_ram.S	Common for all CM33 devices
iec60730b_cm33_reg.S	Common for all CM33 devices
Common functions	FS_CM33_CPU_Register()
	FS_CM33_CPU_NonStackedRegister()
	FS_CM33_CPU_Float1()
	FS_CM33_CPU_Float2()
Devices with TrustZone support:	FS_CM33_CPU_Primask_S()

Table continues on the next page...

Table 3. LPC55Sxx dedicated functions (continued)

File	Suitable function
	FS_CM33_CPU_Primask_NS()
	FS_CM33_CPU_SPmain_S()
	FS_CM33_CPU_SPmain_NS()
	FS_CM33_CPU_SPmain_Limit_S()
	FS_CM33_CPU_SPmain_Limit_NS()
	FS_CM33_CPU_SPprocess_S()
	FS_CM33_CPU_SPprocess_NS()
	FS_CM33_CPU_SPprocess_Limit_S()
	FS_CM33_CPU_SPprocess_Limit_NS()
	FS_CM33_CPU_Control_S()
	FS_CM33_CPU_Control_NS()
	FS_CM33_CPU_Special8PriorityLevels_S()
	FS_CM33_CPU_Special8PriorityLevels_NS()
Devices without TrustZone support:	FS_CM33_CPU_Primask_S()
	FS_CM33_CPU_SPmain_S()
	FS_CM33_CPU_SPmain_Limit_S()
	FS_CM33_CPU_SPprocess_S()
	FS_CM33_CPU_SPprocess_Limit_S()
	FS_CM33_CPU_Control()
	FS_CM33_CPU_Special8PriorityLevels_S()
iec60730b_cm33_stack.S	Common for all CM33 devices

1.3 Functions performance measurement

This section contains remarks about the functons' 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 LPC55S69 with a clock frequency of 96 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:

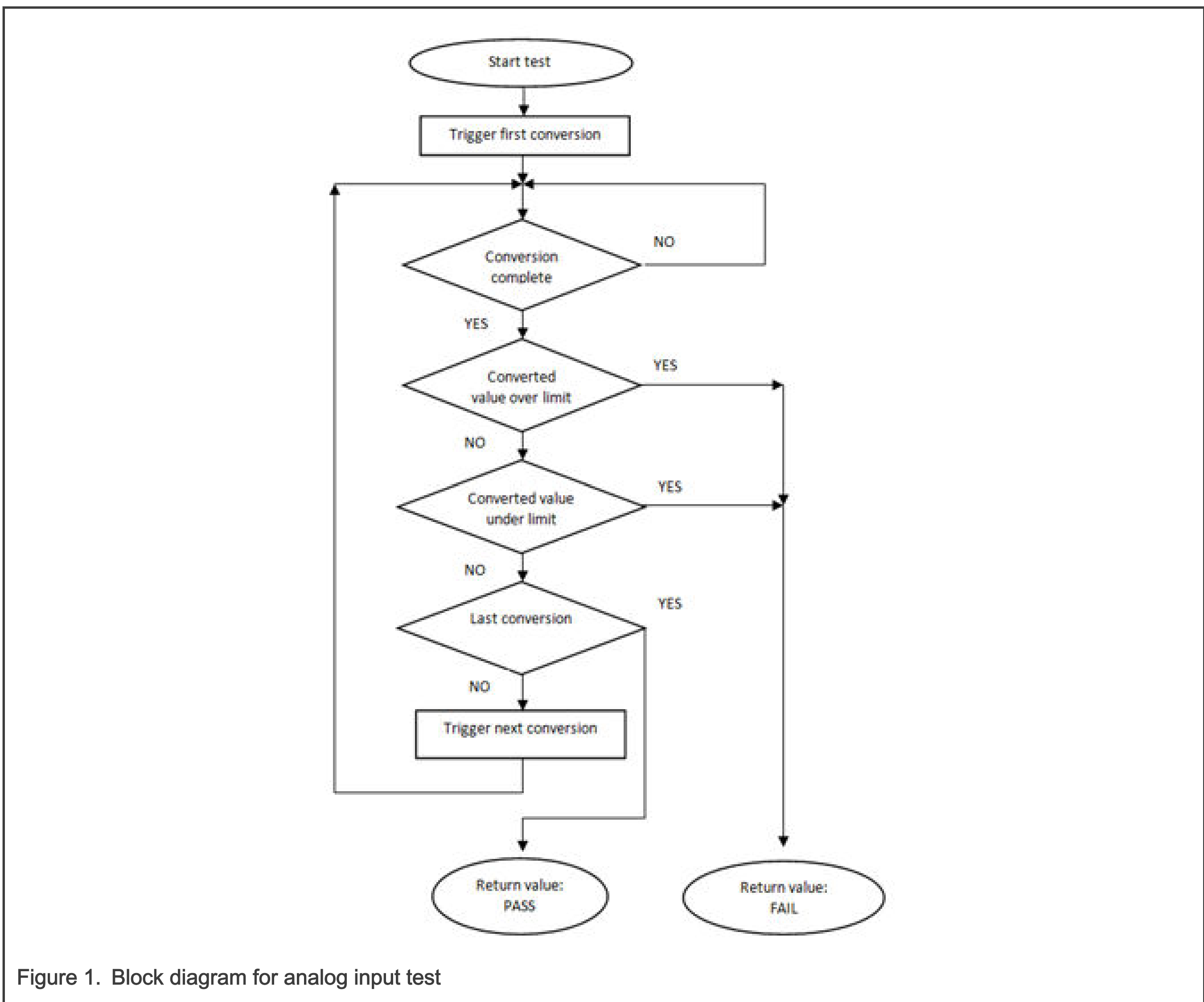

Figure 1. Block diagram for analog input test

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 4. 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 in the *iec60730b_aio.c* file and written as "C" functions. The header file with the function prototypes is *iec60730b_aio.h*. *iec60730b.h* and *iec60730b_types.h* are the common header files for the safety library.

The following functions are called to test the analog input:

- *FS_AIO_InputInit () / FS_AIO_InputInit_CYCLIC() / FS_AIO_InputInit_LPC_ADC16() / FS_AIO_InputInit_LPC_ADC12() / FS_AIO_InputInit_IMXRT117X()*
- *FS_AIO_InputTrigger()*
- *FS_AIO_InputSet() / FS_AIO_InputSet_KE() / FS_AIO_InputSet_CYCLIC() / FS_AIO_InputSet_LPC8XX() / FS_AIO_InputSet_LPC55SXX() / FS_AIO_InputSet_IMXRT10XX_SWTRIG() / FS_AIO_InputSet_IMXRT117X_SWTRIG()*
- *FS_AIO_InputCheck() / FS_AIO_InputCheck_KE() / FS_AIO_InputCheck_CYCLIC() /FS_AIO_InputCheck_LPC8XX() / FS_AIO_InputCheck_LPC55SXX() / FS_AIO_InputCheck_IMXRT10XX_SWTRIG() / FS_AIO_InputCheck_IMXRT117X( )*

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_INIT\
{\
 {ADC_MIN_LIMIT(0), ADC_MAX_LIMIT(10)}, \
 {ADC_MIN_LIMIT(ADC_MAX), ADC_MAX_LIMIT(ADC_MAX)},\
 {ADC_MIN_LIMIT(ADC_BANDGAP_LEVEL_RAW), ADC_MAX_LIMIT(ADC_BANDGAP_LEVEL_RAW)}\
}
#define FS_CFG_AIO_CHANNELS_INIT {30, 29, 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_FAIL_AIO:
 psSafetyCommon->ui32SafetyErrors |= AIO_TEST_ERROR;
 SafetyErrorHandling(psSafetyCommon);
 break;
 case FS_AIO_INIT:
 FS_AIO_InputTrigger(&aio_Str);
 break;
 case FS_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.

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

Function output:

void

Function performance:

For information about the function performance, see 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 limits arrays.

sequence - No effect due to compatibility.

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

Function output:

void

Function performance:

For information about the function performance, see 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 limits arrays.

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

Function output:

void

Function performance:

For information about the function performance, see Core self-test library – source code version.

2.2.6 FS_AIO_InputInit_IMXRT117X()

This function initializes one instance of the analog input test.

Function prototype:

*void FS_AIO_InputInit_IMXRT117X(fs_aio_test_t *pObj, fs_aio_limits_t *pLimits, uint8_t *pInputs, uint8_t cntMax, uint16_t commandBuffer, uint8_t fifo, 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 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.

commandBuffer - The input argument is number of command.

fifo - The input argument is number of trigger register.

sequence - The input argument is index of used sequence - trigger source.

Function output:

void

Function performance:

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

2.2.7 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, fs_aio_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 eans 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()

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_KE(fs_aio_test_t *pObj, fs_aio_ke_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 eans 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_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, fs_aio_cyclic_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_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, fs_aio_lpc8xx_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.

2.2.11 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, fs_aio_lpc55sxx_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.

2.2.12 FS_AIO_InputSet_IMXRT10XX_SWTRIG()

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. On i.MXRT devices, only channel 0 (ADCx->HC[0]) can be used for software triggering, see the reference manual for more information.

Function prototype:

*FS_RESULT FS_AIO_InputSet_IMXRT10XX_SWTRIG(fs_aio_test_t *pObj, fs_aio_imxrt10xx_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:

For information about the function performance, see Core self-test library – source code version.

2.2.13 FS_AIO_InputSet_IMXRT117X_SWTRIG()

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. On i.MXRT117X devices supports channel selection, trigger control register and trigger value selection, see the reference manual for more information.

Function prototype:

*FS_RESULT FS_AIO_InputSet_IMXRT117X_SWTRIG(fs_aio_test_t *pObj, fs_aio_imxrt117x_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:

For information about the function performance, see Core self-test library – source code version.

2.2.14 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_FAIL_AIO.

Function prototype:

*FS_RESULT FS_AIO_InputCheck(fs_aio_test_t *pObj, fs_aio_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 successfull execution of the test (all channels tested).
- *FS_FAIL_AIO* - The converted value does not fit into the limits.
- *FS_AIO_START* - A successfull 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_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_FAIL_AIO.

Function prototype:

*FS_RESULT FS_AIO_InputCheck_CYCLIC(fs_aio_test_t *pObj, fs_aio_cyclic_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 successfull execution of the test (all channels tested).
- *FS_FAIL_AIO* - The converted value does not fit into the limits.
- *FS_AIO_START* - A successfull 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_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_FAIL_AIO.

Function prototype:

*FS_RESULT FS_AIO_InputCheck_KE(fs_aio_test_t *pObj, fs_aio_ke_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 successfull execution of the test (all channels tested).
- *FS_FAIL_AIO* - The converted value does not fit into the limits.
- *FS_AIO_START* - A successfull 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:

For information about the function performance, see Core self-test library – source code version.

2.2.17 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_FAIL_AIO.

Function prototype:

FS_RESULT FS_AIO_InputCheck_LPC55SXX(fs_aio_test_t pObj, fs_aio_lpc55sxx_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 successfull execution of the test (all channels tested).
- *FS_FAIL_AIO* - The converted value does not fit into the limits.
- *FS_AIO_START* - A successfull 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:

For information about the function performance, see Core self-test library – source code version.

2.2.18 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_FAIL_AIO.

Function prototype:

FS_RESULT FS_AIO_InputCheck_LPC8XX(fs_aio_test_t pObj, fs_aio_lpc8xx_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 successfull execution of the test (all channels tested).
- *FS_FAIL_AIO* - The converted value does not fit into the limits.
- *FS_AIO_START* - A successfull 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:

For information about the function performance, see Core self-test library – source code version.

2.2.19 FS_AIO_InputCheck_IMXRT10XX_SWTRIG()

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_FAIL_AIO. This function must be used only with the software-triggered analog I/O test.

Function prototype:

FS_RESULT FS_AIO_InputCheck_IMXRT10XX_SWTRIG(fs_aio_test_t pObj, fs_aio_imxrt10xx_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 successfull execution of the test (all channels tested).
- *FS_FAIL_AIO* - The converted value does not fit into the limits.
- *FS_AIO_START* - A successfull 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:

For information about the function performance, see Core self-test library – source code version.

2.2.20 FS_AIO_InputCheck_IMXRT117X()

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_FAIL_AIO. This function must be used only with the software-triggered analog I/O test.

Function prototype:

FS_RESULT FS_AIO_InputCheck_IMXRT117X(fs_aio_test_t pObj, fs_aio_imxrt117x_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 successfull execution of the test (all channels tested).
- *FS_FAIL_AIO* - The converted value does not fit into the limits.
- *FS_AIO_START* - A successfull 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:

For information about the function performance, see Core self-test library – source code version.

Chapter 3
Clock test

The clock test procedure tests the oscillators of the processor for the wrong frequency. 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 Figure 2.

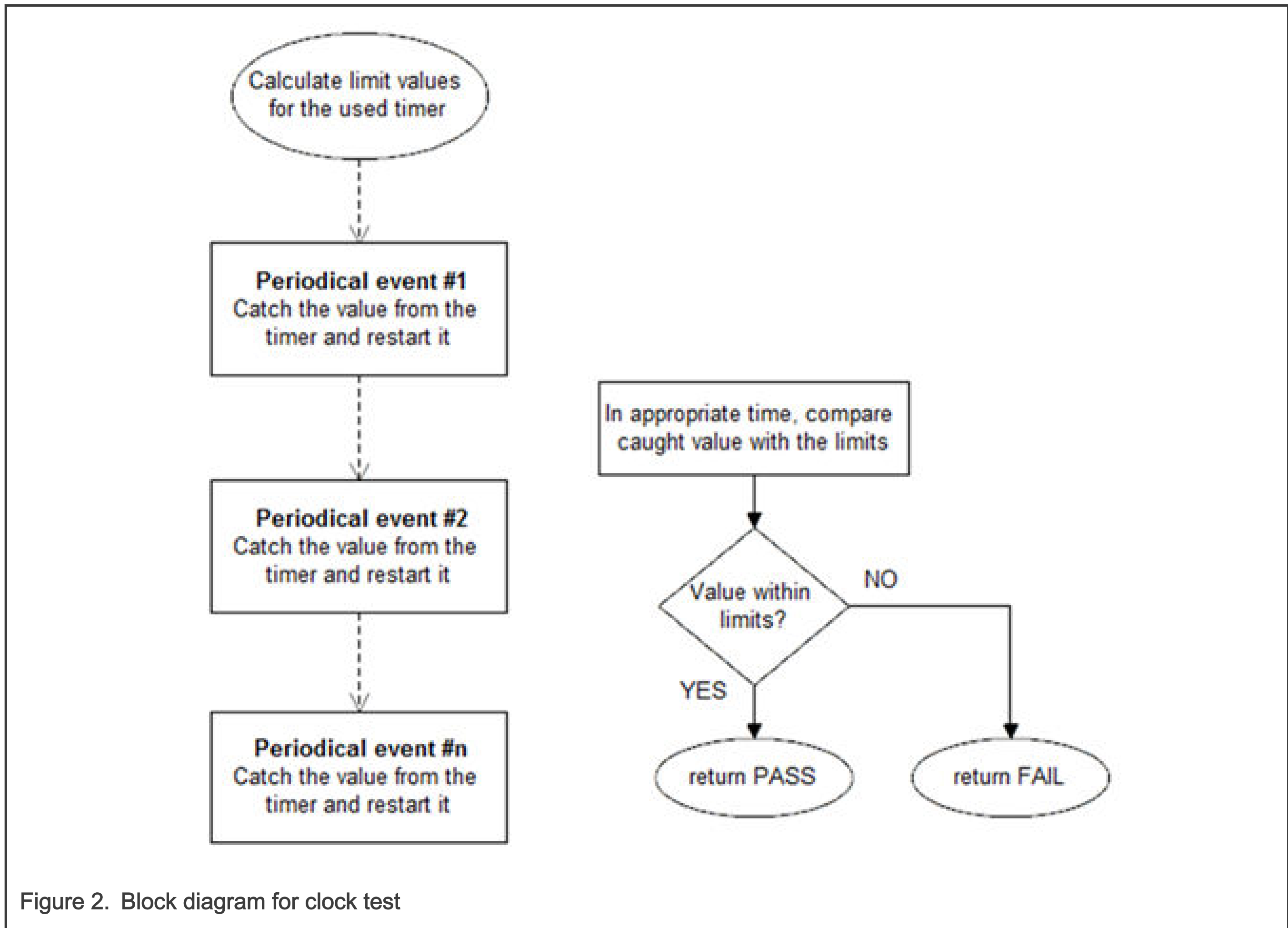

Figure 2. Block diagram for clock test

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 5. 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*. *iec60730b.h* and *iec60730b_types.h* are the common header files for the safety library.

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_CTIMER_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;
#defineISR_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_FAIL_CLK == 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.

<u>Function prototype:</u>

*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 Isr 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(fs_lptmr_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(fs_ctimer_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(fs_gpt_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(fs_rtc_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.

3.2.7 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(fs_wkt_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 Table 6.

Table 6. 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*. *iec60730b.h* and *iec60730b_types.h* are the common header files for the safety library.

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

- *FS_DIO_Input()*
- *FS_DIO_Output() / FS_DIO_Output_IMXRT() / FS_DIO_Output_IMX8M() / FS_DIO_Output_LPC()*
- *FS_DIO_InputExt() / FS_DIO_InputExt_IMXRT() / FS_DIO_InputExt_IMX8M() / FS_DIO_InputExt_LPC()*
- *FS_DIO_ShortToSupplySet() / FS_DIO_ShortToSupplySet_IMXRT() / FS_DIO_ShortToSupplySet_IMX8M() / FS_DIO_ShortToSupplySet_LPC()*
- *FS_DIO_ShortToAdjSet() / FS_DIO_ShortToAdjSet_IMXRT() / FS_DIO_ShortToAdjSet_IMX8M() / 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 = GPIOE_BASE,
 .pcr = PORTE_BASE,
 .pinNum = 24,
 .pinDir = PIN_DIRECTION_IN,
 .pinMux = PIN_MUX_GPIO,
};
fs_dio_test_t dio_safety_test_item_1 =
{
 .gpio = GPIOA_BASE,
 .pcr = PORTA_BASE,
 .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 == GPIOE_BASE)
 dio_safety_test_item_0 .pcr = PORTE_BASE;

if (dio_safety_test_item_1 .gpio == GPIOA_BASE)
 dio_safety_test_item_1 .pcr = PORTA_BASE;
```

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:

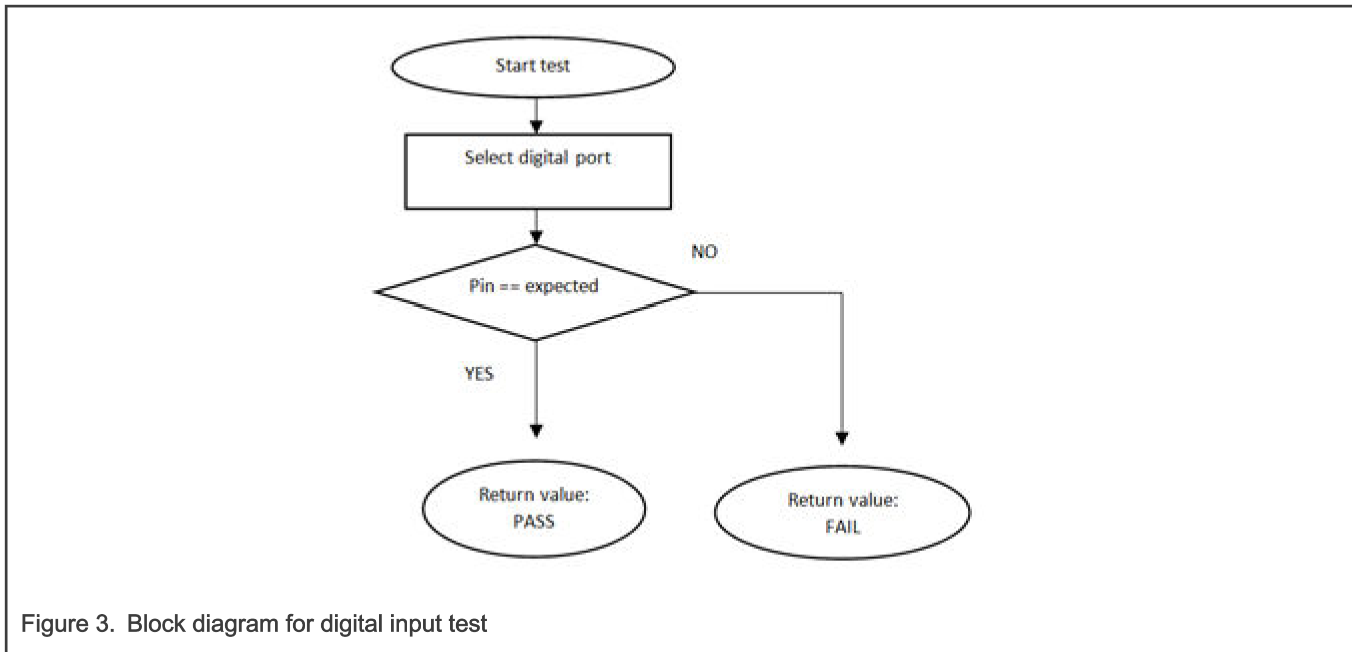

Figure 3. Block diagram for digital input test

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.

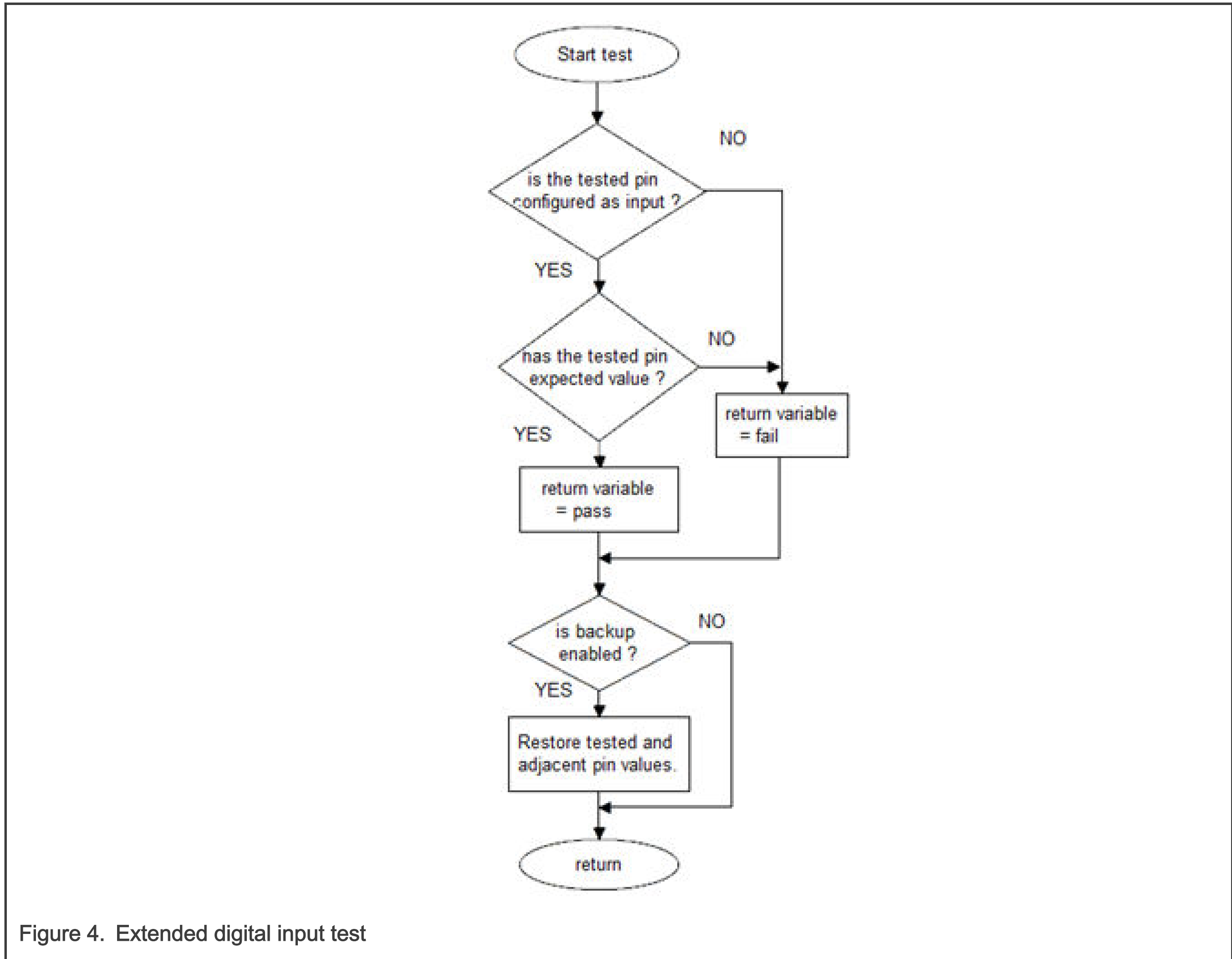

Figure 4. Extended digital input test

<u>Function prototype:</u>

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

<u>Function inputs:</u>

**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.

<u>Function output:</u>

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO*

<u>Example of function call:</u>

fs_dio_input_test_result = FS_DIO_InputExt(&dio_safety_test_item_0, &dio_safety_test_item_0, DIO_EXPECTED_VALUE, BACKUP_ENABLE);

<u>Function performance:</u>

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_IMX8M()

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_IMX8M()* function is shown in Figure 5. 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.

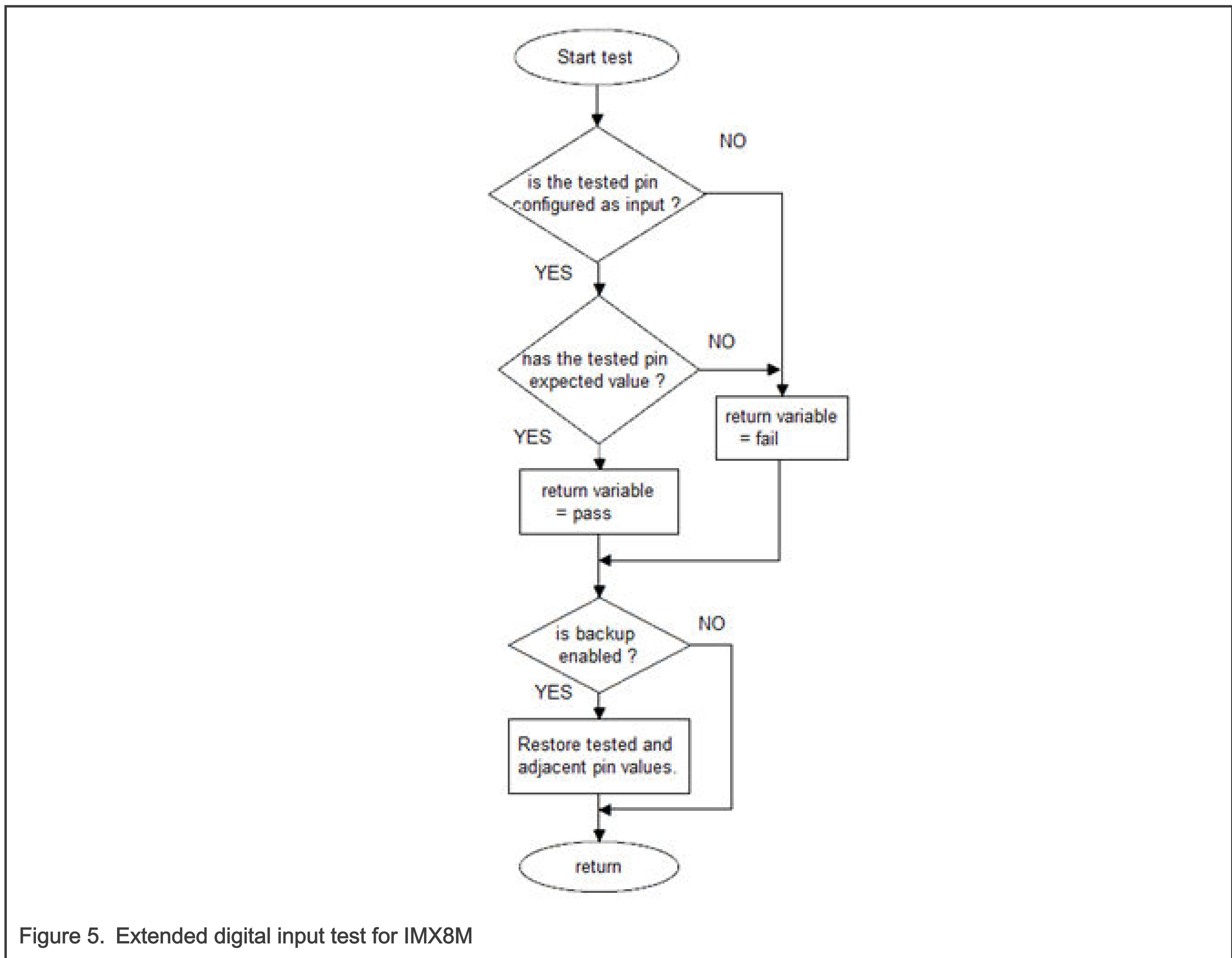

Figure 5. Extended digital input test for IMX8M

Function prototype:

*FS_RESULT FS_DIO_InputExt_IMX8M(fs_dio_test_imx_t *pTestedPin, fs_dio_test_imx_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_IMX8M(&dio_safety_test_item_0, &dio_safety_test_item_0, DIO_EXPECTED_VALUE, BACKUP_ENABLE);

Function performance:

For information about the function performance, see Core self-test library – source code version.

Calling restrictions:

The function can be used only for the i.MX8M 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.

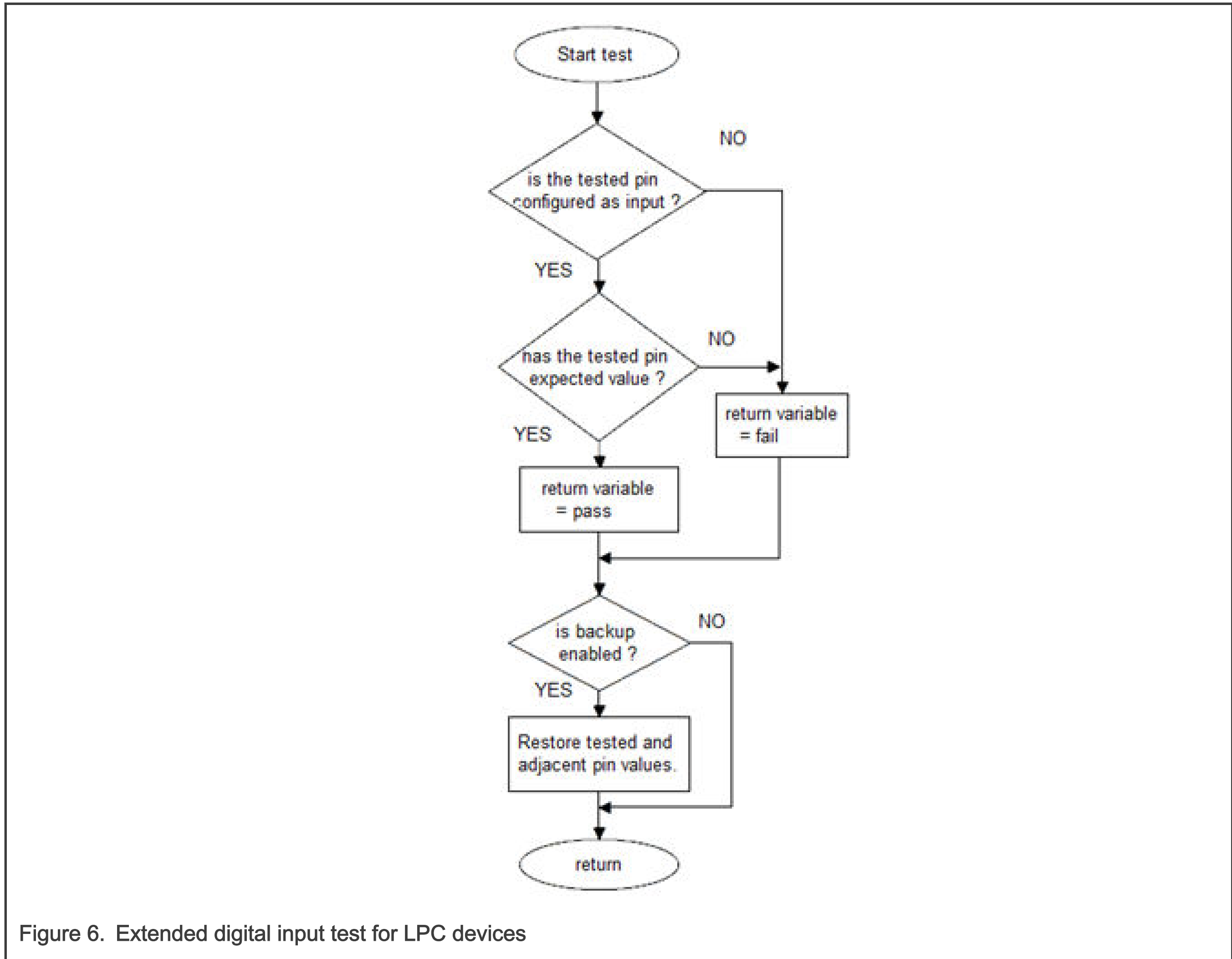

Figure 6. Extended digital input test for LPC devices

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_InputExt_IMXRT()

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_IMXRT()* function is shown in Figure 7. 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.

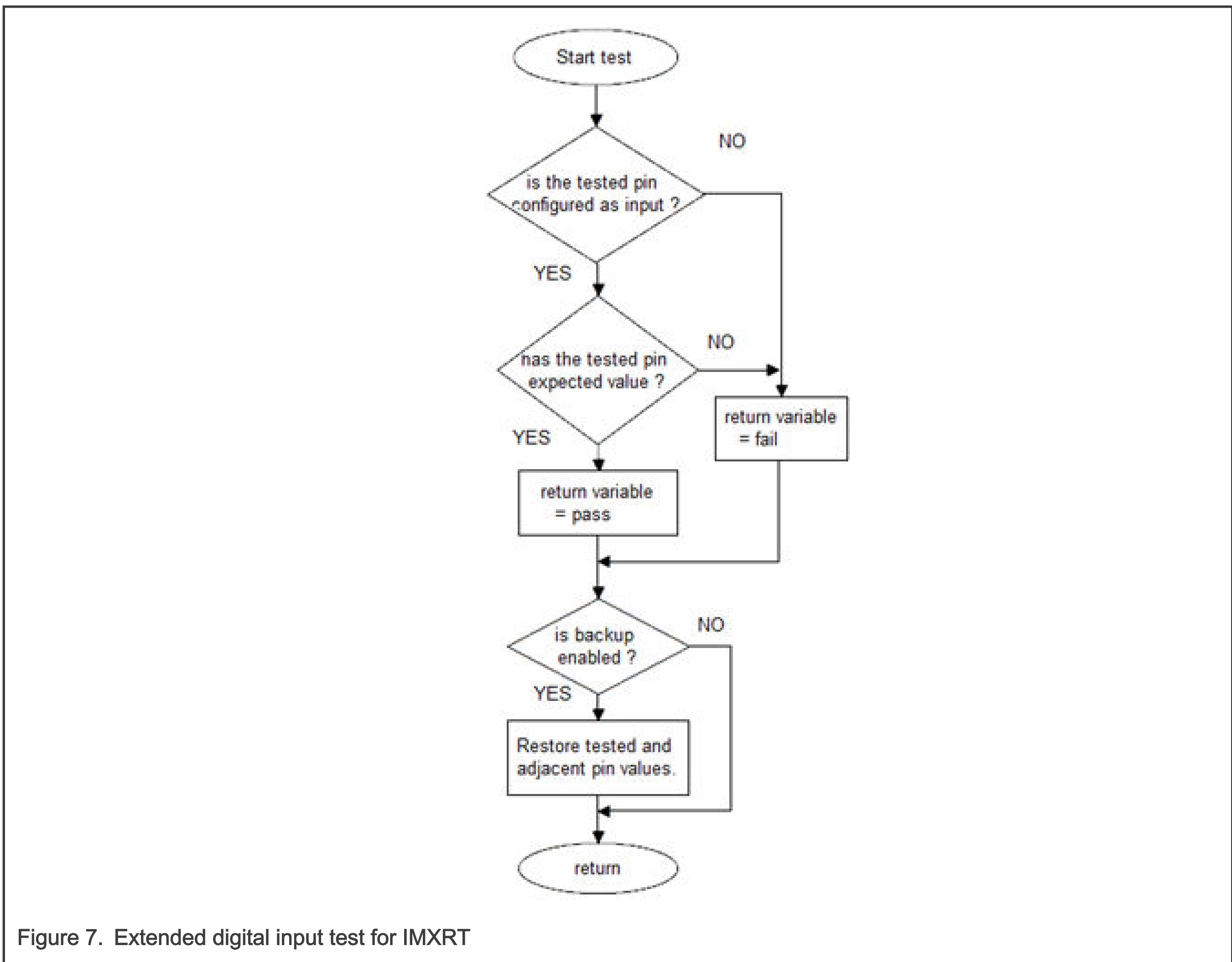

Figure 7. Extended digital input test for IMXRT

Function prototype:

*FS_RESULT FS_DIO_InputExt_IMXRT(fs_dio_test_imx_t *pTestedPin, fs_dio_test_imx_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.

<u>Function output:</u>

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO*

<u>Example of function call:</u>

fs_dio_input_test_result = FS_DIO_InputExt_IMXRT(&dio_safety_test_item_0, &dio_safety_test_item_0, DIO_EXPECTED_VALUE, BACKUP_ENABLE);

<u>Function performance:</u>

For information about the function performance, see Core self-test library – source code version.

<u>Calling restrictions:</u>

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".

4.2.6 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.

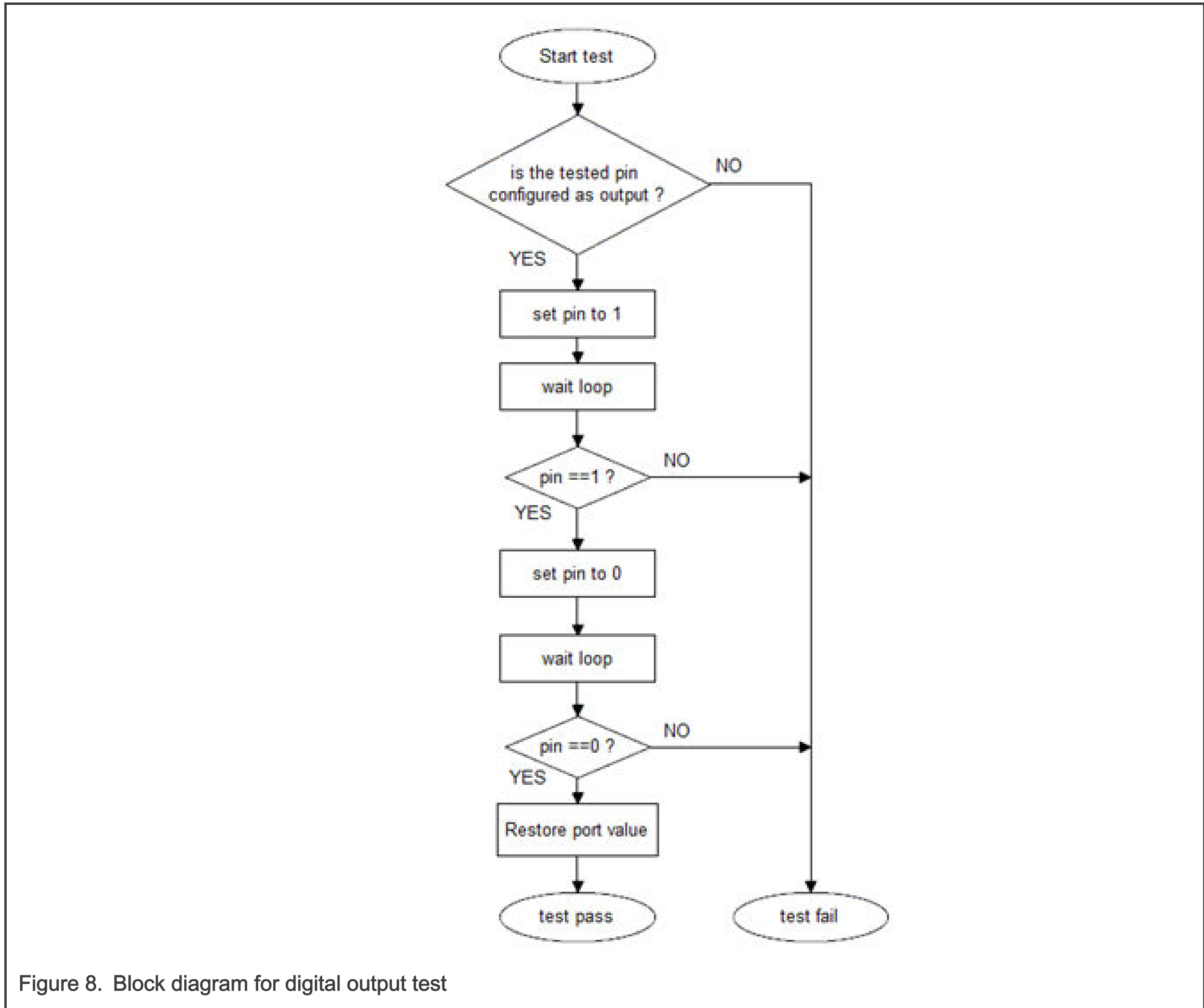

Figure 8. Block diagram for digital output test

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:

For information about the function performance, see 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.

4.2.7 FS_DIO_Output_IMX8M()

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.

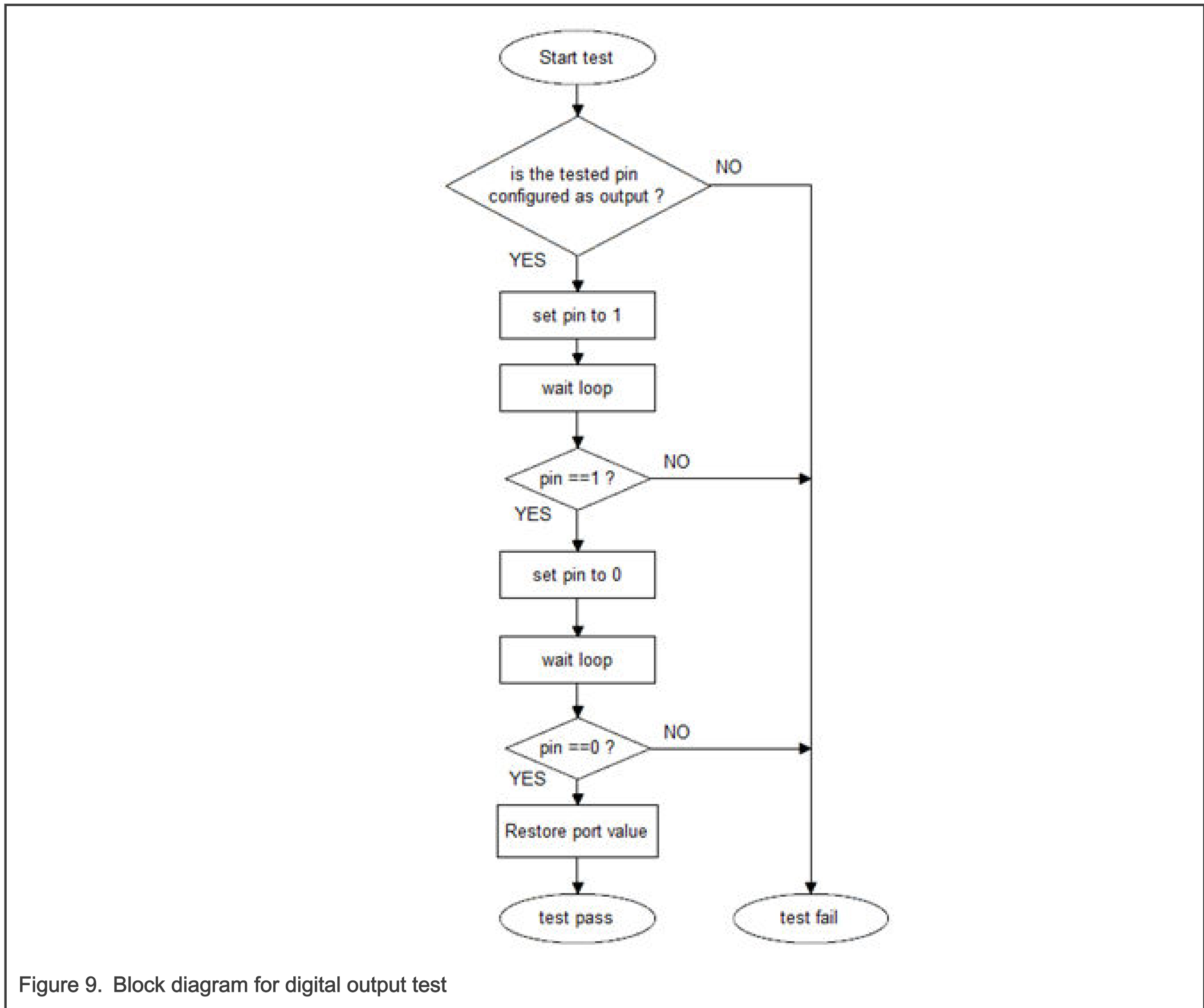

Figure 9. Block diagram for digital output test

Function prototype:

*FS_RESULT FS_DIO_Output_IMX8M(fs_dio_test_imx_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_IMX8M(&dio_safety_test_items[1], DIO_WAIT_CYCLE);

Function performance:

For information about the function performance, see 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_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.

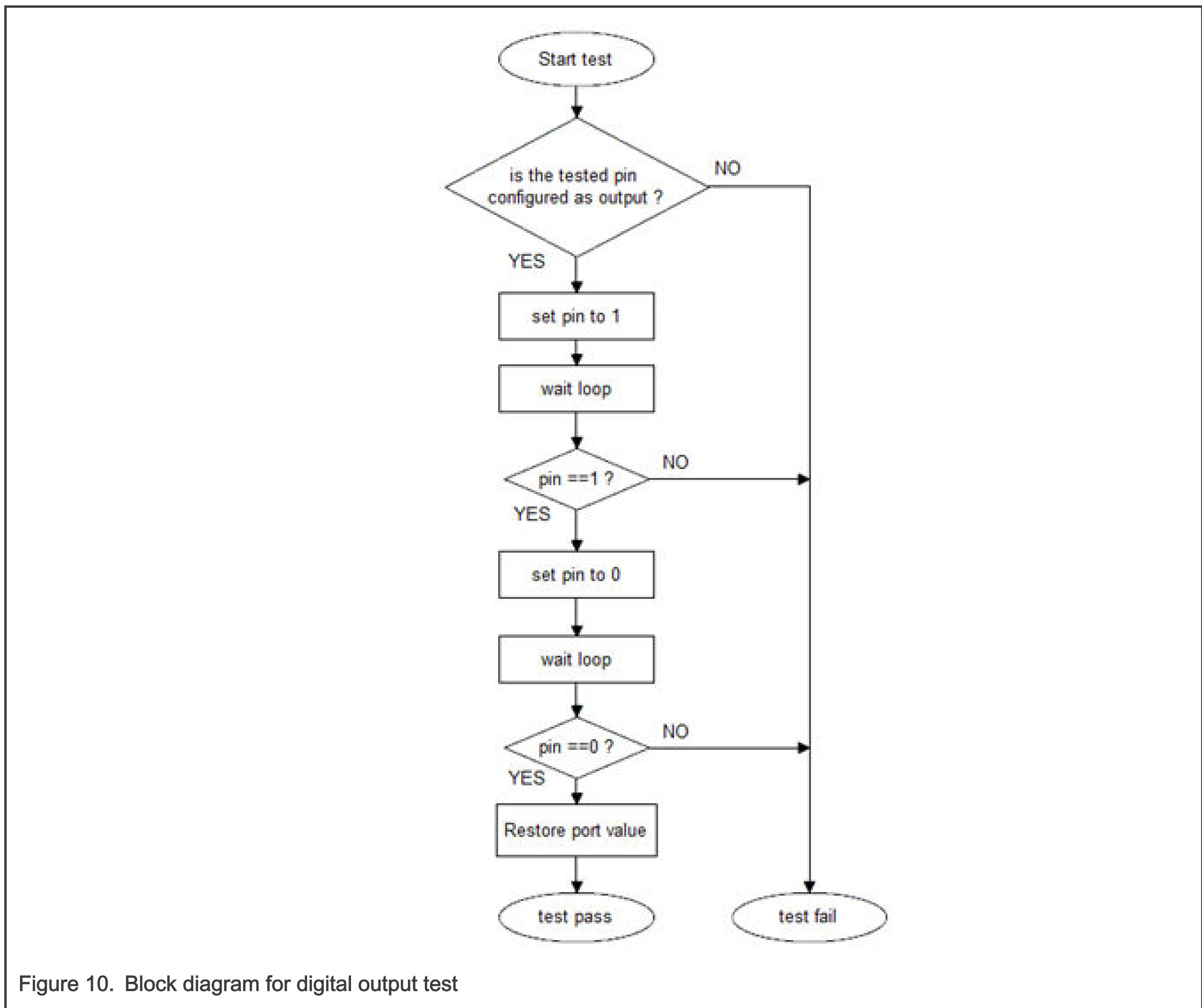

Figure 10. Block diagram for digital output test

Function prototype:

*FS_RESULT FS_DIO_Output_LPC(fs_dio_test_lpc_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.

4.2.9 FS_DIO_Output_IMXRT()

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.

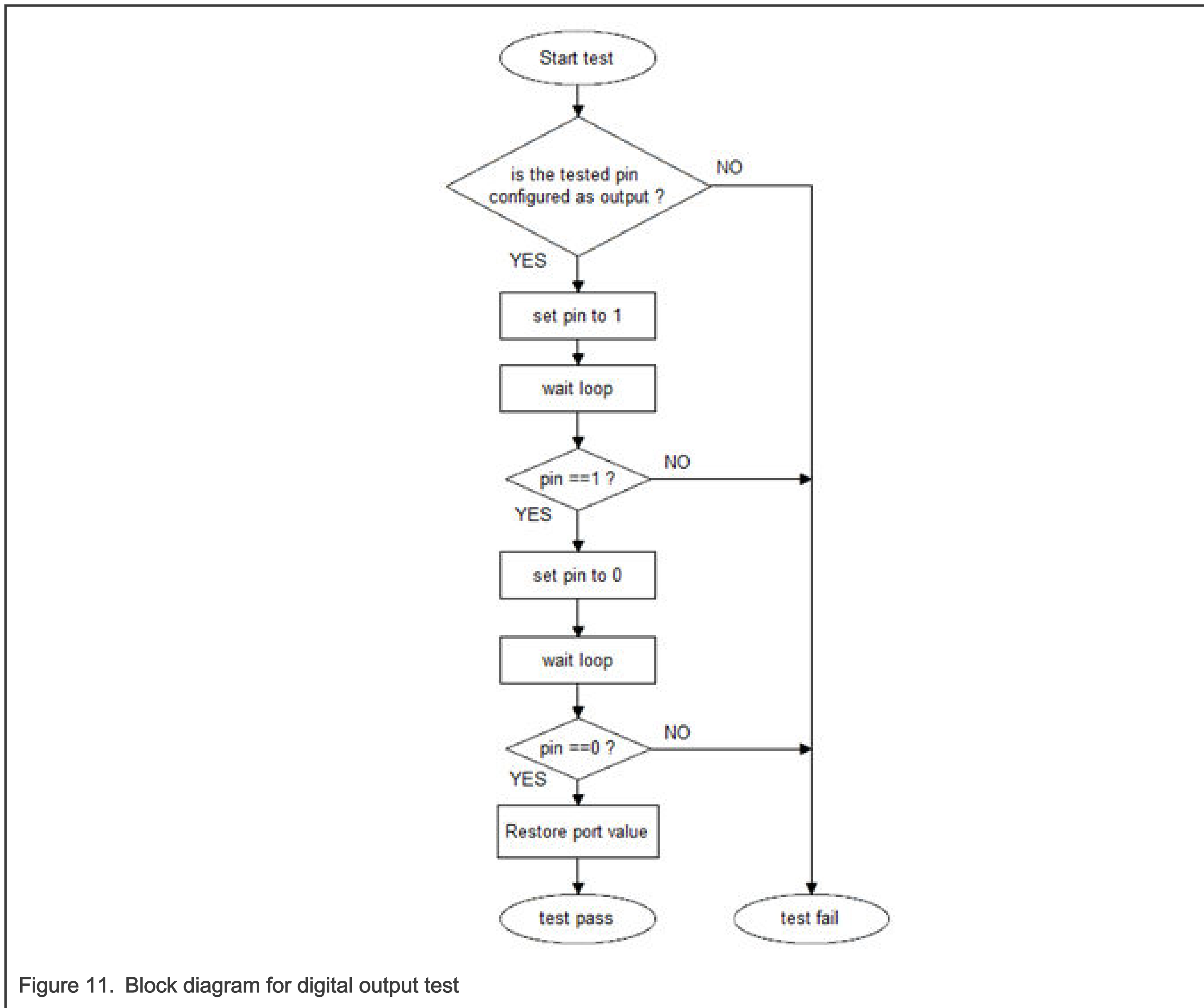

Figure 11. Block diagram for digital output test

Function prototype:

*FS_RESULT FS_DIO_Output_IMXRT(fs_dio_test_imx_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_IMXRT(&dio_safety_test_items[1], DIO_WAIT_CYCLE);

Function performance:

For information about the function performance, see 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.

4.2.10 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 Figure 12. 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.

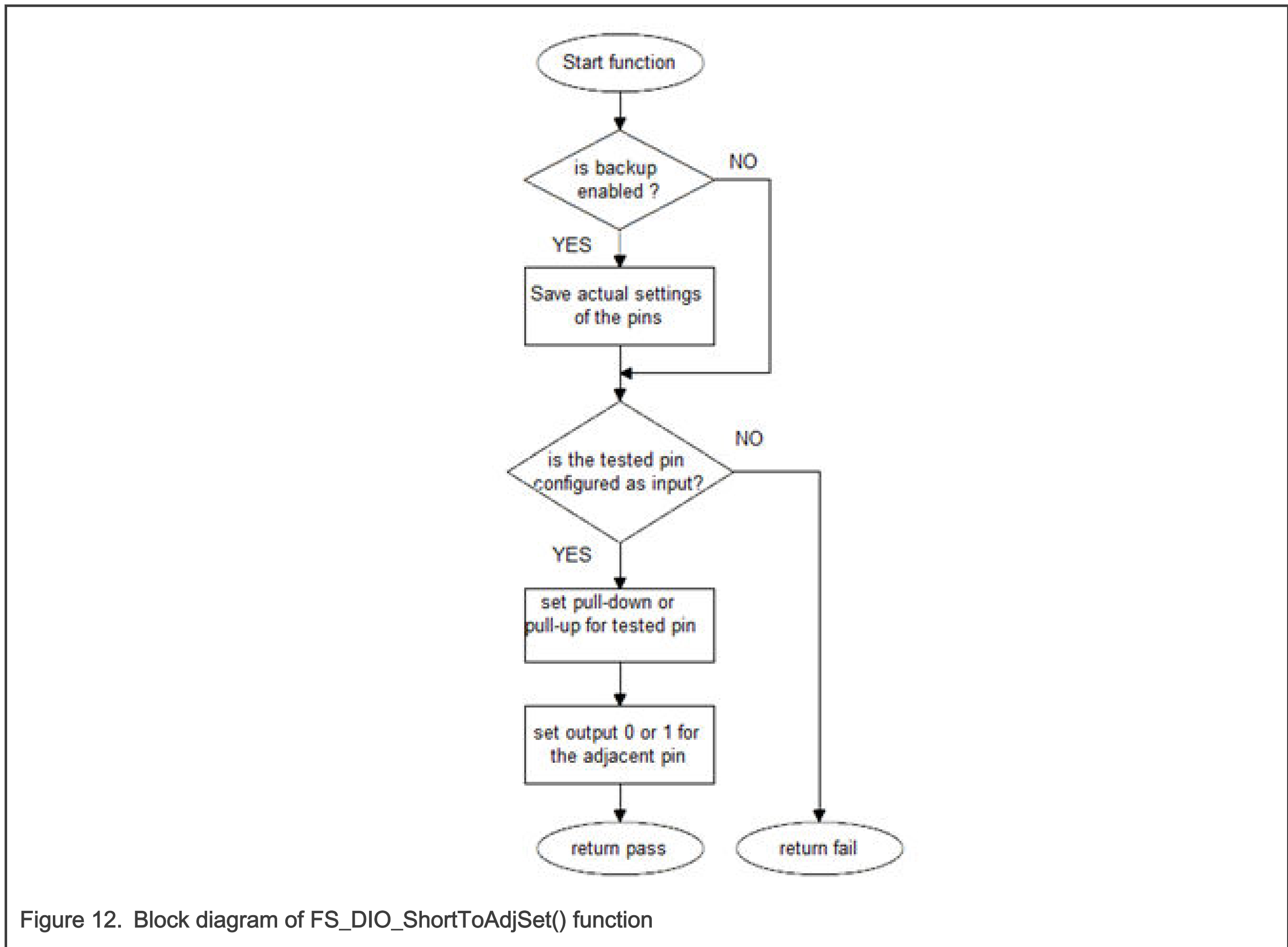

Figure 12. Block diagram of FS_DIO_ShortToAdjSet() function

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:

For information about the function performance, see 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 and the adjacent pins must be configured as GPIO outputs 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.

4.2.11 FS_DIO_ShortToAdjSet_IMX8M()

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 Figure 13. 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_IMX8M()* function is described in the respective chapter. Specify the tested pin and the adjacent pin for the input test function.

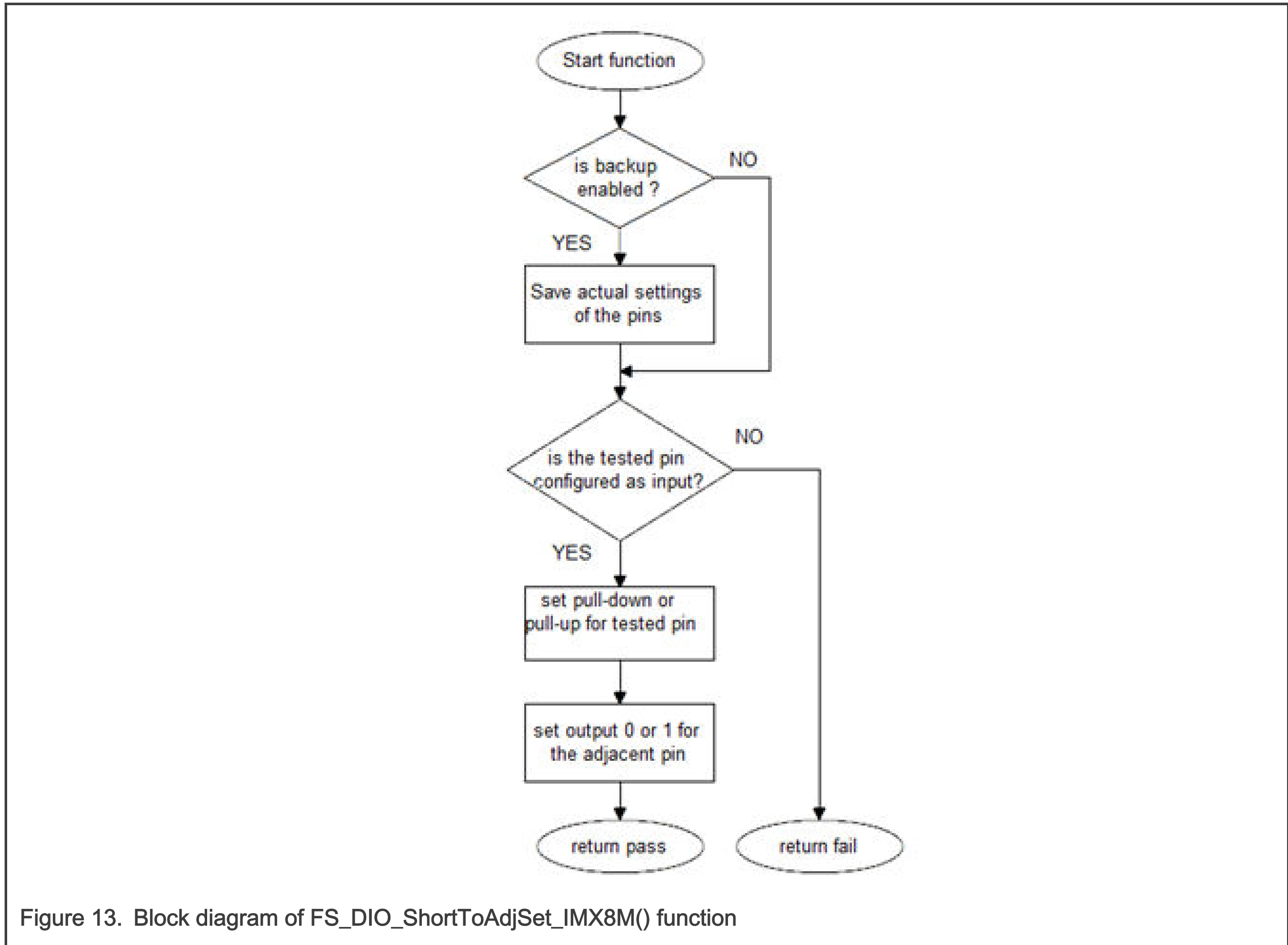

Figure 13. Block diagram of FS_DIO_ShortToAdjSet_IMX8M() function

Function prototype:

*FS_RESULT FS_DIO_ShortToAdjSet_IMX8M(fs_dio_test_imx_t *pTestedPin, fs_dio_test_imx_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_IMX8M(&dio_safety_test_items[0],
&dio_safety_test_items[1], LOGICAL_ONE, BACKUP_ENABLE);

dio_short_to_adj_test_result =FS_DIO_InputExt_IMX8M(&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 configured as a 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 (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 pin nor the adjacent pin until the *FS_DIO_InputExt_IMX8M()* function for these pins is called.

4.2.11.1 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 Figure 14. 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.

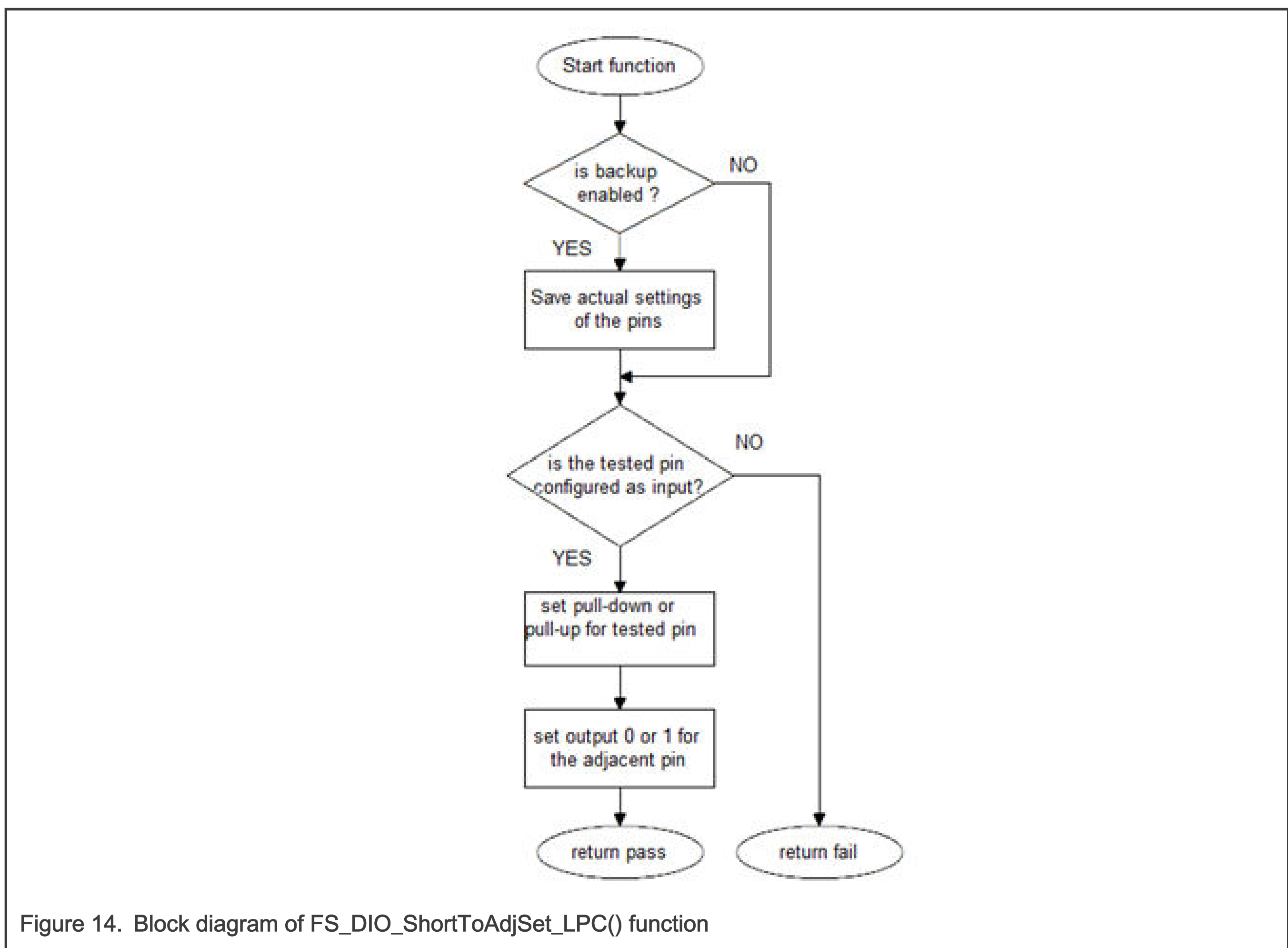

Figure 14. Block diagram of FS_DIO_ShortToAdjSet_LPC() function

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:

For information about the function performance, see Core self-test library – source code version.

Calling restrictions:

The tested must be configured as a GPIO input and adjacent pins must be configured as GPIO outputs 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.

4.2.12 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 Figure 15. 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.

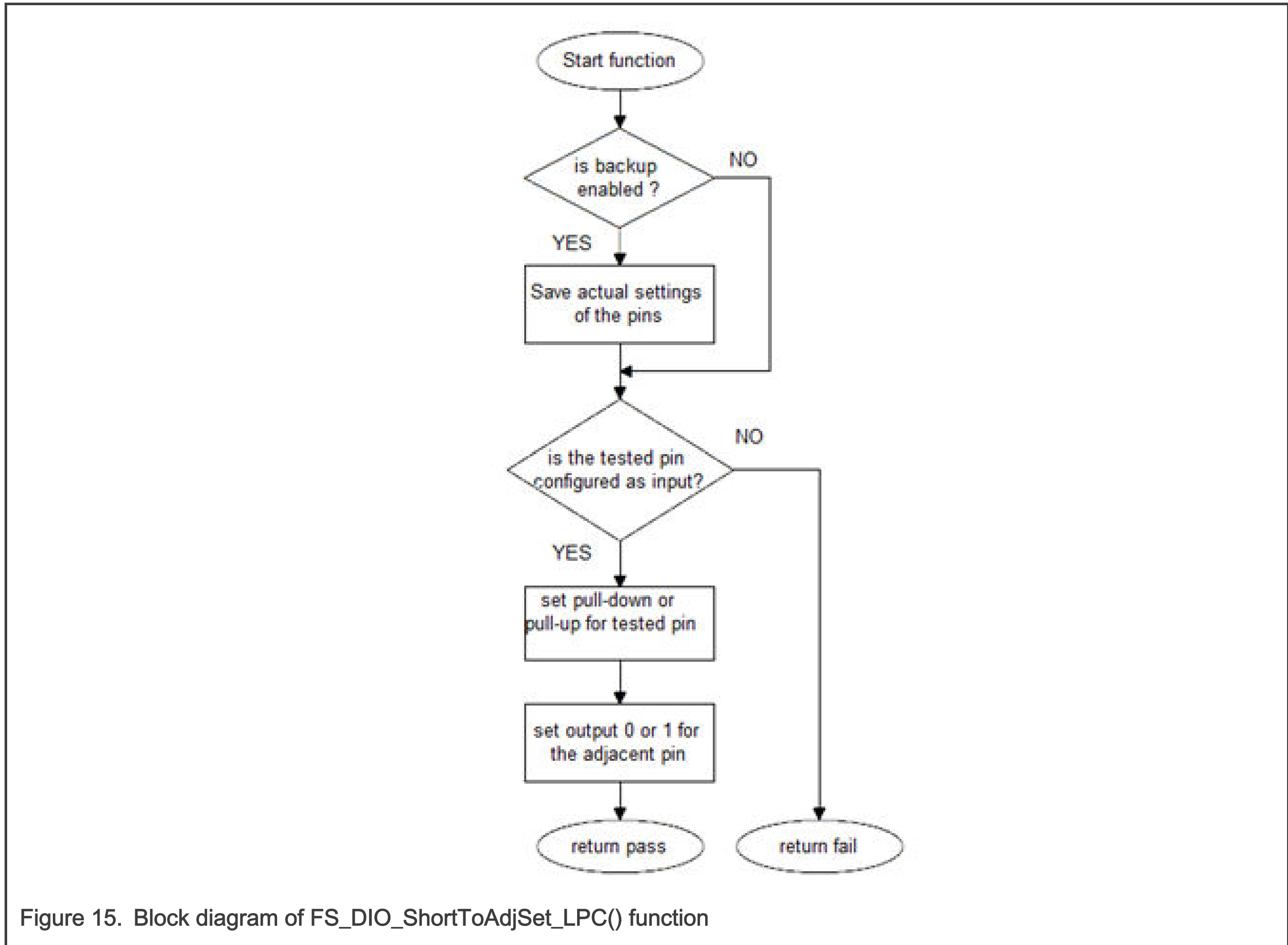

Figure 15. Block diagram of FS_DIO_ShortToAdjSet_LPC() function

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:

For information about the function performance, see Core self-test library – source code version.

Calling restrictions:

The tested must be configured as a GPIO input and adjacent pins must be configured as GPIO outputs 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.

4.2.13 FS_DIO_ShortToAdjSet_IMXRT()

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 Figure 16. 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_IMXRT()* function is described in the respective chapter. Specify the tested pin and the adjacent pin for the input test function.

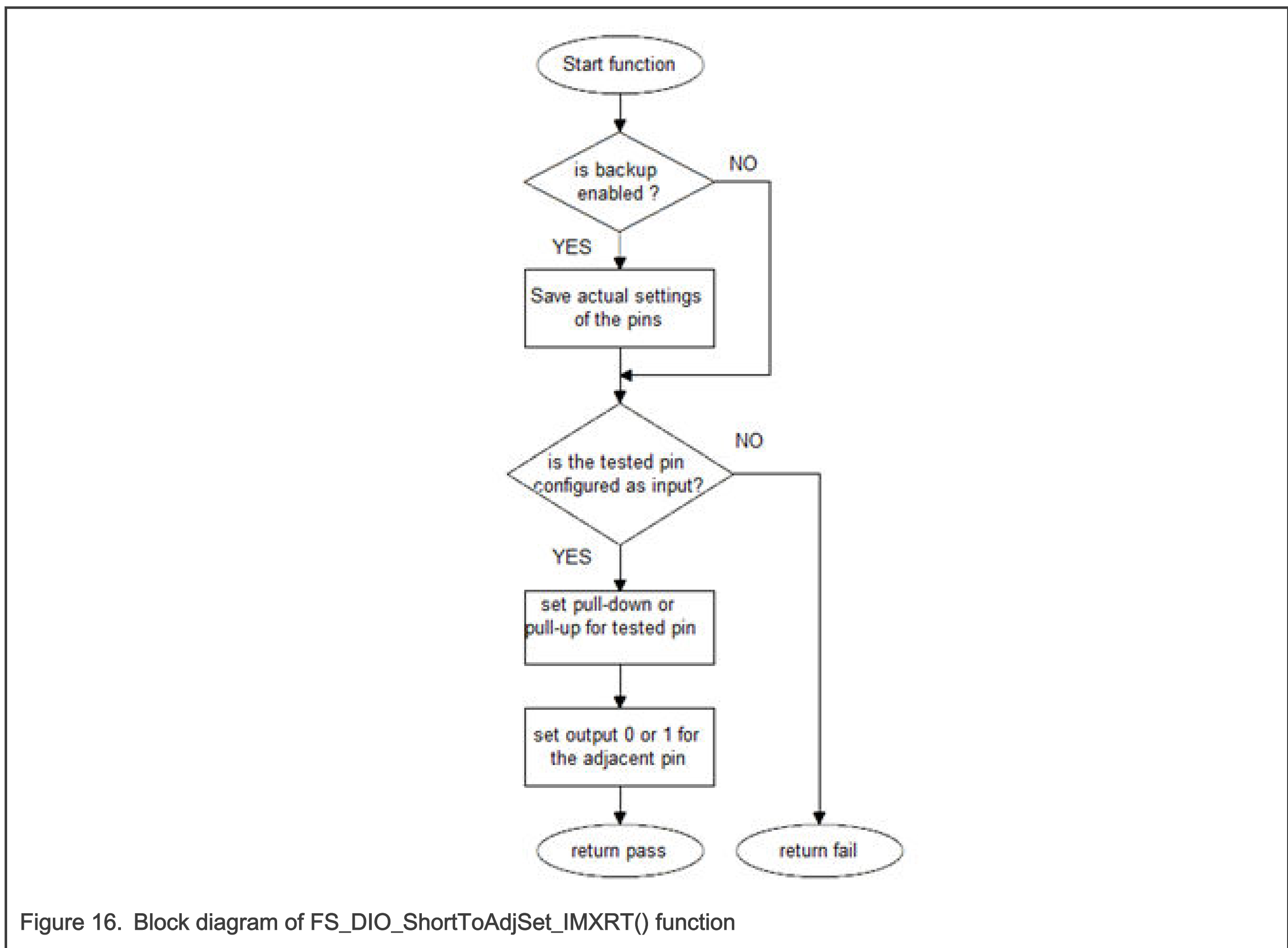

Figure 16. Block diagram of FS_DIO_ShortToAdjSet_IMXRT() function

Function prototype:

*FS_RESULT FS_DIO_ShortToAdjSet_IMXRT(fs_dio_test_imx_t *pTestedPin, fs_dio_test_imx_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_IMXRT(&dio_safety_test_items[0],
&dio_safety_test_items[1], LOGICAL_ONE, BACKUP_ENABLE);

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

Function performance:

For information about the function performance, see Core self-test library – source code version.

Calling restrictions:

The tested pin must be configured as a 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_IMXRT()* function is called for these pins.

4.2.14 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 Figure 17. 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).

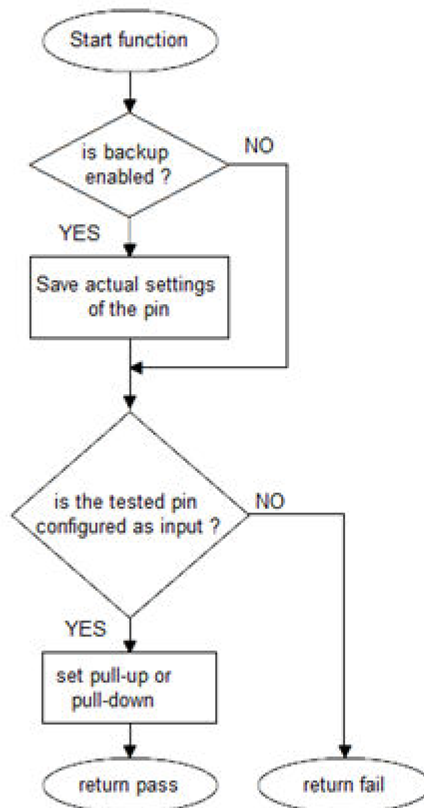

Figure 17. Block diagram of FS_DIO_ShortToSupplySet function

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 a 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:

For information about the function performance, see 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 is called for the tested pin.

4.2.15 FS_DIO_ShortToSupplySet_IMX8M()

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 Figure 18. The second part of the test (result evaluation) is ensured by the *FS_DIO_InputExt_IMX8M()* function described in the respective section. The main purpose of the *FS_DIO_InputExt_IMX8M()* 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).

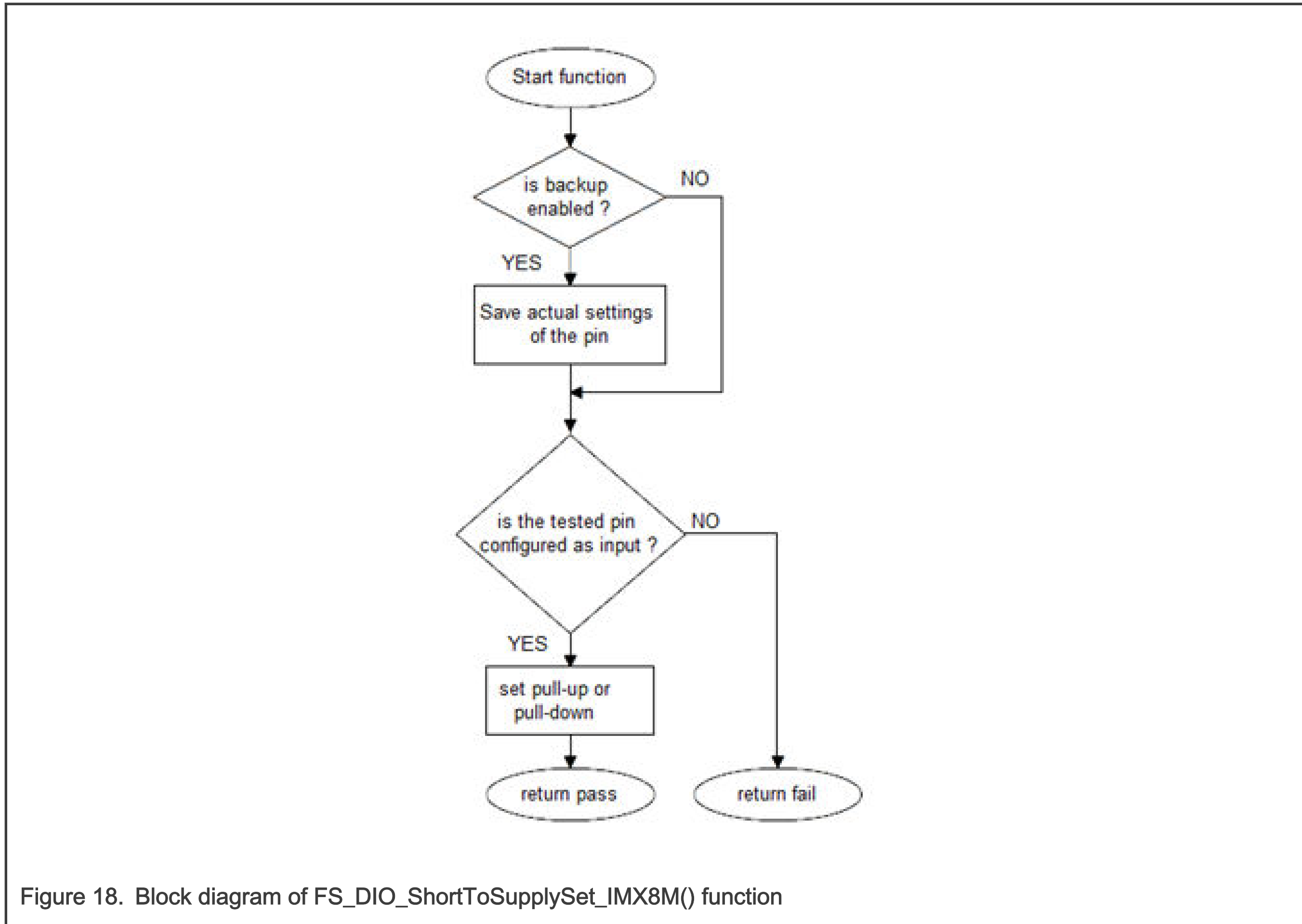

Figure 18. Block diagram of FS_DIO_ShortToSupplySet_IMX8M() function

Function prototype:

*FS_RESULT FS_DIO_ShortToSupplySet_IMX8M(fs_dio_test_imx_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 (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_IMX8M(&dio_safety_test_items[0], DIO_SHORT_TO_GND_TEST,
```

```
BACKUP_ENABLE);

dio_short_to_vcc_test_result = FS_DIO_InputExt_IMX8M(&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_IMX8M(&dio_safety_test_items[0], DIO_SHORT_TO_VDD_TEST,
BACKUP_ENABLE);

dio_short_to_vcc_test_result = FS_DIO_InputExt_IMX8M(&dio_safety_test_items[0],
&dio_safety_test_items[0], DIO_SHORT_TO_VDD_TEST, BACKUP_ENABLE);
```

Function performance:

For information about the function performance, see 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_IMX8M()* function is called for the tested pin.

4.2.16 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 Figure 19. 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).

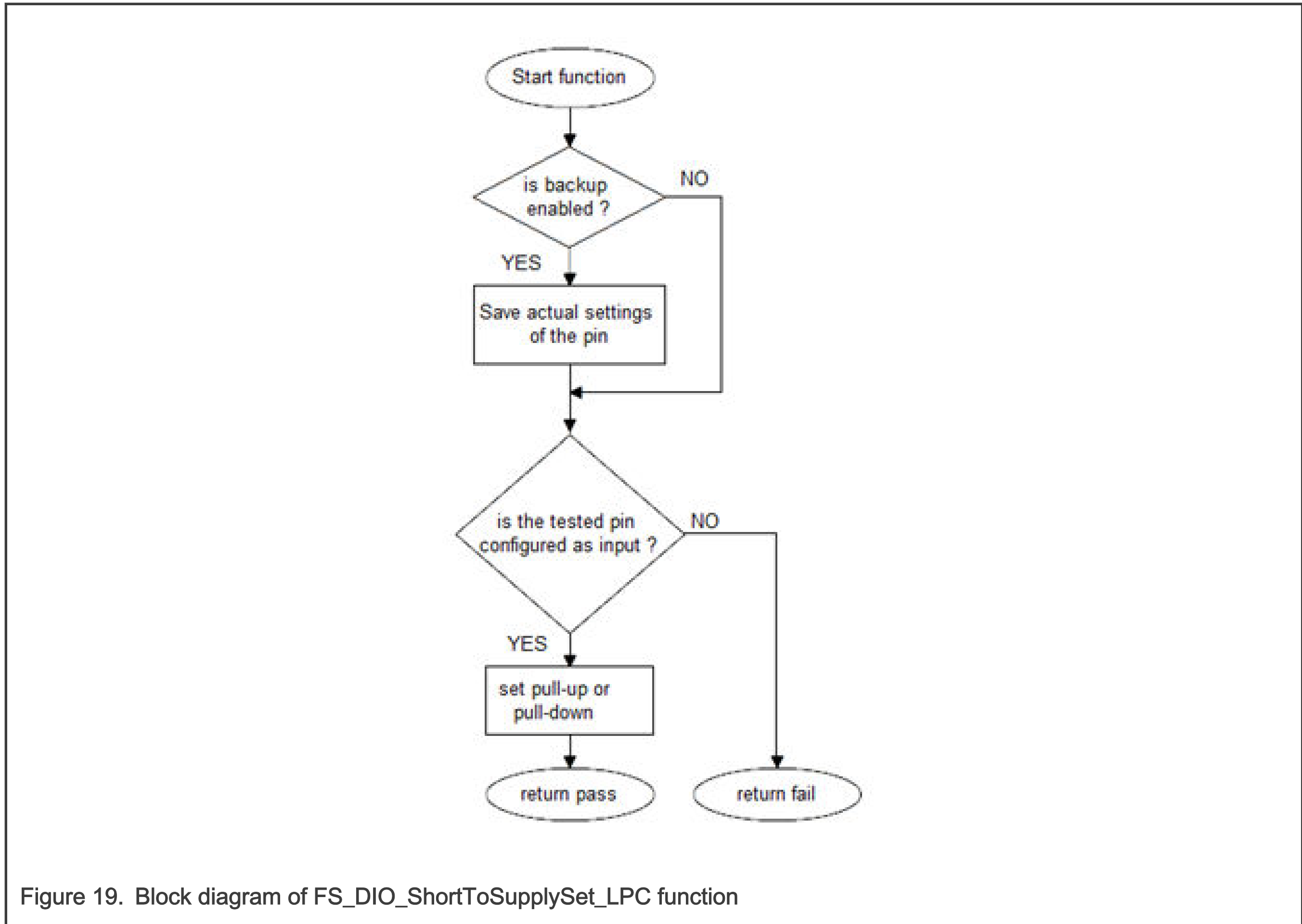

Figure 19. Block diagram of FS_DIO_ShortToSupplySet_LPC function

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 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 GND is tested, the parameter must have a 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:

For information about the function performance, see 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 is called for the tested pin.

4.2.17 FS_DIO_ShortToSupplySet_IMXRT()

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 Figure 20. The second part of the test (result evaluation) is ensured by the *FS_DIO_InputExt_IMXRT()* function described in the respective section. The main purpose of the *FS_DIO_InputExt_IMXRT()* 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).

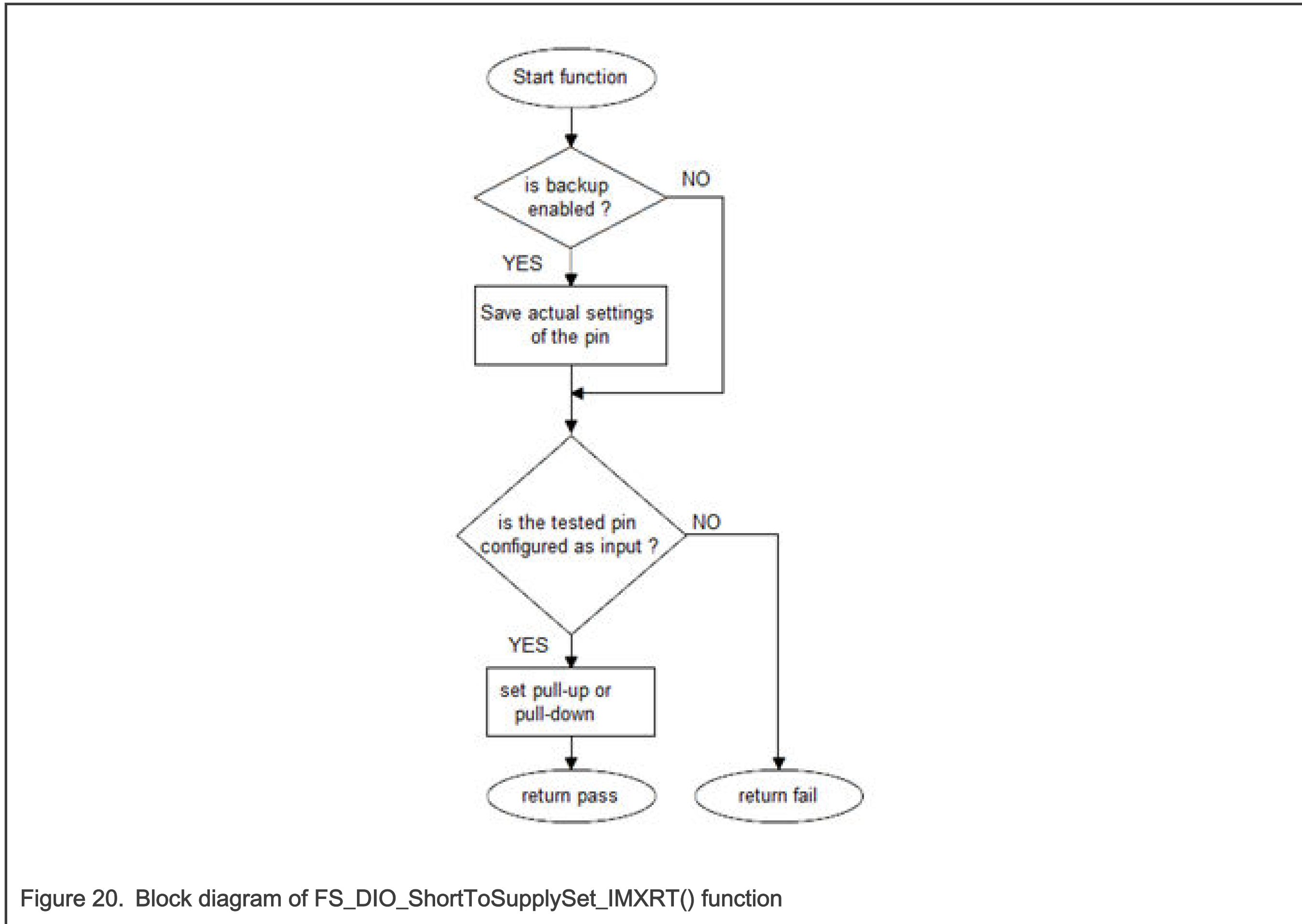

Figure 20. Block diagram of FS_DIO_ShortToSupplySet_IMXRT() function

Function prototype:

*FS_RESULT FS_DIO_ShortToSupplySet_IMXRT(fs_dio_test_imx_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 (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_IMXRT(&dio_safety_test_items[0], DIO_SHORT_TO_GND_TEST,
```

```
BACKUP_ENABLE);

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

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

Function performance:

For information about the function performance, see 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_IMXRT()* function is called for the tested pin.

Chapter 5
Invariable memory test

The invariable memory on the supported 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 execution. 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 the same as that used in runtime (16-bit CRC polynomial (0x1021) for SW16 and HW16 or 0x04C11DB7 for HW32 and SW32) to generate a CRC code for error detection. The same algorithm is implemented in the hardware CRC module. In the IAR IDE, you can calculate the CRC using 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 Table 7.

Table 7. 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_cm33_flash.S* and written as assembler functions. The header file with the definitions and function prototypes is *iec60730b_cm33_flash.h*. The *iec60730b.h* and *asm_mac_common.h* and *iec60730b_types.h* are the common header files for the safety library.

The following functions are implemented in *iec60730b_cm33_flash.S*:

- *FS_CM33_FLASH_HW16()*
- *FS_CM33_FLASH_HW32()*
- *FS_CM33_FLASH_SW16()*
- *FS_CM33_FLASH_SW32()*

The functions have the same input and output parameters and the same functionality. The hardware function uses the hardware CRC module that is included in the supported MCU. The software 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 tools in the Keil-uVision IDE, see *Calculating Post-Build CRC in Arm® Keil®* (document AN12520).

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*.

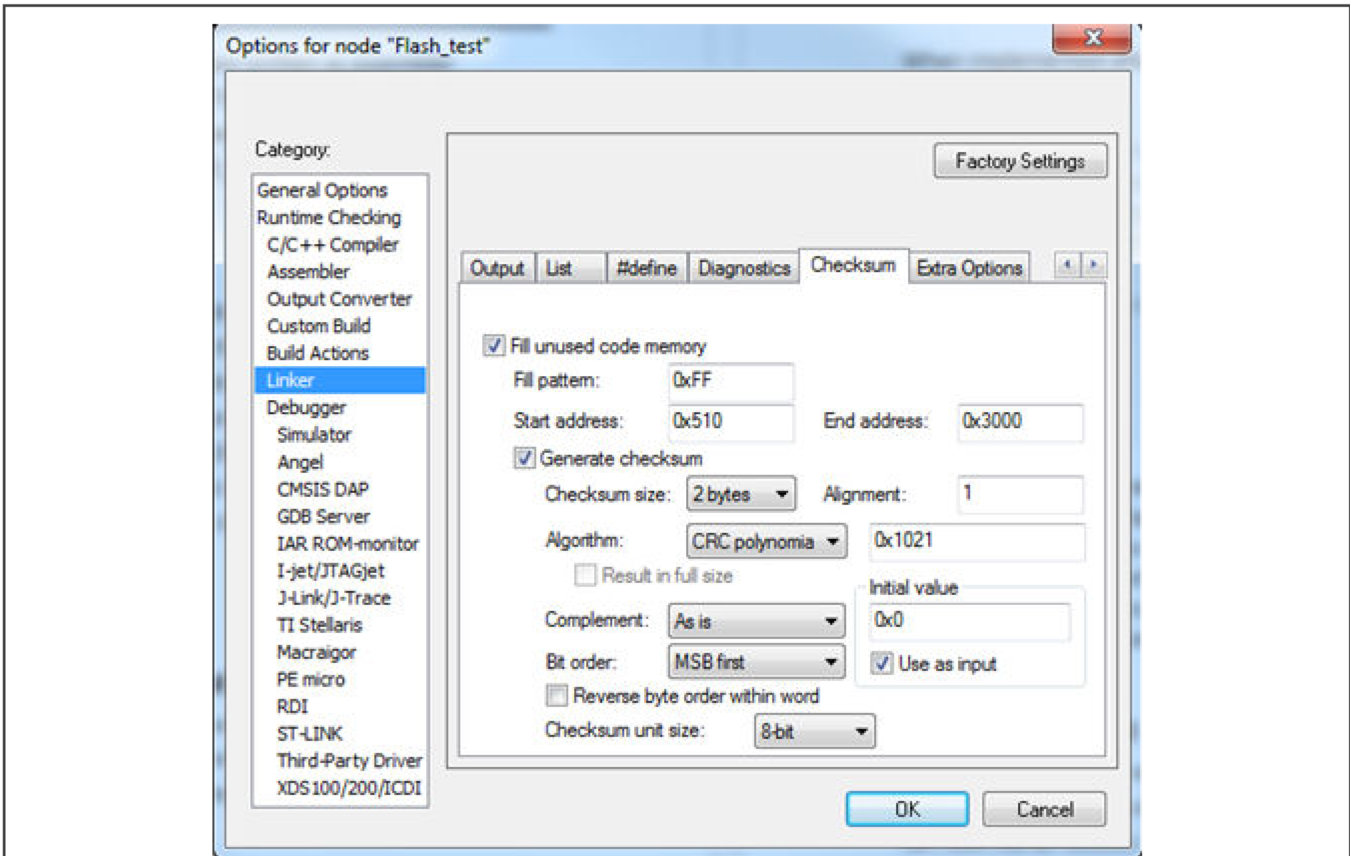

Figure 21. Checksum settings for linker

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 the following 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"
```

```
 extern uint16_t const __checksum;
if((uint16_t)__checksum != FS_CM33_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 and 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_CM33_FLASH_HW16(flash_crc.actual_address,
flash_crc.block_size, CRC_BASE, flash_crc.part_crc);
if (FS_FAIL_FLASH == 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_CM33_FLASH_HW16()

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

Function prototype:

uint16_t FS_CM33_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 of 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.[1]

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

Table 8. Duration of FS_CM33_FLASH_HW16() in dependence of tested block size

Block size (Bytes)	Clock cycles	Execution time (approximately)
0x10	205	2.14 µs
0x20	341	3.55 µs
0x50	749	7.80 µs

Calling restrictions:

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

5.2.5 FS_CM33_FLASH_SW16()

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

Function prototype:

uint16_t FS_CM33_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 - It has no effect. It is here only 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 54 B.[1]

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

Table 9. Duration of IEC60730B_Flash_SWTest() in dependence of tested block size

Block size (Bytes)	Clock cycles	Execution time (approximately)
0x4	1907	19.87 µs
0x8	3687	38.41 µs
0x10	9091	94.70 µs

Calling restrictions:

None.

5.2.6 FS_CM33_FLASH_HW32()

This function generates the 32-bit CRC value using the hardware CRC module.

Function prototype:

uint32_t FS_CM33_FLASH_HW32(uint32_t startAddress, uint32_t size, uint32_t moduleAddress, uint32_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:

uint32_t - The 32-bit CRC value of the memory range (CRC-32/MPEG-2 - 0x04C11DB7).

Function performance:

The function size is 40 B.[1]

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

Table 10. Duration of FS_CM33_FLASH_HW32() in dependence of tested block size

Block size (Bytes)	Clock cycles	Execution time (approximately)
0x10	192	2.00 µs
0x20	336	3.50 µs
0x50	744	7.75 µs

Calling restrictions:

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

5.2.7 FS_CM33_FLASH_SW32()

This function calculates the 32-bit CRC polynomial (0x04C11DB7) without using hardware.

Function prototype:

uint32_t FS_CM33_FLASH_SW32(uint32_t startAddress, uint32_t size, uint32_t moduleAddress, uint32_t crcVal);

Function inputs:

startAddress - The first address of the tested memory.

size - The size of the tested memory.

moduleAddress - It has no effect. It is here only 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:

uint32_t - The 32-bit CRC value of the memory range (CRC-32/MPEG-2 - 0x04C11DB7).

Function performance:

The function size is 65 B.[1]

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

Table 11. Duration of FS_CM33_FLASH_SW32() in dependence of tested block size

Block size (Bytes)	Clock cycles	Execution time (approximately)
0x4	1725	17.97 µs
0x8	3405	35.47 µs
0x10	8369	87.18 µs

Calling restrictions:

None.

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:

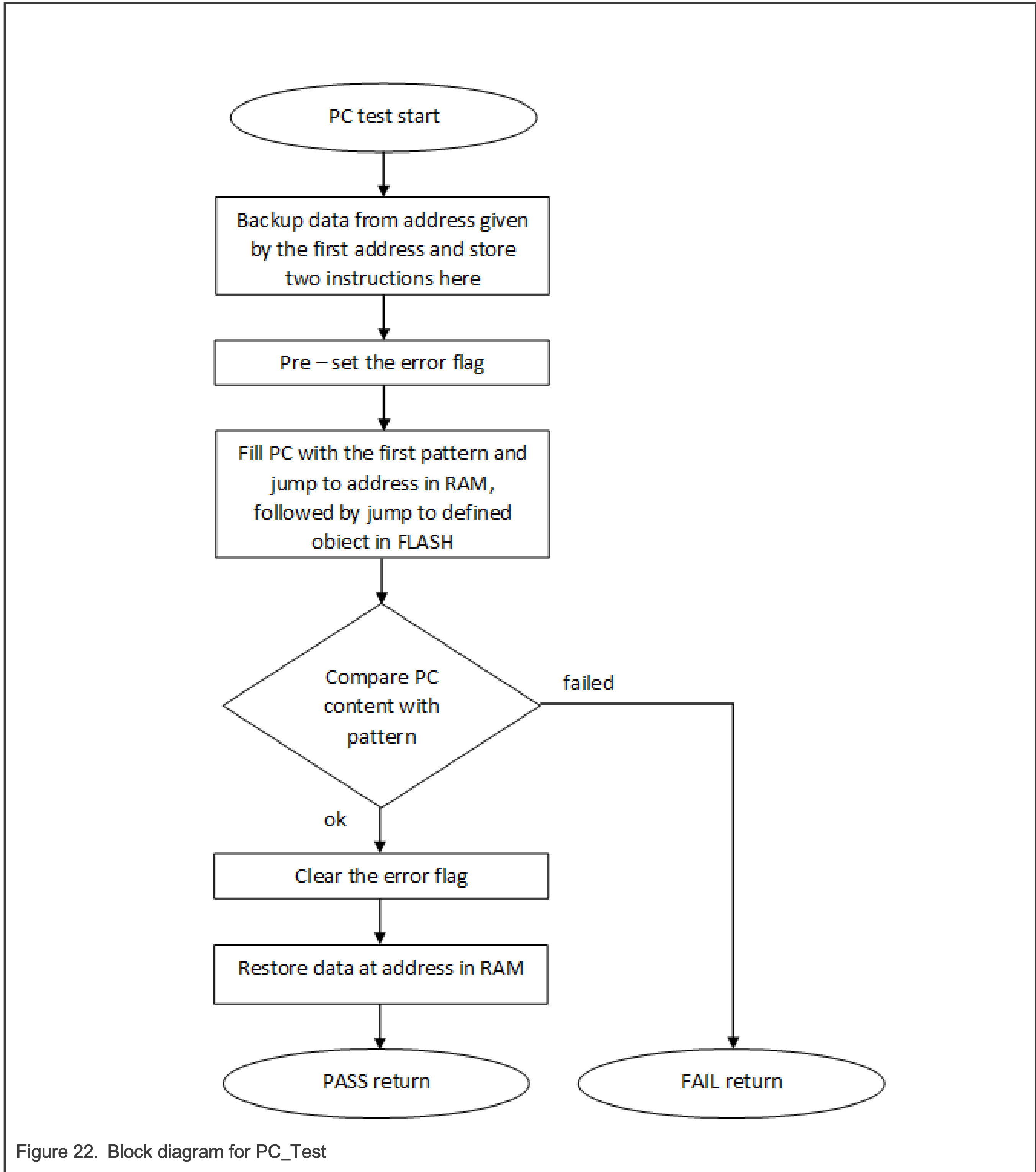

Figure 22. Block diagram for PC_Test

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 12. 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_cm33_pc.S* file and written as assembler functions. The header file with the test patterns and the function prototypes is *iec60730b_cm33_pc.h*. The *iec60730b.h* and *asm_mac_common.h* and *iec60730b_types.h* are the common header files for the safety library. For the second test type, the *iec60730b_cm33_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 as follows:

FS_CM33_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_cm33_pc_object.S* file to an appropriate address in the flash memory.

The following 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_CM33_PC_Test(0x20000013, FS_PC_object, PC_TEST_FLAG);
if (FS_FAIL_PC == fs_pc_test_result)
 SafetyError();
```

6.2.1 FS_CM33_PC_Test()

The program counter register is tested according to the block diagram in Figure 22.

Function prototype:

*FS_RESULT FS_CM33_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 the memory used as a flag. If the flag is "0", the test is successful ("1" if the test failed).

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_PC* - In case of incorrect test execution, PC_flag has a value of "1".

Function performance:

The function takes approximately 99 cycles (1.03 μs).[1]

The function size is 48 B.[1]

Calling restrictions:

This function cannot be interrupted.

6.2.2 FS_PC_Object()

This function is internally used by the *FS_CM33_PC_Test()* function. Function is used for performing PC test, it should be called only by *FS_CM33_PC_Test()* function. It should be placed in reliable address - by editing the linker file.

The following example shows how to place the function at the desired address in the linker configuration file for the IAR tool:

```
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_cm33_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_CM33_PC_Test()* function. Its size is 20 bytes.[1]

Calling restrictions:

This function is used to perform the PC test, it should be called only by the *FS_CM33_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:

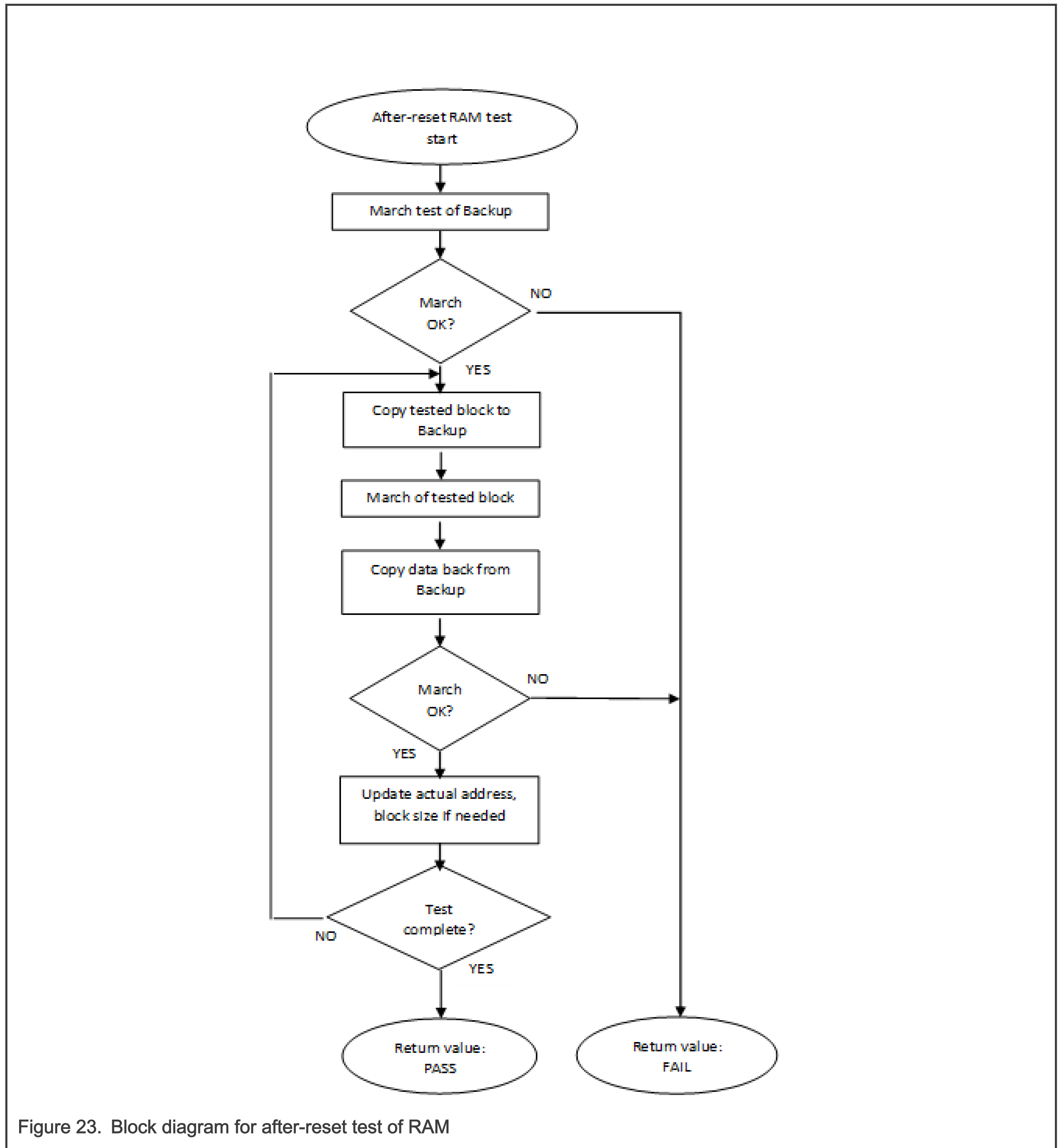

Figure 23. Block diagram for after-reset test of RAM

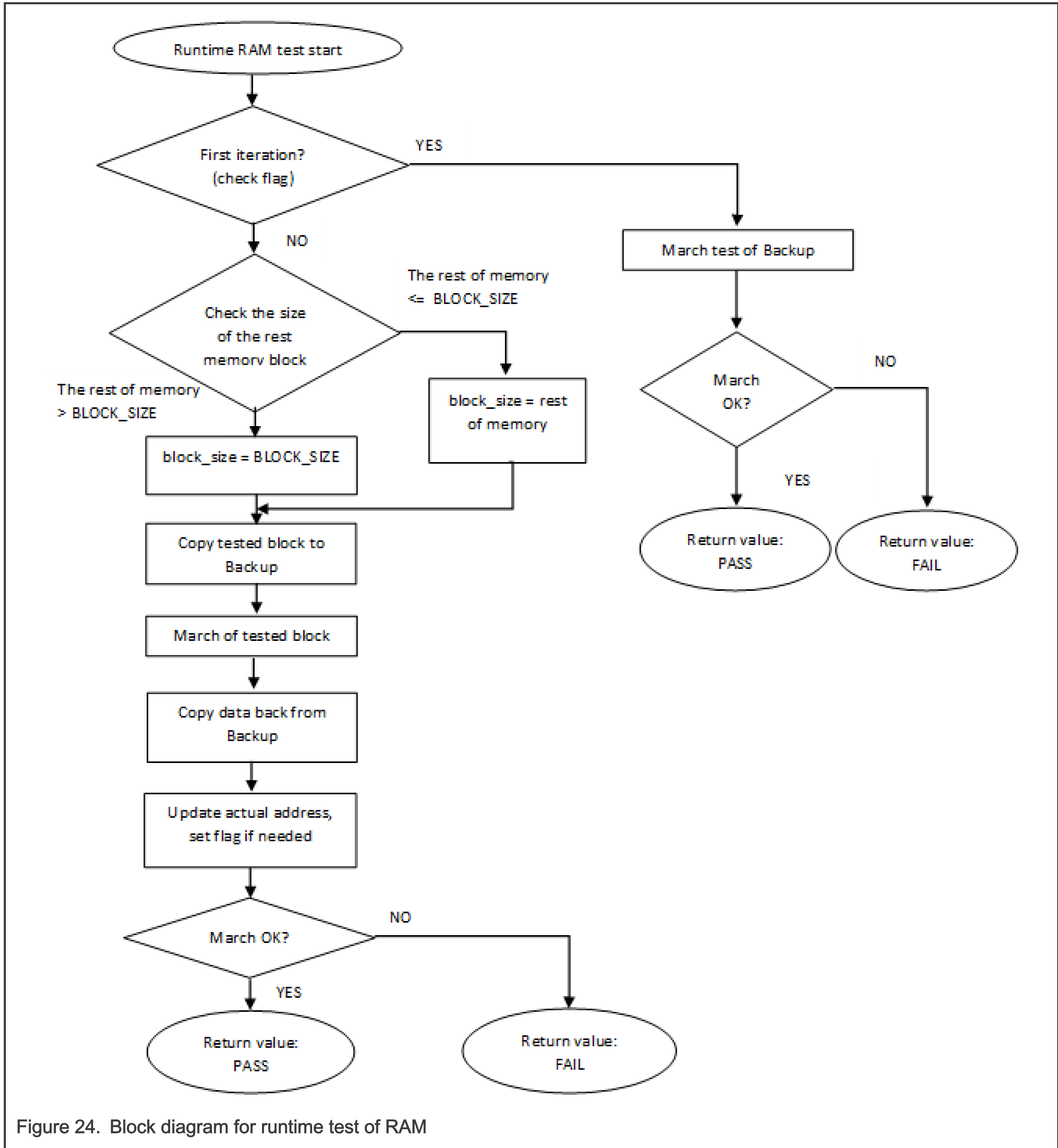

Figure 24. Block diagram for runtime test of RAM

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 13. 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_cm33_ram.S* file and written as assembler functions. The header file with return values and function prototypes is *iec60730b_cm33_ram.h.* The *iec60730b.h* and *asm_mac_common.h* and *iec60730b_types.h* are the common header files for the safety library.

The RAM test consists of these public functions:

- *FS_CM33_RAM_RuntimeTest()*
- *FS_CM33_RAM_AfterResetTest()*
- *FS_CM33_RAM_SegmentMarchC()*
- *FS_CM33_RAM_SegmentMarchX()*
- *FS_CM33_RAM_CopyToBackup()*
- *FS_CM33_RAM_CopyFromBackup()*

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

7.2.1 FS_CM33_RAM_AfterReset()

The after-reset test is done by the *FS_CM33_RAM_AfterReset()* function. This function is called once after the 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 by 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 23.

The following is an example of a function call:

```
#include "iec60730b.h"
if (FS_FAIL_RAM == FS_CM33_RAM_AfterReset(start_address, end_address, block_size,
backup_address, FS_CM33_RAM_SegmentMarchC))
SafetyError();
```

<u>Function prototype:</u>

FS_RESULT FS_CM33_RAM_AfterReset(uint32_t startAddress, uint32_t endAddress, uint32_t blockSize, uint32_t backupAddress, tFcn pMarchType);

<u>Function inputs:</u>

startAddress - The first adress 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).

<u>Function output:</u>

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_RAM*

<u>Function performance:</u>

The function size is 98 B.[1]

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

Table 14. FS_CM33_RAM_AfterReset duration

Memory Size (Bytes)	Block S Size (Bytes)	Cycles - March X	Cycles - March C
0x100	0x20	4212	5686
0x100	0x40	3882	5354
0x100	0x80	4042	5708
0x200	0x20	4218	10766
0x200	0x40	3882	9750
0x200	0x80	4042	9770
0x400	0x20	4218	20926
0x400	0x40	3882	18542
0x400	0x80	4042	17878

<u>Calling restrictions:</u>

This function is used once after the MCU reset, when the execution time is not critical. It cannot be interrupted. The backup area must be at least the same size as the tested block size defined by the block_size parameter.

7.2.2 FS_CM33_RAM_Runtime()

The runtime test is done by the *FS_CM33_RAM_Runtime()* function. Reserve free memory space dedicated for the backup. The block size parameter cannot be larger than the size of the backup area. During the first call, the function checks the backup area. After the call, the blocks of memory are processed in a sequence. They are copied to the backup area and their locations are checked with the respective March test. The data is copied back to the original memory area and the actual address and the block size are updated. This is repeated until the last block of memory is tested. If a DC fault is detected, the function returns a fail pattern. The block diagram is shown in the above figure. This is an example of the function call:

```
#include "iec60730b.h"
if (FS_FAIL_RAM == FS_RESULT FS_CM33_RAM_Runtime(start_address, end_address, &actual_address,
block_size, backup_address,IEC60730B_RAM_SegmentMarchX))
SafetyError();
```

<u>Function prototype:</u>

*FS_RESULT FS_CM33_RAM_Runtime(uint32_t startAddress, uint32_t endAddress, uint32_t *pActualAddress, uint32_t blockSize, uint32_t backupAddress, tFcn pMarchType);*

<u>Function inputs:</u>

startAddress - The first address of the tested RAM area.

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 118 B. [1]

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

Table 15. FS_CM33_RAM_Runtime duration

Block size (Bytes)	Cycles - March X	Cycles - March C
0x4	198	224
0x8	249	313
0x20	417	577
0x40	641	929

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_CM33_RAM_CopyFromBackup()

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

Function prototype:

void FS_CM33_RAM_CopyFromBackup(uint32_t startAddress, uint32_t blockSize, uint32_t backupAddress);

Function inputs:

startAddress - The first adress 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.[1]

7.2.4 FS_CM33_RAM_CopyToBackup()

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

Function prototype:

void FS_CM33_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.[1]

7.2.5 FS_CM33_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_CM33_RAM_SegmentMarchC(uint32_t startAddress, uint32_t blockSize);

Function inputs:

startAddress - The first adress 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.[1]

7.2.6 FS_CM33_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_CM33_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:

The function size is 114 B.[1]

Chapter 8
CPU register test

The CPU register test procedure tests all of the CM33 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-R12

Stack pointer registers:

- MSP + MSPLIM (secure/non-secure)
- PSP + PSPLIM (secure/non-secure)

Special registers:

- APSR
- CONTROL (secure/non-secure)
- PRIMASK (secure/non-secure)
- FAULTMASK (secure/non-secure)
- BASEPRI (secure/non-secure)

Link register:

- LR

FPU registers:

- FPSCR
- S0 – S31

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 corrupt 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 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.

Due to the Arm® TrustZone® support, some core registers are banked between the security states. The Secure (S) or Non-Secure (NS) sets of the corresponding registers are accessible during execution (depending on the current security state). Both register versions are accessible during the S state, but not during the NS state. This is the reason why the NXP Safety Library must be executed in a secure mode. All of the banked registers are listed above.

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. In case of the FPU registers test, the content of the FPSCR is backed up. The CPACR system register contains one bit for enabling the FPU. The block diagrams for the respective registers are shown in the following figures:

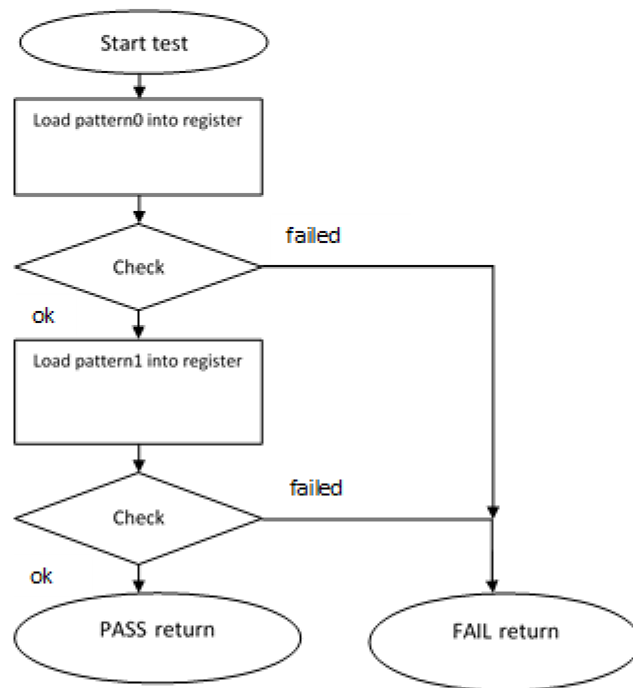

Figure 25. Block diagram for R2 – R12 registers test

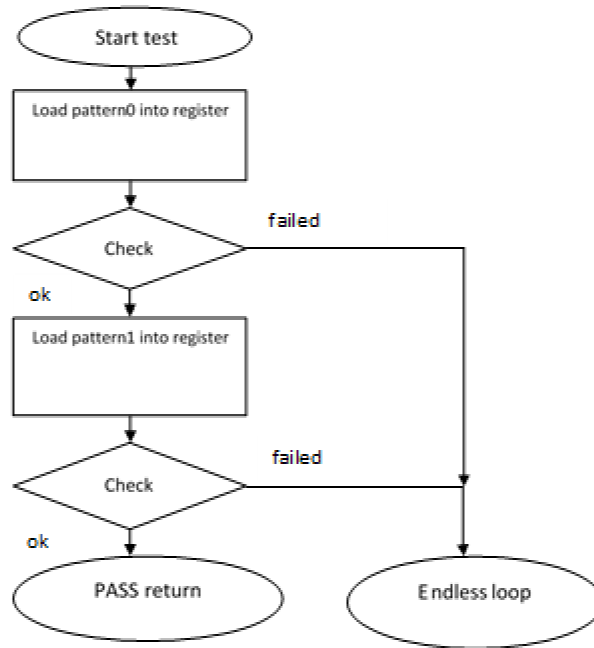

Figure 26. Block diagram for R0, R1, LR, APSR registers test

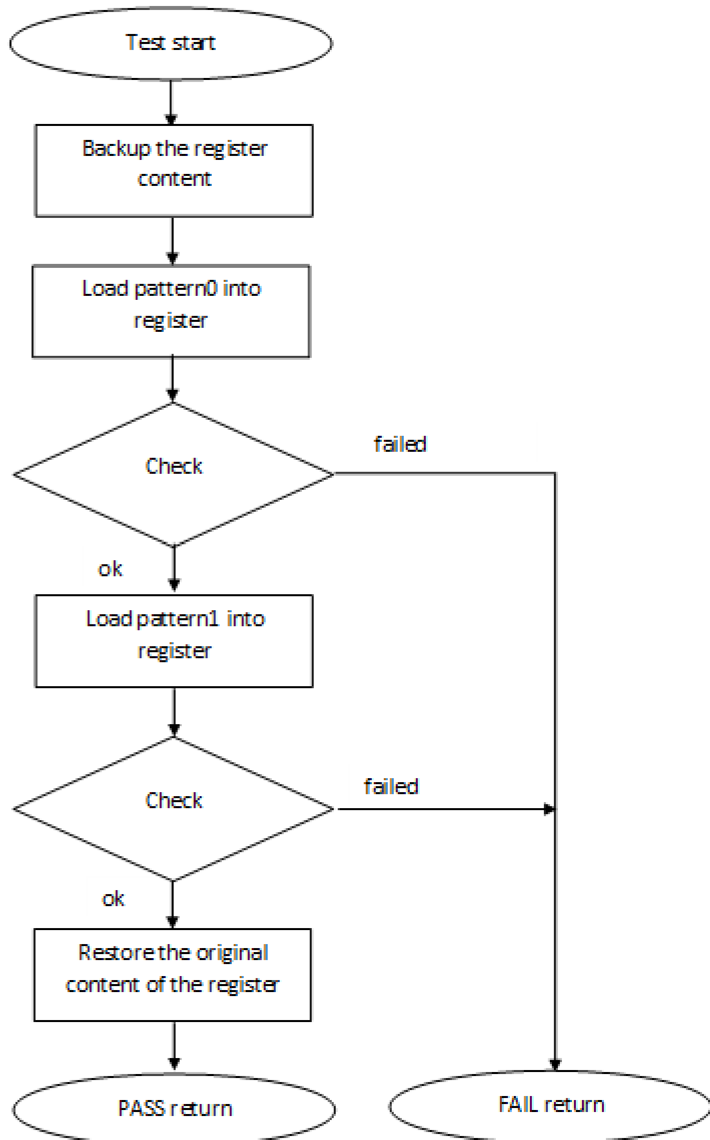

Figure 27. Block diagram for PRIMASK , FAULTMAST, BASEPRI and CONTROL registers test

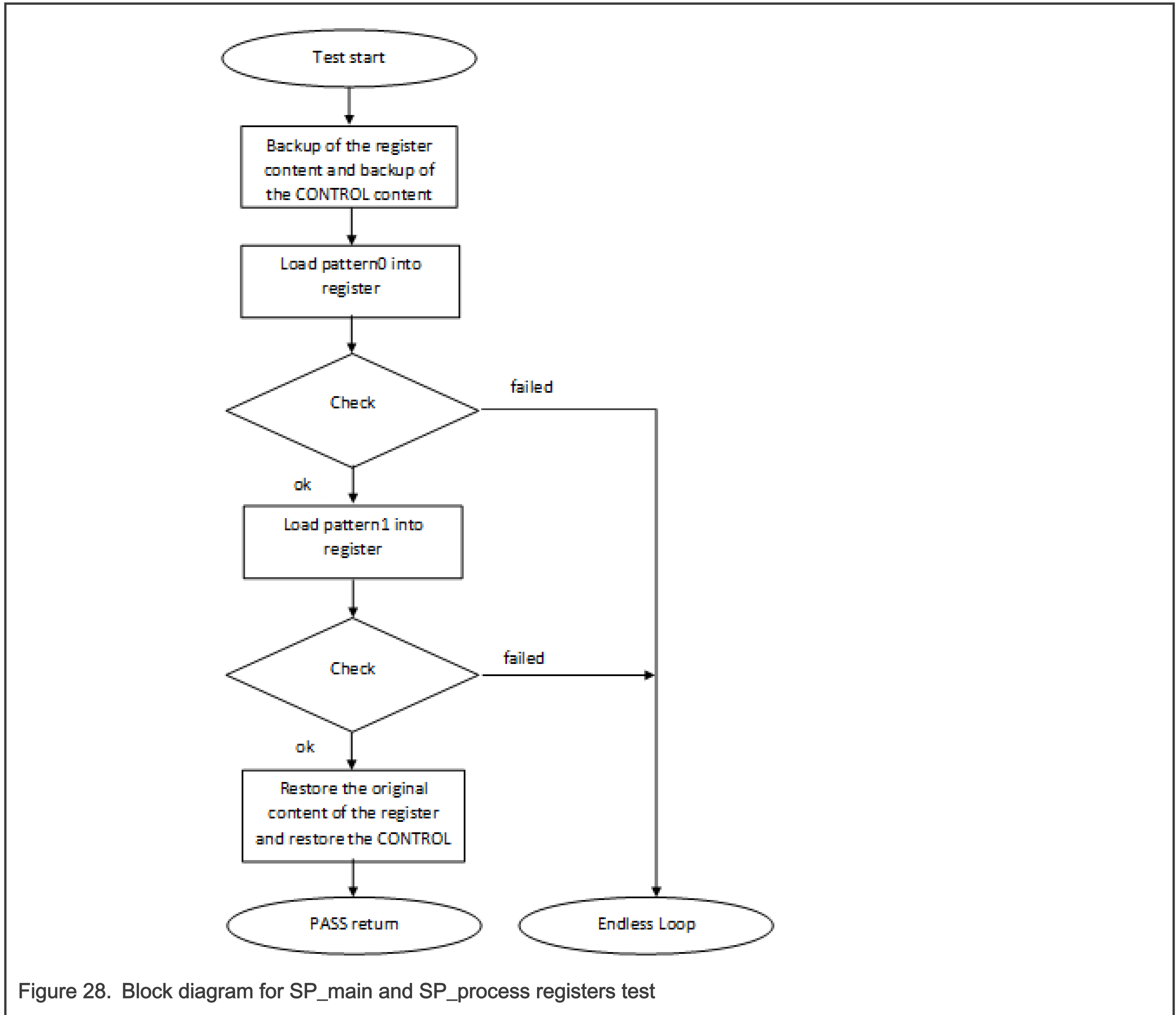

Figure 28. Block diagram for SP_main and SP_process registers test

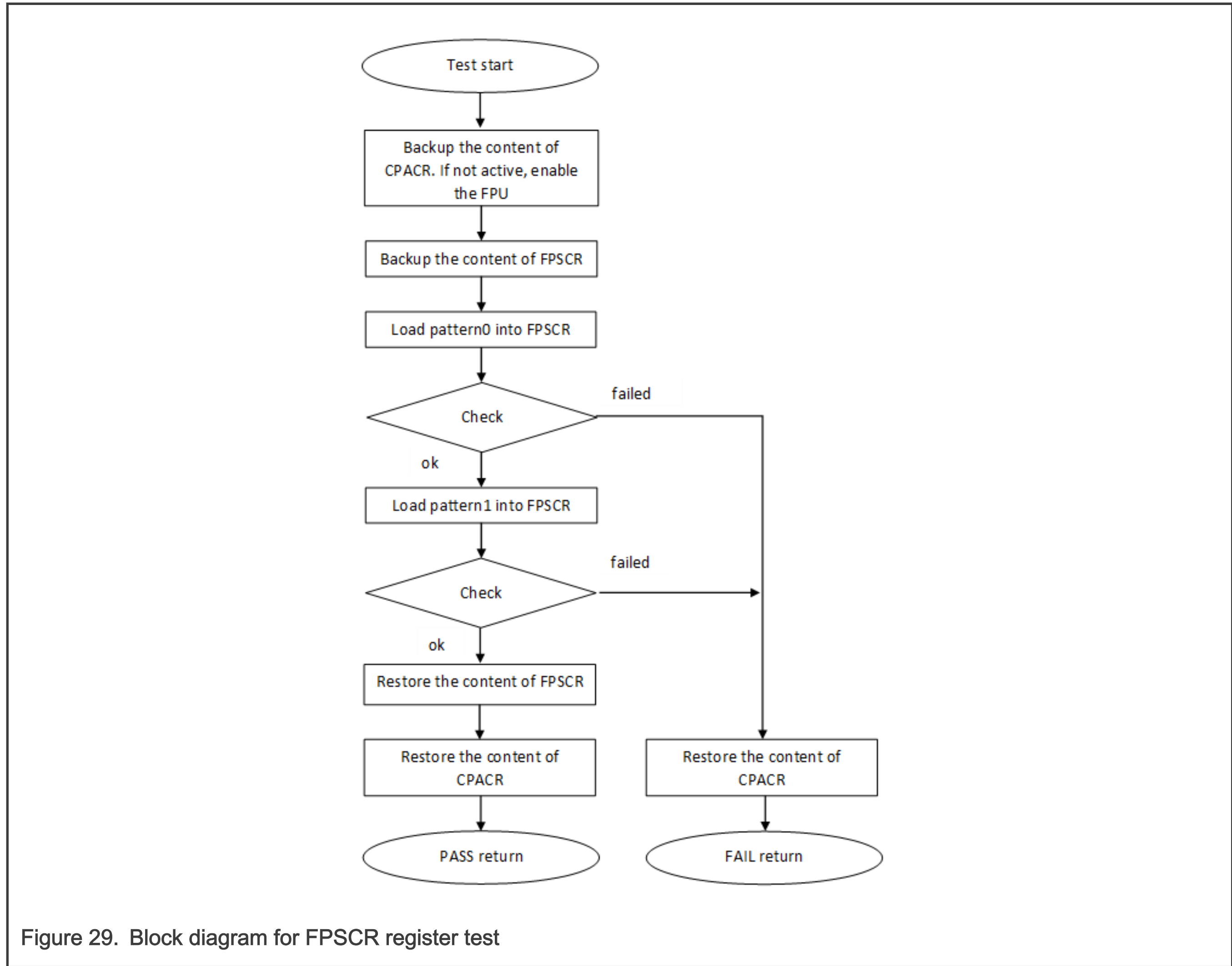

Figure 29. Block diagram for FPSCR register test

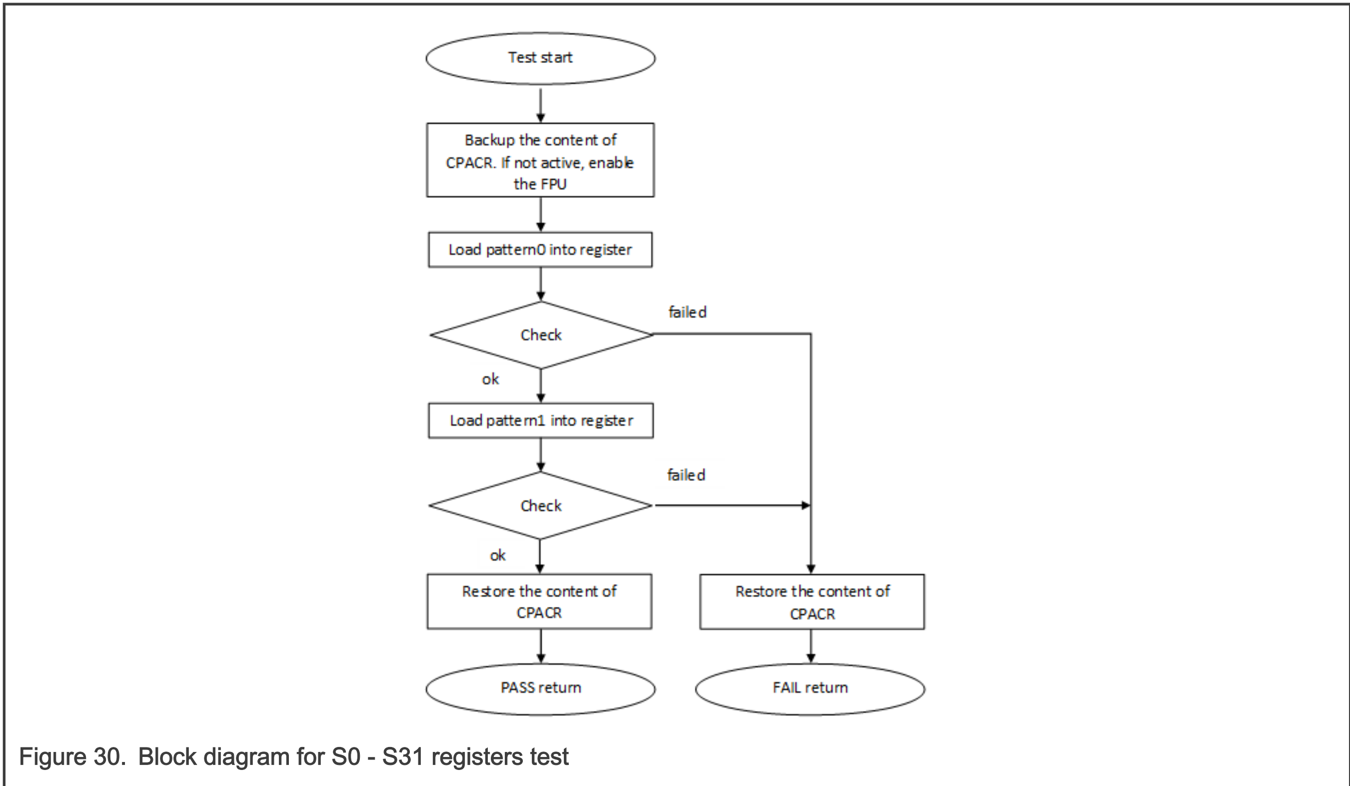

Figure 30. Block diagram for S0 - S31 registers test

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 16. 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_cm33_reg.S* file and written as assembler functions. For devices containing the FPU, *iec60730b_cm33 _reg_fpu.S* is an additional file with the tests of FPU-related registers. The header file with the return values and function prototypes is *iec60730b_cm33_reg.h*.

The *iec60730b.h* and *asm_mac_common.h* and *iec60730b_types.h* are the common header files for the safety library.

The following functions are called to test the corresponding registers:

- *FS_CM33_CPU_Register()*
- *FS_CM33_CPU_NonStackedRegister()*
- *FS_CM33_CPU_Primask_S()*
- *FS_CM33_CPU_Primask_NS()*
- *FS_CM33_CPU_SPmain_S()*
- *FS_CM33_CPU_SPmain_NS()*
- *FS_CM33_CPU_SPmain_Limit_S()*

- *FS_CM33_CPU_SPmain_Limit_NS()*
- *FS_CM33_CPU_SPprocess_S()*
- *FS_CM33_CPU_SPprocess_NS()*
- *FS_CM33_CPU_SPprocess_Limit_S()*
- *FS_CM33_CPU_SPprocess_Limit_NS()*
- *FS_CM33_CPU_Control()*
- *FS_CM33_CPU_Control_S()*
- *FS_CM33_CPU_Control_NS()*
- *FS_CM33_CPU_Special8PriorityLevels_S()*
- *FS_CM33_CPU_Special8PriorityLevels_NS()*

Functions with the **_S** postfix are intended to test the secure part of the tested register. Functions with the **_NS** postfix are intended to test the non-secure part of the tested register. The *FS_CM33_CPU_Register()* and *FS_CM33_CPU_NonStackedRegister()* functions do not have a postfix because the R0-R12, LR, and APSR registers are not banked between security states.

In case that TrustZone (TZ) is supported on the tested device, the CONTROL register is banked between security states. Therefore, it must be tested by the *FS_CM33_CPU_Control_S()* and *FS_CM33_CPU_Control_NS()* functions. When the TrustZone is not supported, the CONTROL register must be tested by the *FS_CM33_CPU_Control()* function. The list of functions which are called when the TZ is/isn't supported is located in the *<device> dedicated functions* chapter (for example *LPC55Sxx dedicated functions*).

When the device has an FPU, the following functions are placed in *iec60730b_cm33_reg_fpu.S*:

- *FS_CM33_CPU_Float1()*
- *FS_CM33_CPU_Float2()*

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 corrupt, the application is in an endless loop instead of returning an error value. If some of these registers are corrupt, the application cannot make standard operations to identify the safety error (to compare something, to move out from the function, or to return a value).

The use of functions after the reset and during runtime is the same. Be careful when using functions during runtime, as described in the following sections.

The following is an example of a function call:

```
#include “iec60730b.h”
if (FS_FAIL_CPU_REGISTER == FS_CM33_CPU_Register())
SafetyError();
```

8.2.1 FS_CM33_CPU_Control()

This function tests the CONTROL register according to the Figure 27. This function is intended to be executed only on devices without TrustZone support.

Function prototype:

FS_RESULT FS_CM33_CPU_Control(void);

Test pattern:

CONTROL: 0x00000002, 0x00000004

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_CONTROL*

Function performance:

The function takes approximately 47 cycles (0.49 μs). [1]

The function size is 50 B.[1]

Calling restrictions:

This function cannot be interrupted.

This test must be executed only on devices without TrustZone support.

8.2.2 FS_CM33_CPU_Control_NS()

This function tests the CONTROL_NS (Non-Secure) register according to the Figure 27.

Function prototype:

FS_RESULT FS_CM33_CPU_Control_NS(void);

Test pattern:

CONTROL_NS: 0x00000002, 0x00000004

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_CONTROL*

Function performance:

The function takes approximately 37 cycles (0.39 μs). [1]

The function size is 52 B.[1]

Calling restrictions:

This function cannot be interrupted.

The core must be in the secure state.

8.2.3 FS_CM33_CPU_Control_S()

This function tests the CONTROL (secure) register according to the Figure 27.

Function prototype:

FS_RESULT FS_CM33_CPU_Control_S(void);

Test pattern:

CONTROL: 0x0000000A, 0x00000004

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_CONTROL*

Function performance:

The function takes approximately 47 cycles (0.49 μs). [1]

The function size is 50 B.[1]

Calling restrictions:

This function cannot be interrupted.

The core must be in the secure state.

8.2.4 FS_CM33_CPU_Float1()

This function checks the FPSCR and S0-S15 registers according to the Figure 29 and Figure 30. Within the function, the FPU is enabled in the CPACR register. At the end of the function, the original content of CPACR is restored.

Function prototype:

FS_RESULT FS_CM33_CPU_Float1(void);

Test patterns for respective registers:

FPSCR: 0x55400015, 0xA280008A

S0-S15: 0x55555555, 0xAAAAAAAA

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_FLOAT_1*

Function performance:

The function takes approximately 201 cycles (2.1 μs). [1]

The function size is 204 B.[1]

Calling restrictions:

The core must be in the secure state.

Only for devices with the Floating Point Unit (FPU).

8.2.5 FS_CM33_CPU_Float2()

This function checks the S16-S31 registers according to the Figure 30. Within the function, the FPU is enabled in the CPACR register. At the end of the function, the original content of the CPACR is restored.

Function prototype:

FS_RESULT FS_CM33_CPU_Float2(void);

Test patterns for respective registers:

S16-S31: 0x55555555, 0xAAAAAAAA

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_FLOAT_2*

Function performance:

The function takes approximately 201 cycles (2.1 μs). [1]

The function size is 204 B.[1]

Calling restrictions:

The core must be in the secure state.

Only for devices with the Floating Point Unit (FPU).

8.2.6 FS_CM33_CPU_NonStackedRegister()

This function tests the following CPU registers in a sequence: R8, R9, R10, R11. Each register is tested according to the Figure 25

Function prototype:

FS_RESULT FS_CM33_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 75 cycles (0.78 μs). [1]

The function size is 80 B.[1]

Calling restrictions:

Can be executed in both the secure and non-secure modes.

8.2.7 FS_CM33_CPU_Primask_NS()

This function tests the PRIMASK_NS (Non-Secure) register according to the Figure 27.

Function prototype:

FS_RESULT FS_CM33_CPU_Primask_NS(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 (0.46 µs). [1]

The function size is 44 B.[1]

Calling restrictions:

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

The core must be in the secure state.

8.2.8 FS_CM33_CPU_Primask_S()

This function tests the PRIMASK_S (secure) register according to the Figure 27.

Function prototype:

FS_RESULT FS_CM33_CPU_Primask_S(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 38 cycles (0.40 µs). [1]

The function size is 44 B.[1]

Calling restrictions:

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

The core must be in the secure state.

8.2.9 FS_CM33_CPU_Register()

This function tests the following CPU registers in a sequence: R0-R7, R12, LR, APSR. Each register is tested according to Figure 25 the and Figure 26.

Function prototype:

FS_RESULT FS_CM33_CPU_Register(void);

Test patterns for respective registers:

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

APSR: 0x50050000, 0xA80A0000

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_REGISTER*

If R0, R1, LR, or APSR are corrupted, the function stucks 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 201 cycles (2.1 µs). [1]

The function size is 204 B.[1]

Calling restrictions:

Can be executed in secure or non-secure mode.

8.2.10 FS_CM33_CPU_Special8PriorityLevels_NS()

This function tests the BASEPRI_NS and FAULTMASK_NS (Non-Secure) registers according to the Figure 27.

Function prototype:

FS_RESULT FS_CM33_CPU_Special8PriorityLevels_NS(void);

Test pattern:

BASEPRI: 0xA0, 0x40

FAULTMASK: 0x1, 0x0

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_SPECIAL*

Function performance:

The function takes approximately 81 cycles (0.84 µs). [1]

The function size is 88 bytes.[1]

Calling restrictions:

This function cannot be interrupted.

For devices with eight priority levels for interrupts.

The core must be in the secure state.

8.2.11 FS_CM33_CPU_Special8PriorityLevels_S()

This function tests the BASEPRI and FAULTMASK (secure) registers according to the Figure 27.

Function prototype:

FS_RESULT FS_CM33_CPU_Special8PriorityLevels_S(void);

Test pattern:

BASEPRI: 0xA0, 0x40

FAULTMASK: 0x1, 0x0

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_SPECIAL*

Function performance:

The function takes approximately 81 cycles (0.84 µs). [1]

The function size is 88 B.[1]

Calling restrictions:

This function cannot be interrupted.

The core must be in the secure state.

For devices with eight priority levels for interrupts.

8.2.12 FS_CM33_CPU_SPmain_Limit_NS()

This function tests the MSPLIM_NS (Main Stack Pointer Limit, Non-Secure) register according to the Figure 28.

Function prototype:

FS_RESULT FS_CM33_CPU_SPmain_Limit_NS(void);

Test pattern:

MSPLIM_NS: 0x55555550, 0xAAAAAAA8

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*

If MSPLIM_NS is corrupted, the function stucks 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 49 cycles (0.51 µs). 1

The function size is 56 B.[1]

Calling restrictions:

This function cannot be interrupted.

The core must be in the secure state.

8.2.13 FS_CM33_CPU_SPmain_Limit_S()

This function tests the MSPLIM (Main Stack Pointer Limit, secure) register according to the Figure 28.

Function prototype:

FS_RESULT FS_CM33_CPU_SPmain_Limit_S(void);

<u>Test pattern:</u>

MSPLIM: 0x55555550, 0xAAAAAAA8

<u>Function inputs:</u>

void

<u>Function output:</u>

typedef uint32_t FS_RESULT;

- *FS_PASS*

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

<u>Function performance:</u>

The function takes approximately 57 cycles (0.59 µs). [1]

The function size is 56 B.[1]

<u>Calling restrictions:</u>

This function cannot be interrupted.

The core must be in the secure state.

8.2.14 FS_CM33_CPU_SPmain_NS()

This function tests the MSP_NS (Main Stack Pointer, Non-Secure) register according to the Figure 28.

<u>Function prototype:</u>

FS_RESULT FS_CM33_CPU_SPmain_NS(void);

<u>Test pattern:</u>

MSP_NS: 0x55555554, 0xAAAAAAA8

<u>Function inputs:</u>

void

<u>Function output:</u>

typedef uint32_t FS_RESULT;

- *FS_PASS*

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

<u>Function performance:</u>

The function takes approximately 34 cycles (0.35 µs). [1]

The function size is 56 B.[1]

<u>Calling restrictions:</u>

This function cannot be interrupted.

The core must be in the secure state.

8.2.15 FS_CM33_CPU_SPmain_S()

This function tests the MSP (Main Stack Pointer, secure) register according to the Figure 28.

Function prototype:

FS_RESULT FS_CM33_CPU_SPmain_S(void);

Test pattern:

MSP: 0x55555554, 0xAAAAAAA8

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*

If the MSP is corrupted, the function stucks 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 60 cycles (0.63 µs). [1]

The function size is 62 B.[1]

Calling restrictions:

This function cannot be interrupted.

The core must be in the secure state.

8.2.16 FS_CM33_CPU_SPprocess_Limit_NS()

This function tests the PSPLIM_NS (Process Stack Pointer Limit, Non-Secure) register according to the Figure 28.

Function prototype:

FS_RESULT FS_CM33_CPU_SPprocess_Limit_NS(void);

Test pattern:

PSPLIM_NS: 0x55555550, 0xAAAAAAA8

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*

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

Function performance:

The function duration is approximately 49 cycles (0.51 µs). [1]

The function size is 56 B.[1]

Calling restrictions:

This function cannot be interrupted.

The core must be in the secure state.

8.2.17 FS_CM33_CPU_SPprocess_Limit_S()

This function tests the PSPLIM (Process Stack Pointer Limit, secure) register according to the Figure 28.

Function prototype:

FS_RESULT FS_CM33_CPU_SPprocess_Limit_S(void);

Test pattern:

PSPLIM: 0x55555550, 0xAAAAAAA8

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*

If the PSPLIM is corrupted, the function stucks 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 57 cycles, including the result comparison (0.59 μs). [1]

The function size is 56 B.[1]

Calling restrictions:

This function cannot be interrupted.

The core must be in the secure state.

8.2.18 FS_CM33_CPU_SPprocess_NS()

This function tests the PSP_NS (Process Stack Pointer, Non-Secure) register according to the Figure 28.

Function prototype:

FS_RESULT FS_CM33_CPU_SPprocess_NS(void);

Test pattern:

PSP_NS: 0x55555554, 0xAAAAAAA8

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*

If the PSP_NS is corrupted, the function stucks 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 49 cycles, including the result comparison (0.51 μs). [1]

The function size is 56 B.[1]

Calling restrictions:

This function cannot be interrupted.

The core must be in the secure state.

8.2.19 FS_CM33_CPU_SPprocess_S()

This function tests the PSP (secure) register according to the Figure 28.

Function prototype:

FS_RESULT FS_CM33_CPU_SPprocess_S(void);

Test pattern:

PSP: 0x55555554, 0xAAAAAAA8

Function inputs:

void

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*

If the PSP is corrupted, the function stucks 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 57 cycles, including the result comparison (0.59 μs). [1]

The function size is 56 B.[1]

Calling restrictions:

This function cannot be interrupted.

The core must be in the secure state.

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, you must define the areas below and under the stack here as well. There are other methods to achieve this, but only one example is shown here. The size of the areas must be a multiple of 0x4. The minimal 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 they 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_cm33_stack.S* file and written as assembler functions. The header file with the return values and the function prototypes is *iec60730b_cm33_stack.h*. The *iec60730b.h* and *asm_mac_common.h* and *iec60730b_types.h* are the common header files for the safety library. The following sections show the example of the linker setup, process of initialization, and implementation.

9.3.1 FS_CM33_STACK_Init()

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

Example of initialization:

```
#include "iec60730b.h"

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_CM33_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.

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

Function output:

void

Function performance:

The function takes approximately 105 cycles (1.10 μs) for a block size of 0x10.[1]

The function size is 26 B.[1]

Calling restrictions:

None.

9.3.2 FS_CM33_STACK_Test()

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

Function prototype:

FS_RESULT FS_CM33_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 the areas under and above the stack.

Function output:

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_STACK*

Function performance:

The function takes approximately 139 cycles (1.45 µs) for a block size of 0x10.[1]

The function size is 41 B.[1]

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:

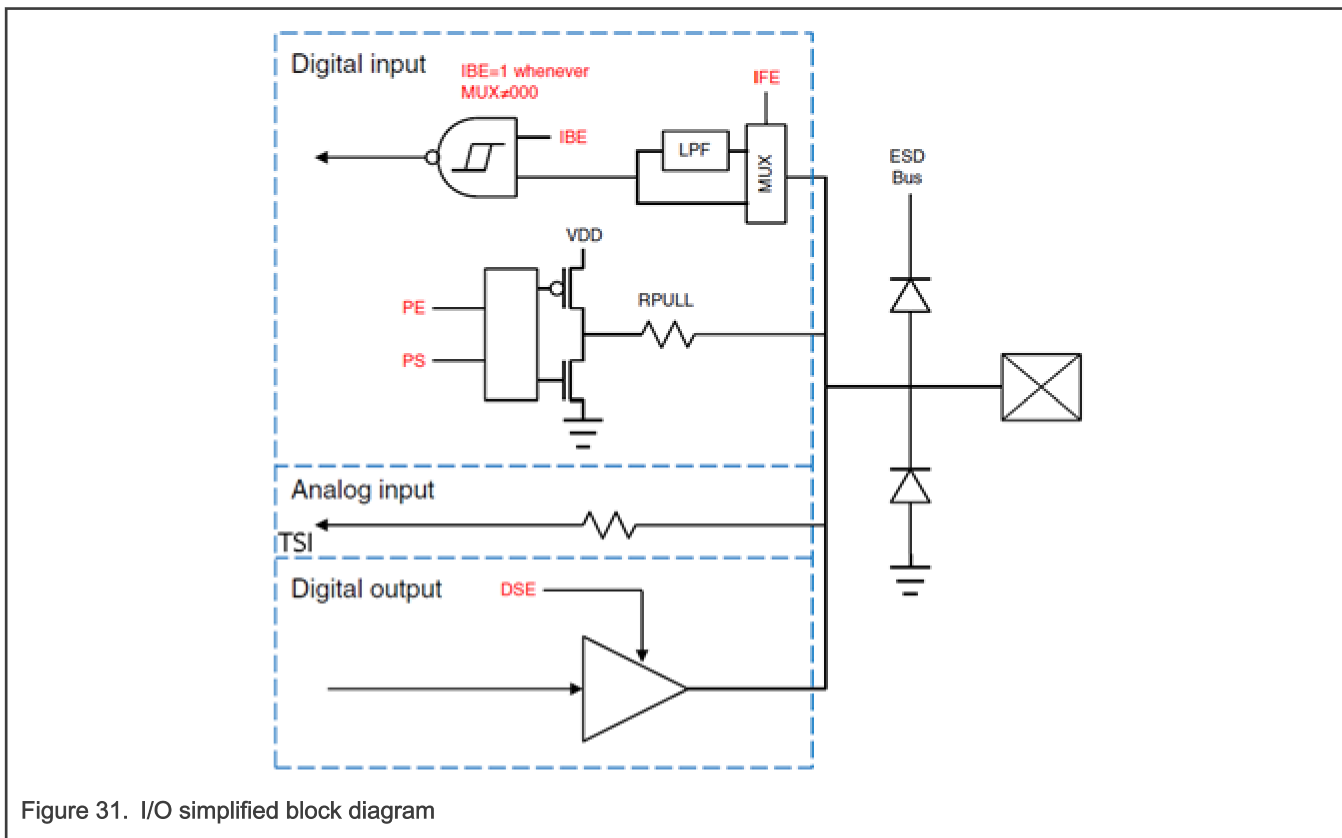

Figure 31. I/O simplified block diagram

10.1 TSI signal shorts tests

Because the analog TSI channels are shared with the digital I/O pins and the analog or digital features 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.

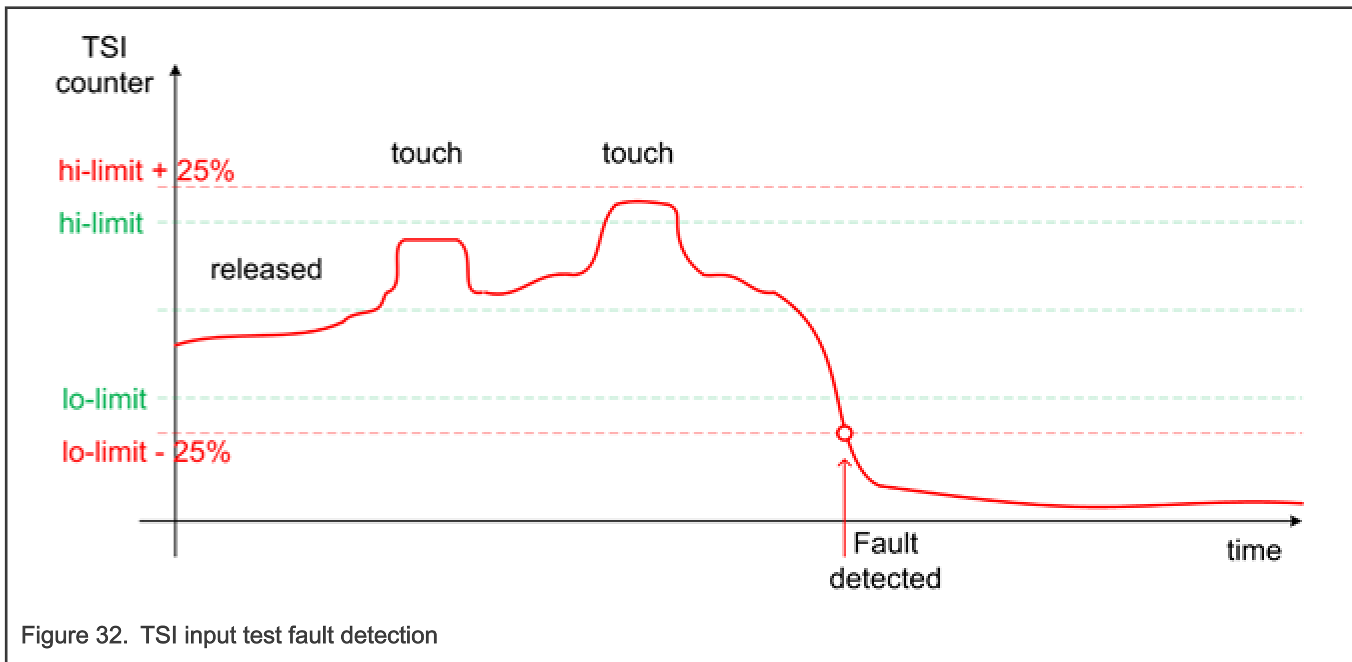

Figure 32. TSI input test fault detection

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:

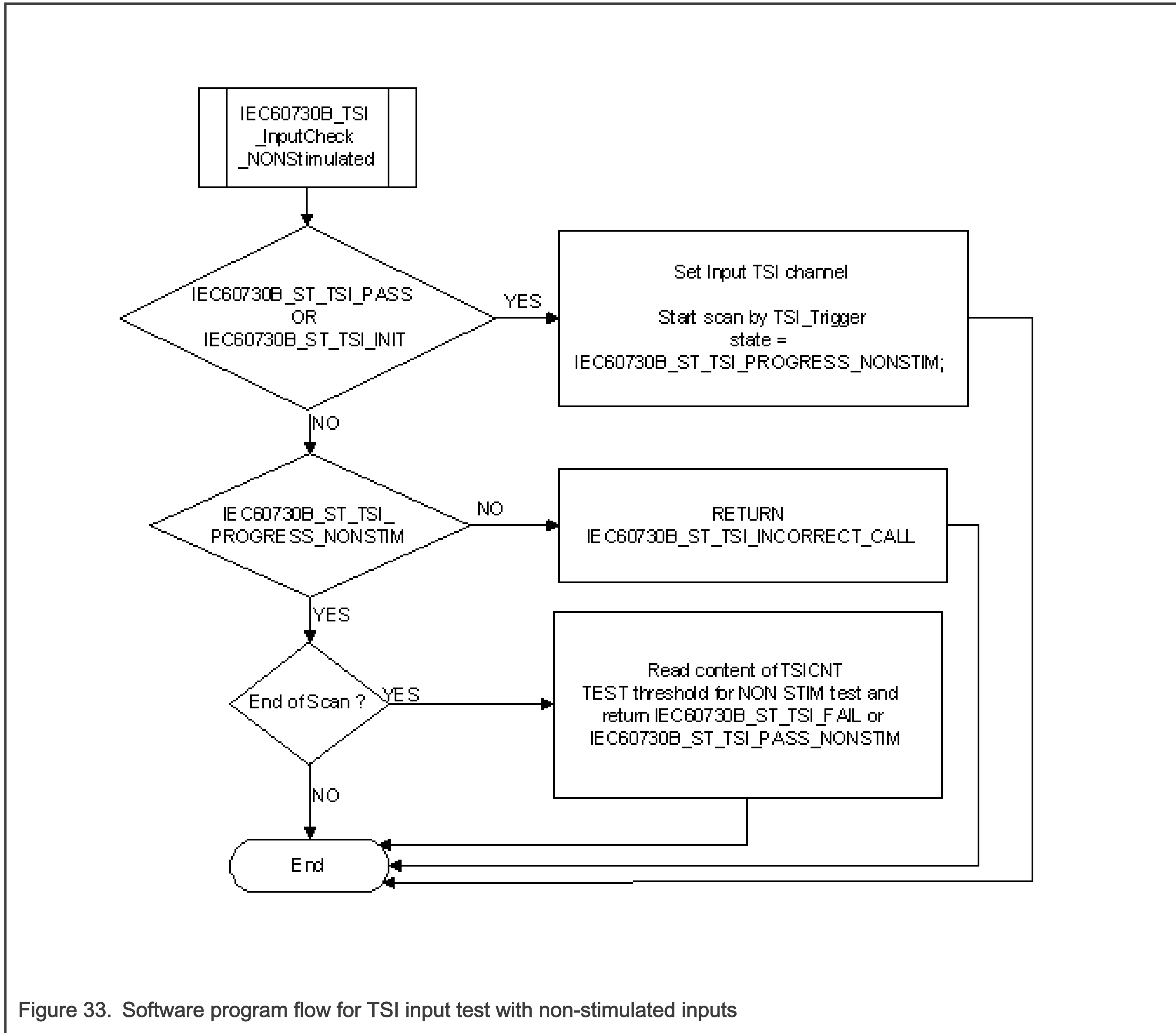

Figure 33. Software program flow for TSI input test with non-stimulated inputs

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.

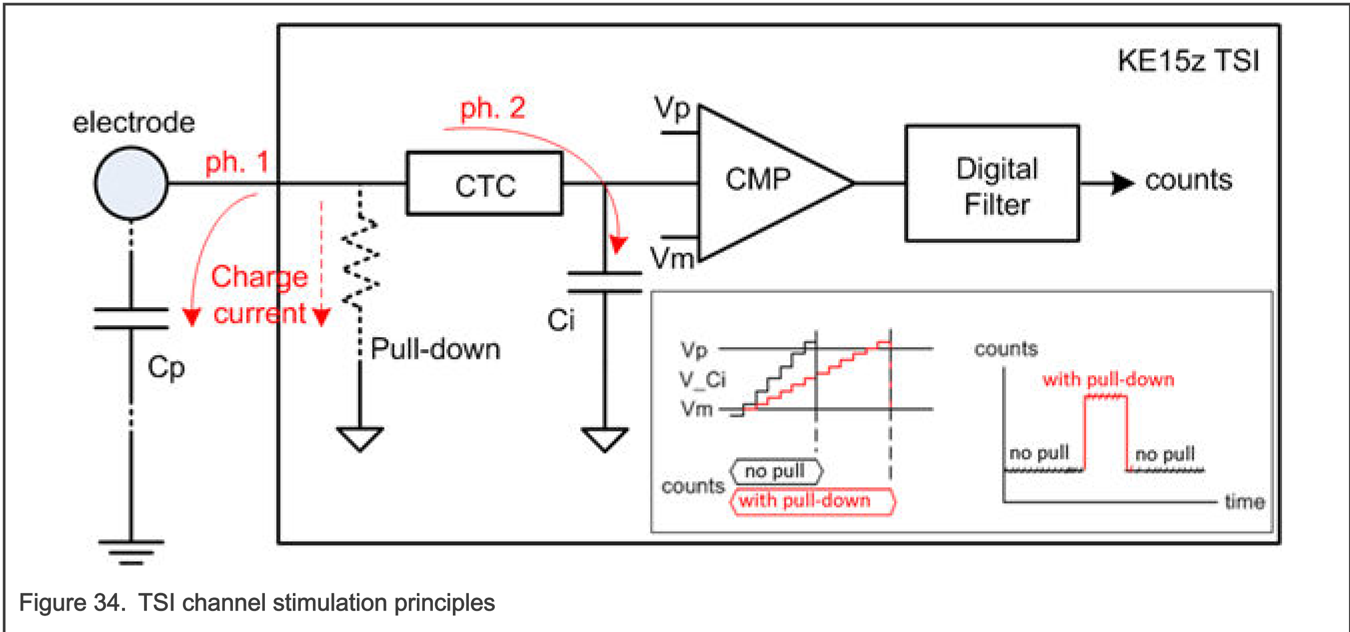

Figure 34. TSI channel stimulation principles

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 Cp 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 Cp is charged to a smaller level (and the charge accumulated by the Cp 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 Cp is transferred to the reference internal capacitor Ci. When the internal pull-up resistor is enabled, the charge steps are smaller. You need more charging steps to charge the Ci to the appropriate level. More charging steps result in longer time and higher count accumulated in the TSI result counter.

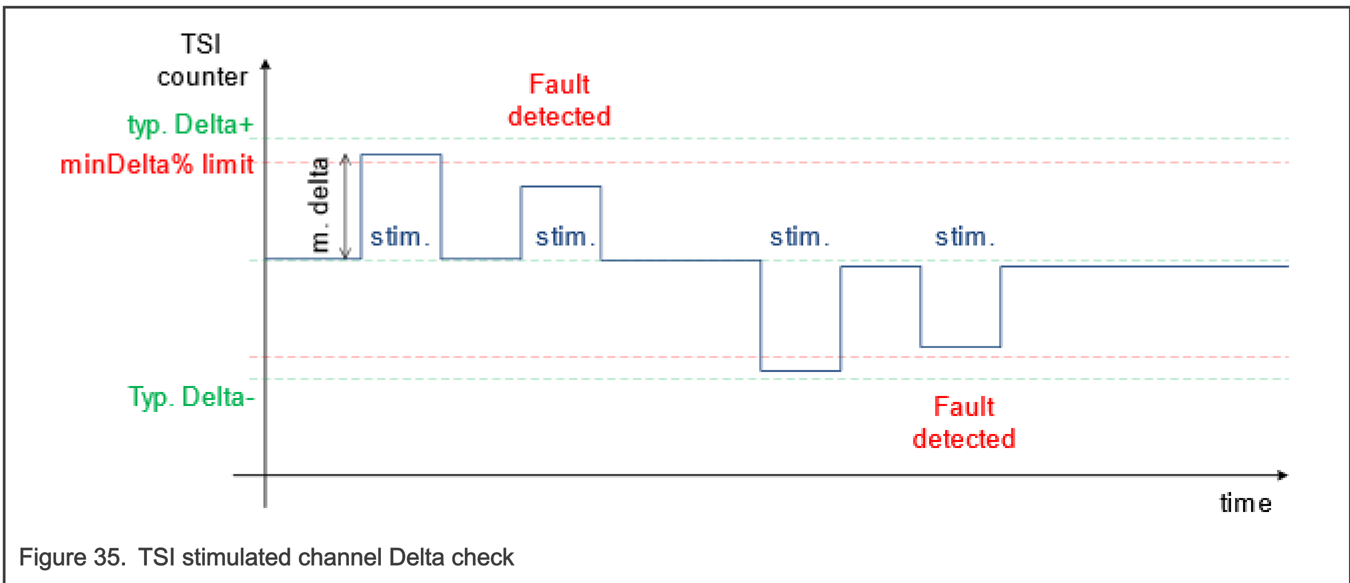

Figure 35. TSI stimulated channel Delta check

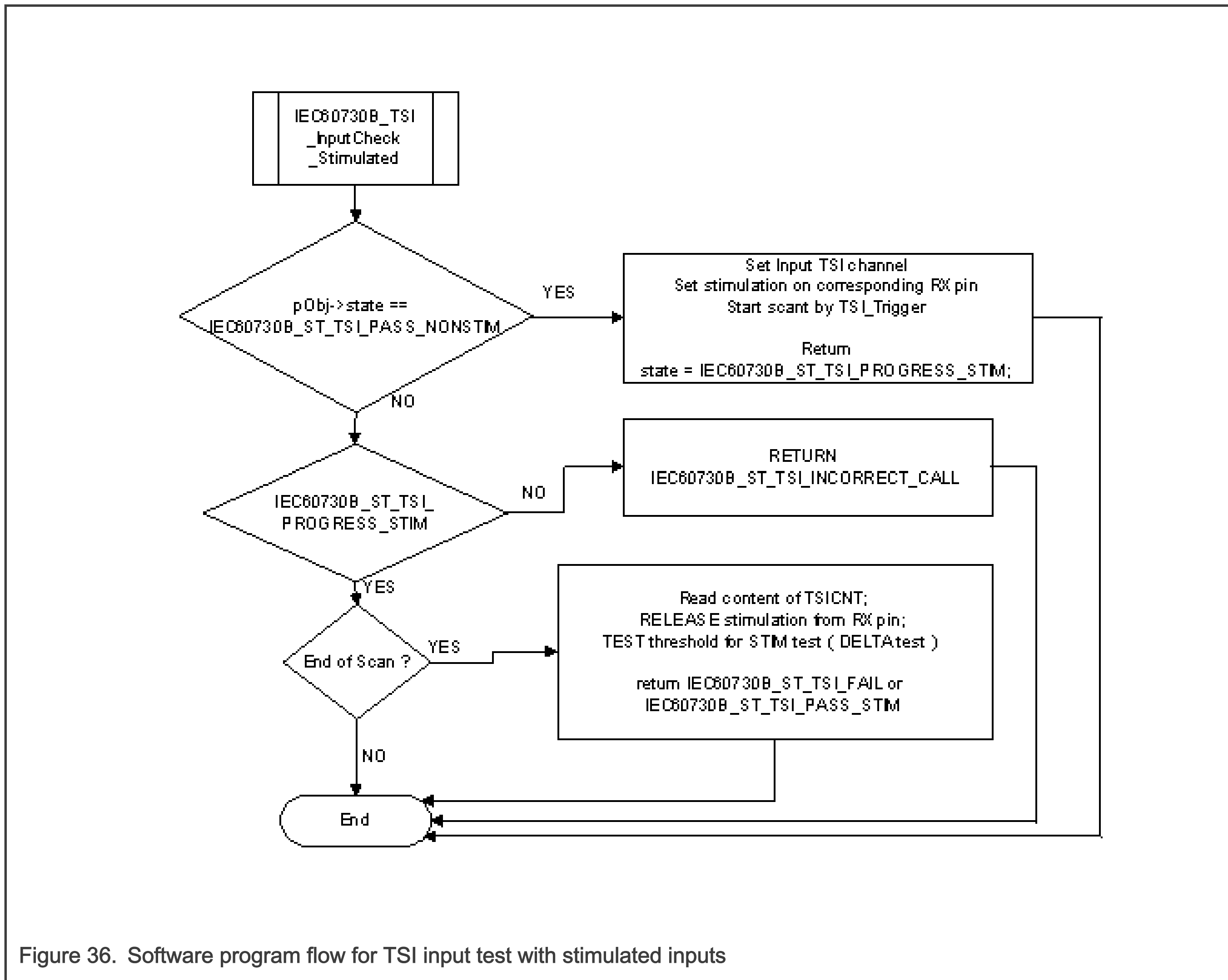

Figure 36. Software program flow for TSI input test with stimulated inputs

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*. *iec60730b.h* and *iec60730b_types.h* are the common header files 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 the 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_PASS;
 }
 if (pObj->state == FS_FAIL_TSI )
 { /*TEST FAIL */
 psSafetyCommon->TSI_test_result = FS_FAIL_TSI;
 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 the 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 address of 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 adress of 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 struncture.

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 Table 17.

Table 17. 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_wdog.c* file. The header file is *iec60730b_wdog.h*. The *iec60730b.h*, *iec60730b.h* and *iec60730b_types.h* are the common header files for the safety library.

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_t* type, which is a structure with three members. It is defined in the *iec60730b _wdog.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_ENABLED
#define Watchdog_refresh WDOG_REFRESH = 0xA602;WDOG_REFRESH = 0xB480
```

```
extern uint32_t WD_TEST_BACKUP; /* from Linker configuration file */
const uint32_t WD_backup_address = (uint32_t)&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
#define REFRESH_INDEX FS_KINETIS_WDOG
#define REG_WIDE FS_WDOG_SRS_WIDE_8b
#define CLEAR_FLAG 0

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, REFRESH_INDEX );
 }

 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, CLEAR_FLAG, REG_WIDE);
 }
```

11.2.1 FS_WDOG_Setup_LPTMR()

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_LPTMR(fs_wdog_test_t *pWatchdogBackup, uint8_t refresh_index)*

Function inputs:

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

refresh_index- The index to select the WDOG refresh sequence. Use the following macros: FS_KINETIS_WDOG, FS_WDOG32, or FS_COP_WDOG.

Function output:

void

Function performance:

For information about the function performance, see 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.

The "refresh_index" parameters must be filled corectly if your example application is set to a correct version. For other devices, compare the reference manual of your device with Table 18 or with the reference device in the following table.

Table 18. Refresh sequence

Refresh Index parameter	Refresh sequence	Reference device
FS_KINETIS_WDOG	• WdogBase->REFRESH = 0xA602U; • WdogBase->REFRESH = 0xB480U;/* refresh sequence */	MKV11
FS_WDOG32	WdogBase->CNT = 0xB480A602U; /* refresh sequence */	MK32L2A
FS_COP_WDOG	• WdogBase->SRVCOP = FS_SIM_KL2X_SRVCOP_SRVCOP(0x55U); • WdogBase->SRVCOP = FS_SIM_KL2X_SRVCOP_SRVCOP(0xAAU);	MKL26z

11.2.2 FS_WDOG_Setup_KE0XZ()

This function can be used for 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:

For information about the function performance, see Core self-test library – source code version.

Calling restrictions:

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

It is necessary to fill the following variables before calling the WDOG test:

fs_wdog_test_t * wdogBackup

- *wdogBackup->pResetDetectRegister* - The address of the "ResetDetect" register.
- *wdogBackup->ResetDetectMask* - The mask for the WDOG reset source (in the reset-detect register).
- *wdogBackup->RefTimerBase* - The base address of the RTC timer used.

- *wdogBackup->WdogBase* - The base address of the WDOG used.

11.2.3 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, uint8_t channel);*

Function inputs:

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

channel - The channel index of the MRT timer.

Function output:

void

Function performance:

For information about the function performance, see 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_t* type must be declared and placed into the RAM area that is not overwritten during application startup. Interrupts should be disabled.

It is necessary to fill the following variables before calling the WDOG test:

fs_wdog_test_t * wdogBackup

- *wdogBackup->pResetDetectRegister* - The address of the "ResetDetect" register.
- *wdogBackup->ResetDetectMask* - The mask for the WDOG reset source (in the reset-detect register).
- *wdogBackup->RefTimerBase* - The base address of the MRT timer used.
- *wdogBackup->WdogBase* - The base address of the WDOG used.

11.2.4 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);*

Function inputs:

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

Function output:

void

Function performance:

The duration of this function depends on the WDOG timeout, because the function waits in the WDOG reset. The size of function is 70 bytes.

Calling restrictions:

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

It is necessary to fill the following variables before calling the WDOG test:

fs_wdog_test_t * wdogBackup

- *wdogBackup->pResetDetectRegister* - The address of the "ResetDetect" register.
- *wdogBackup->ResetDetectMask* - The mask for the WDOG reset source (in the reset-detect register).
- *wdogBackup->RefTimerBase* - The base address of the CTIMER timer used.
- *wdogBackup->WdogBase* - The base address of the WDOG used.

11.2.5 FS_WDOG_Setup_IMX_GPT()

This function can be used for MIMXRT10xx and MIMX8Mini 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_IMX_GPT(fs_wdog_test_t *pWatchdogBackup, uint8_t refresh_index)*

Function inputs:

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

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

refresh_index - Select a correct refresh sequence for WDOG - FS_IMXRT or FS_IMX8M.

Function output:

void

Function performance:

For information about the function performance, see 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_t* type must be declared and placed into a reliable place. Interrupts should be disabled.

The "refresh_index" parameters must be filled correctly according to your example application version. For other devices, compare the reference manual of your device with the reference device. See Table 19.

Table 19. Refresh sequence

Refresh Index parameter	Refresh sequence	Reference device
FS_IMXRT	WdogBase->CNT = 0xB480A602U;	MIMXRT10xx
FS_IMX8M	• WdogBase->WSR = 0x5555; • WdogBase->WSR = 0xAAAA;	IMX8M

11.2.6 FS_WDOG_Check()

This function compares the captured value of the reference 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 and a corresponding return error returned. With the "endless_loop_enable" parameter, the endless loop within the function is enabled or disabled, (by setting it to 1 or 0). If the endless loop is disabled, the function returns a correspond error under the following condtions:

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

Function prototype:

*uint32_t FS_WDOG_Check(uint32_t limitHigh, uint32_t limitLow, uint32_t limitResets, bool_t endlessLoopEnable, fs_wdog_test_t *pWatchdogBackup, bool_t clear_flag, bool_t RegWide8b)*

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.

clear_flag- bool value, if TRUE, the WDOG reset flag from the reset detect register is deleted.

RegWide8b- When TRUE, the reset detect register is accesed as 8b, otherwise 32b.

Function output:

The function can stay in an endless loop if the "endlessLoopEnable" parameter is set to 1 or the return value:

FS_FAIL_WDOG_WRONG_RESET, *FS_FAIL_WDOG_VALUE*, *FS_FAIL_WDOG_OVER_RESET* or *FS_PASS*

Function performance:

For information about the function performance, see Core self-test library – source code version.

Calling restrictions:

The respective setup function must be executed first.

11.2.7 FS_WDOG_Check_WWDT_LPC()

This function can be used for the devices with the WWDT watchdog. 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 the endless loop is disabled, the function returns the corresponding error under the following condtions:

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

Function prototype:

*uint32_t 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 the endless loop within the function.

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

<u>Function output:</u>

The function can stay in the endless loop, if the "endlessLoopEnable" parameter is set to 1 or the return value:

FS_FAIL_WDOG_WRONG_RESET, *FS_FAIL_WDOG_VALUE*, *FS_FAIL_WDOG_OVER_RESET* or *FS_PASS*

<u>Function performance:</u>

For information about the function performance, see Core self-test library – source code version.

<u>Calling restrictions:</u>

The respective setup function must be executed first.

If necessary, fill these variables before calling the WDOG test:

fs_wdog_test_t * wdogBackup

- *wdogBackup->pResetDetectRegister* - The address of the "ResetDetect" register.
- *wdogBackup->ResetDetectMask* - The mask for the WDOG reset source (in the reset-detect register).
- *wdogBackup->RefTimerBase* - The base address of the timer used.
- *wdogBackup->WdogBase* - The base address of the WDOG used.

11.2.8 FS_WDOG_Check_WWDT_LPC55SXX()

This function can be used for 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 the endless loop is disabled, the function returns the corresponding error under the following condtions:

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

<u>Function prototype:</u>

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

<u>Function inputs:</u>

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.

<u>Function output:</u>

The function can stay in the endless loop - if the "endlessLoopEnable" parameter is set to 1 or the return value:

FS_FAIL_WDOG_WRONG_RESET, *FS_FAIL_WDOG_VALUE*, *FS_FAIL_WDOG_OVER_RESET* or *FS_PASS*

<u>Function performance:</u>

For information about the function performance, see Core self-test library – source code version.

<u>Calling restrictions:</u>

The respective setup function must be executed first.

It Is necessary to fill these variables before calling the WDOG test:

fs_wdog_test_t * wdogBackup

- *wdogBackup->pResetDetectRegister* - The address of the "ResetDetect" register.
- *wdogBackup->ResetDetectMask* - The mask for the WDOG reset source (in the reset-detect register).
- *wdogBackup->RefTimerBase* - The base address of the timer used.
- *wdogBackup->WdogBase* - The base address of the WDOG used.

How To Reach Us

Home Page:

nxp.com

Web Support:

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.

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, μVision, 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. 2021. All rights reserved.

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

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

Date of release: 12/2020

Document identifier: IEC60730BCM33L41UG