

# UG10315

IEC60730\_B\_CM4\_CM7\_Library\_UG\_v5\_0: IEC60730B Library User's Guide

Rev. 1.0 — 1 September 2025

User guide

## Document information

Information	Content
Keywords	IEC 60730, IEC 60335, UL 60730, UL 1998
Abstract	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).



## 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:

### 1.1 Core-dependent part

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

### 1.2 Peripheral-dependent part

- Clock test
- Digital input/output test
- Analog input/output test
- Watchdog test
- Touch-sensing interface test (only for the TSlv5 and TSlv6 peripherals)

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

The library supports the MCXE24x, MCXE31x, MKV3x, MKV4x, MKV5x, MKE1xF, MK2xF, K32L3Ax, LPC54S0x, LPC540x, MIMXRT10xx, MIMXRT117x, MIMXRT116x, MIMXRT118x, MIMX8MNx, MIMX8MLx, and MIMX8MMx families based on the Arm-CM4 or Arm-CM7 cores.

### 1.3 Core self-test library – source code

The library name is IEC60730B\_CM4\_CM7. The main header files are *iec60730b.h* and *iec60730b\_core.h*. All the data types necessary for the library are defined in the *iec60730b\_types.h* file.

Library could be configured by *device\_information.h*, header file.

This file allows developers to configure support for specific hardware features such as the Floating Point Unit (FPU), Digital Signal Processor (DSP), and TrustZone (TZ), depending on the capabilities of the target device.

The following preprocessor macros are defined in the file:

FPU\_SUPPORT: Enables/disables FPU-related code

DSP\_SUPPORT: Enables/disables DSP-related code

TZ\_SUPPORT: Enables/disables TrustZone-related code

These macros allow conditional compilation of library functions that rely on these features. In dedicated functions chapter you can found configuration for your device family.

```

/*
 * Copyright 2025 NXP.
 * All rights reserved.
 *
 * SPDX-License-Identifier: BSD-3-Clause

```

```

*/

#ifndef _DEVICE_INFORMATION_H_
#define _DEVICE_INFORMATION_H_

#define FPU_SUPPORT 1
#define DSP_SUPPORT 0
#define TZ_SUPPORT 0

#endif /* _DEVICE_INFORMATION_H_ */

```

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

Table 1. List of library items

File name	Test type	Function name	Functions size [bytes]	Functions duration approximately [μs]
iec60730b.h	Library header file	-		
iec60730b_core.h	Core-dependent library header file	-		
iec60730b_types.h	Data types for the library	-		
asm_mac_common.h	Common assembler directives	-		
iec60730b_aio.c	Analog I/O test	FS_AIO_LimitCheck()	52 <sup>3</sup>	1.02 <sup>3</sup>
	Analog I/O test	FS_AIO_InputSet_A1()	90 <sup>6</sup>	1.41 <sup>6</sup>
	Analog I/O test	FS_AIO_InputSet_A23()	40 <sup>1</sup>	0.66 <sup>1</sup>
	Analog I/O test	FS_AIO_InputSet_A4()	40 <sup>3</sup>	0.89 <sup>3</sup>
	Analog I/O test	FS_AIO_InputSet_A4_2()	-	
	Analog I/O test	FS_AIO_InputSet_A5()	112 <sup>7</sup>	19.08 <sup>7</sup>
	Analog I/O test	FS_AIO_InputSet_A6()	44 <sup>4</sup>	0.14 <sup>4</sup>
	Analog I/O test	FS_AIO_InputSet_A7()	114 <sup>2</sup>	1.88 <sup>2</sup>
	Analog I/O test	FS_AIO_InputSet_A8()	40 <sup>9</sup>	0.94 <sup>9</sup>
	Analog I/O test	FS_AIO_ReadResult_A1()	44 <sup>6</sup>	1.69 <sup>6</sup>
	Analog I/O test	FS_AIO_ReadResult_A23()	32 <sup>1</sup>	0.65 <sup>1</sup>
	Analog I/O test	FS_AIO_ReadResult_A4()	32 <sup>3</sup>	0.79 <sup>3</sup>
	Analog I/O test	FS_AIO_ReadResult_A5()	48 <sup>7</sup>	14.42 <sup>7</sup>
	Analog I/O test	FS_AIO_ReadResult_A6()	112 <sup>4</sup>	1.71 <sup>4</sup>
	Analog I/O test	FS_AIO_ReadResult_A7()	136 <sup>2</sup>	0.55 <sup>2</sup>

Table 1. List of library items...continued

File name	Test type	Function name	Functions size [bytes]	Functions duration approximately [µs]
	Analog I/O test	FS_AIO_ReadResult_A8()	32 <sup>9</sup>	0.81 <sup>9</sup>
iec60730b_clock.c	Clock test	FS_CLK_Check()	44 <sup>1</sup>	0.51 <sup>1</sup>
	Clock test	FS_CLK_Init()	8 <sup>1</sup>	0.23 <sup>1</sup>
	Clock test	FS_CLK_LPTMR()	12 <sup>1</sup>	1.68 <sup>1</sup>
	Clock test	FS_CLK_RTC()	-	-
	Clock test	FS_CLK_GPT()	12 <sup>4</sup>	2.16 <sup>4</sup>
	Clock test	FS_CLK_WKT_LPC()	-	
	Clock test	FS_CLK_TIMER()	-	-
	Clock test	FS_CLK_STM()	12 <sup>8</sup>	0.21 <sup>8</sup>
iec60730b_dio.c	Digital I/O test	FS_DIO_Input()	-	-
	Digital I/O test	FS_DIO_Output()	126 <sup>1</sup>	17.4 (delay=100) <sup>1</sup>
	Digital I/O test	FS_DIO_Output_IMXRT()	124 <sup>4</sup>	94.33 (delay=3500) <sup>4</sup>
	Digital I/O test	FS_DIO_Output_IMX8M()	130 <sup>5</sup>	71.1 (delay=2000) <sup>5</sup>
	Digital I/O test	FS_DIO_Output_LPC()	156 <sup>7</sup>	34.65 (delay=75) <sup>7</sup>
iec60730b_dio_ext.c	Extended digital I/O test	FS_DIO_InputExt()	228 <sup>1</sup>	1.78 <sup>1</sup>
	Extended digital I/O test	FS_DIO_ShortToSupplySet()	152 <sup>1</sup>	1.24 <sup>1</sup>
	Extended digital I/O test	FS_DIO_ShortToAdjSet()	288 <sup>1</sup>	2.23 <sup>1</sup>
	Extended digital I/O test	FS_DIO_InputExt_IMXRT()	278 <sup>4</sup>	0.86 <sup>4</sup>
	Extended digital I/O test	FS_DIO_ShortToSupplySet_IMXRT()	130 <sup>4</sup>	2.00 <sup>4</sup>
	Extended digital I/O test	FS_DIO_ShortToAdjSet_IMXRT()	232 <sup>4</sup>	1.76 <sup>4</sup>
	Extended digital I/O test	FS_DIO_InputExt_IMX8M()	294 <sup>5</sup>	13.77 <sup>5</sup>
	Extended digital I/O test	FS_DIO_ShortToSupplySet_IMX8M()	156 <sup>5</sup>	13.21 <sup>5</sup>
	Extended digital I/O test	FS_DIO_ShortToAdjSet_IMX8M()	280 <sup>5</sup>	23.25 <sup>5</sup>

Table 1. List of library items...continued

File name	Test type	Function name	Functions size [bytes]	Functions duration approximately [ $\mu$ s]
	Extended digital I/O test	FS_DIO_InputExt_LPC()	180 <sup>Z</sup>	21.04 <sup>Z</sup>
	Extended digital I/O test	FS_DIO_ShortToSupplySet_LPC()	130 <sup>Z</sup>	21.79 <sup>Z</sup>
	Extended digital I/O test	FS_DIO_ShortToAdjSet_LPC()	254 <sup>Z</sup>	35.3 <sup>Z</sup>
	Extended digital I/O test	FS_DIO_InputExt_MCX()	-	-
	Extended digital I/O test	FS_DIO_ShortToSupplySet_MCX()	-	-
	Extended digital I/O test	FS_DIO_ShortToAdjSet_MCX()	-	-
	Extended digital I/O test	FS_DIO_InputExt_SIUL2()	132 <sup>B</sup>	0.78 <sup>B</sup>
	Extended digital I/O test	FS_DIO_ShortToSupplySet_SIUL2()	218 <sup>B</sup>	1.64 <sup>B</sup>
	Extended digital I/O test	FS_DIO_ShortToAdjSet_SIUL2()	322 <sup>B</sup>	2.19 <sup>B</sup>
	Extended digital I/O test	FS_DIO_InputExt_RGPIIO()	-	-
	Extended digital I/O test	FS_DIO_ShortToSupplySet_RGPIIO()	-	-
	Extended digital I/O test	FS_DIO_ShortToAdjSet_RGPIIO()	-	-
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()	-	-
	Touch-sensing interface test	FS_TSI_InputStimulate_v6()	-	-
	Touch-sensing interface test	FS_TSI_InputRelease_v6()	-	-
	Touch-sensing interface test	FS_TSI_InputCheckNONStimulated_v6()	-	-
	Touch-sensing interface test	FS_TSI_InputCheckStimulated_v6()	-	-

Table 1. List of library items...continued

File name	Test type	Function name	Functions size [bytes]	Functions duration approximately [µs]
iec60730b_invariable_memory.c	Invariable memory test (Flash)	FS_FLASH_C_HW16_K()	<a href="#">See the function dedicated chapter</a>	
iec60730b_invariable_memory.c	Invariable memory test (Flash)	FS_FLASH_C_HW32_K()	<a href="#">See the function dedicated chapter</a>	
	Invariable memory test (Flash)	FS_FLASH_C_HW16_L()	<a href="#">See the function dedicated chapter</a>	
	Invariable memory test (Flash)	FS_CM4_CM7_FLASH_HW32_DCP()	<a href="#">See the function dedicated chapter</a>	
iec60730b_cm4_cm7_flash.S	Invariable memory test (Flash)	FS_CM4_CM7_FLASH_HW16()	<a href="#">See the function dedicated chapter</a>	
	Invariable memory test (Flash)	FS_CM4_CM7_FLASH_SW16()	<a href="#">See the function dedicated chapter</a>	
	Invariable memory test (Flash)	FS_CM4_CM7_FLASH_SW32()	<a href="#">See the function dedicated chapter</a>	
iec60730b_cm4_cm7_pc.S	Program counter test	FS_CM4_CM7_PC_Test()	<a href="#">See the function dedicated chapter</a>	
iec60730b_cm4_cm7_pc_object.S	Program counter test	FS_PC_Object()	<a href="#">See the function dedicated chapter</a>	
iec60730b_cm4_cm7_ram.S	Variable memory test (RAM)	FS_CM4_CM7_RAM_AfterReset()	<a href="#">See the function dedicated chapter</a>	
	Variable memory test (RAM)	FS_CM4_CM7_RAM_Runtime()	<a href="#">See the function dedicated chapter</a>	
	Variable memory test (RAM)	FS_CM4_CM7_RAM_CopyToBackup()	<a href="#">See the function dedicated chapter</a>	
	Variable memory test (RAM)	FS_CM4_CM7_RAM_CopyFromBackup()	<a href="#">See the function dedicated chapter</a>	
	Variable memory test (RAM)	FS_CM4_CM7_RAM_SegmentMarchC()	<a href="#">See the function dedicated chapter</a>	
	Variable memory test (RAM)	FS_CM4_CM7_RAM_SegmentMarchX()	<a href="#">See the function dedicated chapter</a>	
iec60730b_cm4_cm7_reg.S	Register test	FS_CM4_CM7_CPU_Register()	<a href="#">See the function dedicated chapter</a>	
	Register test	FS_CM4_CM7_CPU_NonStackedRegister()	<a href="#">See the function dedicated chapter</a>	
	Register test	FS_CM4_CM7_CPU_Primask()	<a href="#">See the function dedicated chapter</a>	
	Register test	FS_CM4_CM7_CPU_SPmain()	<a href="#">See the function dedicated chapter</a>	
	Register test	FS_CM4_CM7_CPU_SPprocess()	<a href="#">See the function dedicated chapter</a>	
	Register test	FS_CM4_CM7_CPU_Control()	<a href="#">See the function dedicated chapter</a>	
	Register test	FS_CM4_CM7_CPU_Special()	<a href="#">See the function dedicated chapter</a>	
	Register test	FS_CM4_CM7_CPU_Special8PriorityLevels()	<a href="#">See the function dedicated chapter</a>	

Table 1. List of library items...continued

File name	Test type	Function name	Functions size [bytes]	Functions duration approximately [µs]
iec60730b_cm4_cm7_reg_fpu.S	Register test	FS_CM4_CM7_CPU_ControlFpu()	<a href="#">See the function dedicated chapter</a>	
	Register test	FS_CM4_CM7_CPU_Float1()	<a href="#">See the function dedicated chapter</a>	
	Register test	FS_CM4_CM7_CPU_Float2()	<a href="#">See the function dedicated chapter</a>	
iec60730b_cm4_cm7_stack.S	Stack test	FS_CM4_CM7_STACK_Init()	<a href="#">See the function dedicated chapter</a>	
	Stack test	FS_CM4_CM7_STACK_Test()	<a href="#">See the function dedicated chapter</a>	
iec60730b_wdog.c	Watchdog test	FS_WDOG_Setup_LPTMR()	90 <sup>1</sup>	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()	64 <sup>5</sup>	Duration time depends on the WDOG timeout
	Watchdog test	FS_WDOG_Setup_WWDT_CTIMER()	-	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()	188 <sup>1</sup>	1.2 <sup>1</sup>
	Watchdog test	FS_WDOG_Check_WWDT_LPC()	-	-
	Watchdog test	FS_WDOG_Check_WWDT_LPC55SXX()	-	-
	Watchdog test	FS_WDOG_Check_WWDT_MCX()	-	-

1.3.1 MIMX8MMx dedicated functions

For MIMX8MMx family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1
- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 2](#) shows the list of functions dedicated for the MIMX8M Mini device family.

Table 2. MIMX8MMx dedicated functions

File	Suitable function
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_GPT()"</a>

Table 2. MIMX8MMx dedicated functions...continued

File	Suitable function
iec60730b_dio.c	<a href="#">Section "FS_DIO_Output_IMX8M()"</a>
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt_IMX8M()"</a>
	<a href="#">Section "FS_DIO_ShortToSupplySet_IMX8M()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet_IMX8M()"</a>
iec60730b_wdog.c	<a href="#">Section "FS_WDOG_Setup_IMX_GPT()"</a> refresh_index = "FS_IMX8M"
	<a href="#">Section "FS_WDOG_Check()"</a> RegWide8b = "FS_WDOG_SRS_WIDE_32b"
iec60730b_cm4_cm7_flash.S	<a href="#">Section "FS_CM4_CM7_FLASH_SW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW32()"</a>
iec60730b_cm4_cm7_pc.S	Common for all CM4/CM7 devices
iec60730b_cm4_cm7_ram.S	Common for all CM4/CM7 devices
iec60730b_cm4_cm7_reg.S	Common for all CM4/CM7 devices
iec60730b_cm4_cm7_stack.S	Common for all CM4/CM7 devices

### 1.3.2 MIMX8MNx and MIMX8MLx dedicated functions

For MIMX8MNx and MIMX8MLx family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1
- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 3](#) shows the list of functions dedicated for the MIMX8M Nano and MIMX8M Plus device families.

Table 3. MIMX8MNx and MIMX8MLx dedicated functions

File	Suitable function
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_GPT()"</a>
iec60730b_dio.c	<a href="#">Section "FS_DIO_Output_IMX8M()"</a>
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt_IMX8M()"</a>
	<a href="#">Section "FS_DIO_ShortToSupplySet_IMX8M()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet_IMX8M()"</a>
iec60730b_wdog.c	<a href="#">Section "FS_WDOG_Setup_IMX_GPT()"</a> refresh_index = "FS_IMX8M"
	<a href="#">Section "FS_WDOG_Check()"</a> RegWide8b = "FS_WDOG_SRS_WIDE_32b"
iec60730b_cm4_cm7_flash.S	<a href="#">Section "FS_CM4_CM7_FLASH_SW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW32()"</a>
iec60730b_cm4_cm7_pc.S	Common for all CM4/CM7 devices
iec60730b_cm4_cm7_ram.S	Common for all CM4/CM7 devices
iec60730b_cm4_cm7_reg.S	Common for all CM4/CM7 devices

Table 3. MIMX8MNx and MIMX8MLx dedicated functions...continued

File	Suitable function
iec60730b_cm4_cm7_stack.S	<a href="#">Common for all CM4/CM7 devices</a>

### 1.3.3 MIMXRT10xx dedicated functions

For MIMXRT10xx family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1
- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 4](#) shows the list of functions dedicated for the MIMXRT10xx device family.

Table 4. MIMXRT10xx dedicated functions

File	Suitable function
iec60730b_aio.c	<a href="#">Section "FS_AIO_LimitCheck()"</a>
	<a href="#">Section "FS_AIO_InputSet_A6()"</a>
	<a href="#">Section "FS_AIO_ReadResult_A6()"</a>
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_GPT()"</a>
iec60730b_dio.c	<a href="#">Section "FS_DIO_Output_IMXRT()"</a>
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt_IMXRT()"</a>
	<a href="#">Section "FS_DIO_ShortToSupplySet_IMXRT()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet_IMXRT()"</a>
iec60730b_wdog.c	<a href="#">Section "FS_WDOG_Setup_IMX_GPT()"</a> refresh_index = "FS_IMXRT"
	<a href="#">Section "FS_WDOG_Check()"</a> RegWide8b = "FS_WDOG_SRS_WIDE_32b"
iec60730b_cm4_cm7_flash.S	<a href="#">Section "FS_CM4_CM7_FLASH_HW32_DCP()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW32()"</a>
iec60730b_cm4_cm7_pc.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_ram.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_reg.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_stack.S	<a href="#">Common for all CM4/CM7 devices</a>

### 1.3.4 MIMXRT117x/116x dedicated functions

For MIMXRT117x/116x family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1
- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 5](#) shows the list of functions dedicated for the MIMXRT117x and MIMXRT116x device families.

Table 5. MIMXRT117x/116x dedicated functions

File	Suitable function
iec60730b_aio.c	<a href="#">Section "FS_AIO_LimitCheck()"</a>
	<a href="#">Section "FS_AIO_InputSet_A1()"</a>
	<a href="#">Section "FS_AIO_ReadResult_A1()"</a>
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_GPT()"</a>
iec60730b_dio.c	<a href="#">Section "FS_DIO_Output_IMXRT()"</a>
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt_IMXRT()"</a>
	<a href="#">Section "FS_DIO_ShortToSupplySet_IMXRT()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet_IMXRT()"</a>
iec60730b_wdog.c	<a href="#">Section "FS_WDOG_Setup_IMX_GPT()"</a> refresh_index = "FS_IMXRTWDOG"
	<a href="#">Section "FS_WDOG_Check()"</a> RegWide8b = "FS_WDOG_SRS_WIDE_32b"
iec60730b_cm4_cm7_flash.S	<a href="#">Section "FS_CM4_CM7_FLASH_SW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW32()"</a>
iec60730b_cm4_cm7_pc.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_ram.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_reg.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_stack.S	<a href="#">Common for all CM4/CM7 devices</a>

### 1.3.5 MIMXRT118x CM7 dedicated functions

For MIMXRT118x CM7 family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1
- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 6](#) shows the list of functions dedicated for the MIMXRT118x CM7 device.

**Note:** For the RGPIO peripheral, it is not possible to solve the "GPIO\_Output" test directly. It is necessary to use a test against an adjacent pin through.

Table 6. MIMXRT118x CM7 dedicated functions

File	Suitable function
iec60730b_aio.c	<a href="#">Section "FS_AIO_LimitCheck()"</a>
	<a href="#">Section "FS_AIO_InputSet_A1()"</a>
	<a href="#">Section "FS_AIO_ReadResult_A1()"</a>
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_GPT()"</a>

Table 6. MIMXRT118x CM7 dedicated functions...continued

File	Suitable function
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt_RGPIO()"</a>
	<a href="#">Section "FS_DIO_ShortToSupplySet_RGPIO()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet_RGPIO()"</a>
iec60730b_wdog.c	Not supported in this release.
iec60730b_cm4_cm7_flash.S	<a href="#">Section "FS_CM4_CM7_FLASH_SW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW32()"</a>
iec60730b_cm4_cm7_pc.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_ram.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_reg.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_stack.S	<a href="#">Common for all CM4/CM7 devices</a>

### 1.3.6 MK2xF dedicated functions

For MK2xF family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1
- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 7](#) shows the list of functions dedicated for the MK2xF device.

Table 7. MK2xF dedicated functions

File	Suitable function
iec60730b_aio.c	<a href="#">Section "FS_AIO_LimitCheck()"</a>
	<a href="#">Section "FS_AIO_InputSet_A23()"</a>
	<a href="#">Section "FS_AIO_ReadResult_A23()"</a>
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_LPTMR()"</a>
iec60730b_dio.c	<a href="#">Section "FS_DIO_Output()"</a>
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt()"</a>
	<a href="#">Section "FS_DIO_ShortToSupplySet()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet()"</a>
iec60730b_wdog.c	<a href="#">Section "FS_WDOG_Setup_LPTMR()"</a> refresh_index = "FS_KINETIS_WDOG"
	<a href="#">Section "FS_WDOG_Check()"</a> RegWide8b = "FS_WDOG_SRS_WIDE_8b"
iec60730b_invariable_memory.c	<a href="#">Section "FS_FLASH_C_HW16_K()"</a>
iec60730b_cm4_cm7_flash.S	<a href="#">Section "FS_CM4_CM7_FLASH_HW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW32()"</a>

Table 7. MK2xF dedicated functions...continued

File	Suitable function
iec60730b_cm4_cm7_pc.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_ram.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_reg.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_stack.S	<a href="#">Common for all CM4/CM7 devices</a>

### 1.3.7 MKE1xF dedicated functions

For MKE1xF family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1
- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 8](#) shows the list of functions dedicated for the MKE1xF device.

Table 8. MKE1xF dedicated functions

File	Suitable function
iec60730b_aio.c	<a href="#">Section "FS_AIO_LimitCheck()"</a>
	<a href="#">Section "FS_AIO_InputSet_A4()"</a>
	<a href="#">Section "FS_AIO_ReadResult_A4()"</a>
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_LPTMR()"</a>
iec60730b_dio.c	<a href="#">Section "FS_DIO_Output()"</a>
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt()"</a>
	<a href="#">Section "FS_DIO_ShortToSupplySet()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet()"</a>
iec60730b_wdog.c	<a href="#">Section "FS_WDOG_Setup_LPTMR()"</a> refresh_index = "FS_WDOG32"
	<a href="#">Section "FS_WDOG_Check()"</a> RegWide8b = "FS_WDOG_SRS_WIDE_32b"
iec60730b_invariable_memory.c	<a href="#">Section "FS_FLASH_C_HW16_K()"</a>
iec60730b_cm4_cm7_flash.S	<a href="#">Section "FS_CM4_CM7_FLASH_HW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW32()"</a>
iec60730b_cm4_cm7_pc.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_ram.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_reg.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_stack.S	<a href="#">Common for all CM4/CM7 devices</a>

### 1.3.8 MKV3x dedicated functions

For MKV3x family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1
- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 9](#) shows the list of functions dedicated for the MKV3x device.

**Table 9. MKV3x dedicated functions**

File	Suitable function
iec60730b_aio.c	<a href="#">Section "FS_AIO_LimitCheck()"</a>
	<a href="#">Section "FS_AIO_InputSet_A23()"</a>
	<a href="#">Section "FS_AIO_ReadResult_A23()"</a>
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_LPTMR()"</a>
iec60730b_dio.c	<a href="#">Section "FS_DIO_Output()"</a>
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt()"</a>
	<a href="#">Section "FS_DIO_ShortToSupplySet()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet()"</a>
iec60730b_wdog.c	<a href="#">Section "FS_WDOG_Setup_LPTMR()"</a> refresh_index = "FS_KINETIS_WDOG"
	<a href="#">Section "FS_WDOG_Check()"</a> RegWide8b = "FS_WDOG_SRS_WIDE_8b"
iec60730b_invariable_memory.c	<a href="#">Section "FS_FLASH_C_HW16_K()"</a>
iec60730b_cm4_cm7_flash.S	<a href="#">Section "FS_CM4_CM7_FLASH_HW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW32()"</a>
iec60730b_cm4_cm7_pc.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_ram.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_reg.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_stack.S	<a href="#">Common for all CM4/CM7 devices</a>

### 1.3.9 MKV4x dedicated functions

For MKV4x family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1
- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 10](#) shows the list of functions dedicated for the MKV4x device.

**Table 10. MKV4x dedicated functions**

File	Suitable function
iec60730b_aio.c	<a href="#">Section "FS_AIO_LimitCheck()"</a>
	<a href="#">Section "FS_AIO_InputSet_A7()"</a>

Table 10. MKV4x dedicated functions...continued

File	Suitable function
	<a href="#">Section "FS_AIO_ReadResult_A7()"</a>
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_LPTMR()"</a>
iec60730b_dio.c	<a href="#">Section "FS_DIO_Output()"</a>
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt()"</a>
	<a href="#">Section "FS_DIO_ShortToSupplySet()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet()"</a>
iec60730b_wdog.c	<a href="#">Section "FS_WDOG_Setup_LPTMR()"</a> refresh_index = "FS_KINETIS_WDOG"
	<a href="#">Section "FS_WDOG_Check()"</a> RegWide8b = "FS_WDOG_SRS_WIDE_8b"
iec60730b_invariable_memory.c	<a href="#">Section "FS_FLASH_C_HW16_K()"</a>
iec60730b_cm4_cm7_flash.S	<a href="#">Section "FS_CM4_CM7_FLASH_HW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW32()"</a>
iec60730b_cm4_cm7_pc.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_ram.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_reg.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_stack.S	<a href="#">Common for all CM4/CM7 devices</a>

### 1.3.10 MKV5x dedicated functions

For MKV5x family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1
- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 11](#) shows the list of functions dedicated for the MKV5x device.

Table 11. MKV5x dedicated functions

File	Suitable function
iec60730b_aio.c	<a href="#">Section "FS_AIO_LimitCheck()"</a>
	<a href="#">Section "FS_AIO_InputSet_A23()"</a>
	<a href="#">Section "FS_AIO_ReadResult_A23()"</a>
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_LPTMR()"</a>
iec60730b_dio.c	<a href="#">Section "FS_DIO_Output()"</a>
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt()"</a>

Table 11. MKV5x dedicated functions...continued

File	Suitable function
	<a href="#">Section "FS_DIO_ShortToSupplySet()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet()"</a>
iec60730b_wdog.c	<a href="#">Section "FS_WDOG_Setup_LPTMR()"</a> refresh_index = "FS_KINETIS_WDOG"
	<a href="#">Section "FS_WDOG_Check()"</a> RegWide8b = "FS_WDOG_SRS_WIDE_8b"
iec60730b_invariable_memory.c	<a href="#">Section "FS_FLASH_C_HW16_K()"</a>
iec60730b_cm4_cm7_flash.S	<a href="#">Section "FS_CM4_CM7_FLASH_HW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW32()"</a>
iec60730b_cm4_cm7_pc.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_ram.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_reg.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_stack.S	<a href="#">Common for all CM4/CM7 devices</a>

### 1.3.11 LPC54S0x/LPC540x dedicated functions

For LPC54S0x/LPC540x family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1
- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 12](#) shows the list of functions dedicated for the LPC54S0x/LPC540x devices.

Table 12. LPC54S0x/LPC540x dedicated functions

File	Suitable function
iec60730b_aio.c	<a href="#">Section "FS_AIO_LimitCheck()"</a>
	<a href="#">Section "FS_AIO_InputSet_A5()"</a>
	<a href="#">Section "FS_AIO_ReadResult_A5()"</a>
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_CTIMER()"</a>
iec60730b_dio.c	<a href="#">Section "FS_DIO_Output_LPC()"</a>
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt_LPC()"</a>
	<a href="#">Section "FS_DIO_ShortToSupplySet_LPC()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet_LPC()"</a>
iec60730b_wdog.c	<a href="#">Section "FS_WDOG_Setup_WWDT_CTIMER()"</a>
	<a href="#">Section "FS_WDOG_Check_WWDT_LPC()"</a>
iec60730b_cm4_cm7_flash.S	<a href="#">Section "FS_CM4_CM7_FLASH_SW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW32()"</a>

**Table 12. LPC54S0x/LPC540x dedicated functions...continued**

File	Suitable function
iec60730b_invariable_memory.c	<a href="#">Section "FS_FLASH_C_HW16_L()"</a>
iec60730b_cm4_cm7_pc.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_ram.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_reg.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_stack.S	<a href="#">Common for all CM4/CM7 devices</a>

**1.3.12 MK32L3 CM4 dedicated functions**

For MK32L3 CM4 family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1
- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 13](#) shows the list of functions dedicated for the MK32L3 CM4 core.

**Table 13. MK32L3 dedicated functions for CM4 core**

File	Suitable function
iec60730b_aio.c	<a href="#">Section "FS_AIO_LimitCheck()"</a>
	<a href="#">Section "FS_AIO_InputSet_A1()"</a>
	<a href="#">Section "FS_AIO_ReadResult_A1()"</a>
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_LPTMR()"</a>
iec60730b_dio.c	<a href="#">Section "FS_DIO_Output()"</a>
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt()"</a>
	<a href="#">Section "FS_DIO_ShortToSupplySet()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet()"</a>
iec60730b_wdog.c	<a href="#">Section "FS_WDOG_Setup_LPTMR()"</a> refresh_index = "FS_KINETIS_WDOG"
	<a href="#">Section "FS_WDOG_Check()"</a> RegWide8b = "FS_WDOG_SRS_WIDE_32b"
iec60730b_cm4_cm7_flash.S	<a href="#">Functions are described in dedicated chapter</a>
iec60730b_cm4_cm7_pc.S	<a href="#">Functions are common for all CM4 CM7 devices</a>
iec60730b_cm4_cm7_ram.S	<a href="#">Functions are common for all CM4 CM7 devices</a>
iec60730b_cm4_cm7_reg.S	<a href="#">Functions are common for all CM4 CM7 devices</a>
iec60730b_cm4_cm7_Stack.S	<a href="#">Functions are common for all CM4 CM7 devices</a>

**1.3.13 MCXE31x dedicated functions**

For MCXE31x family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1

- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 14](#) shows the list of functions dedicated for the MCXE31x device.

**Table 14. MCXE31x dedicated functions**

File	Suitable function
iec60730b_aio.c	This test is not supported in the current release.
iec60730b_wdog.c	This test is not supported in the current release.
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_STM()"</a>
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt_SIUL2()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet_SIUL2()"</a>
	<a href="#">Section "FS_DIO_ShortToSupplySet_SIUL2()"</a>
iec60730b_invariable_memory.c	<a href="#">Section "FS_FLASH_C_HW32_K()"</a>
iec60730b_cm4_cm7_flash.S	<a href="#">Section "FS_CM4_CM7_FLASH_SW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW32()"</a>
iec60730b_cm4_cm7_pc.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_ram.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_reg.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_stack.S	<a href="#">Common for all CM4/CM7 devices</a>

### 1.3.14 MCXE24x dedicated functions

For MCXE24x family must be *device\_information.h* configured as follow:

- #define FPU\_SUPPORT 1
- #define DSP\_SUPPORT 0
- #define TZ\_SUPPORT 0

[Table 15](#) shows the list of functions dedicated for the MCXE24x device.

**Table 15. MCXE24x dedicated functions**

File	Suitable function
iec60730b_aio.c	<a href="#">Section "FS_AIO_LimitCheck()"</a>
	<a href="#">Section "FS_AIO_InputSet_A8()"</a>
	<a href="#">Section "FS_AIO_ReadResult_A8()"</a>
iec60730b_clock.c	<a href="#">Section "FS_CLK_Check()"</a>
	<a href="#">Section "FS_CLK_Init()"</a>
	<a href="#">Section "FS_CLK_LPTMR()"</a>
iec60730b_dio.c	<a href="#">Section "FS_DIO_Output()"</a>
iec60730b_dio_ext.c	<a href="#">Section "FS_DIO_InputExt()"</a>
	<a href="#">Section "FS_DIO_ShortToAdjSet()"</a>

Table 15. MCXE24x dedicated functions...continued

File	Suitable function
	<a href="#">Section "FS_DIO_ShortToSupplySet()"</a>
iec60730b_wdog.c	This test is not supported in the current release.
iec60730b_cm4_cm7_flash.S	<a href="#">Section "FS_CM4_CM7_FLASH_HW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW16()"</a>
	<a href="#">Section "FS_CM4_CM7_FLASH_SW32()"</a>
iec60730b_cm4_cm7_pc.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_ram.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_reg.S	<a href="#">Common for all CM4/CM7 devices</a>
iec60730b_cm4_cm7_stack.S	<a href="#">Common for all CM4/CM7 devices</a>

### 1.4 Functions performance measurement

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

1. The function parameter was measured on MKV31 with a clock frequency of 80 MHz.
2. The function parameter was measured on MKV46 with a clock frequency of 80 MHz.
3. The function parameter was measured on MKE18F with a clock frequency of 100 MHz.
4. The function parameter was measured on MIMXRT1050 with a clock frequency of 600 MHz.
5. The function parameter was measured on MIMX8MN with a clock frequency of 600 MHz.
6. The function parameter was measured on MIMXRT1170 with a clock frequency of 996 MHz.
7. The function parameter was measured on LPC54S018M with a clock frequency of 96 MHz.
8. The function parameter was measured on MCXE31x with a clock frequency of 160 MHz.
9. The function parameter was measured on MCXE247 with a clock frequency of 48 MHz.

## 2 Analog Input/Output (IO) test

The analog IO test procedure performs the plausibility check of the analog 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 must check the analog input interface with 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:

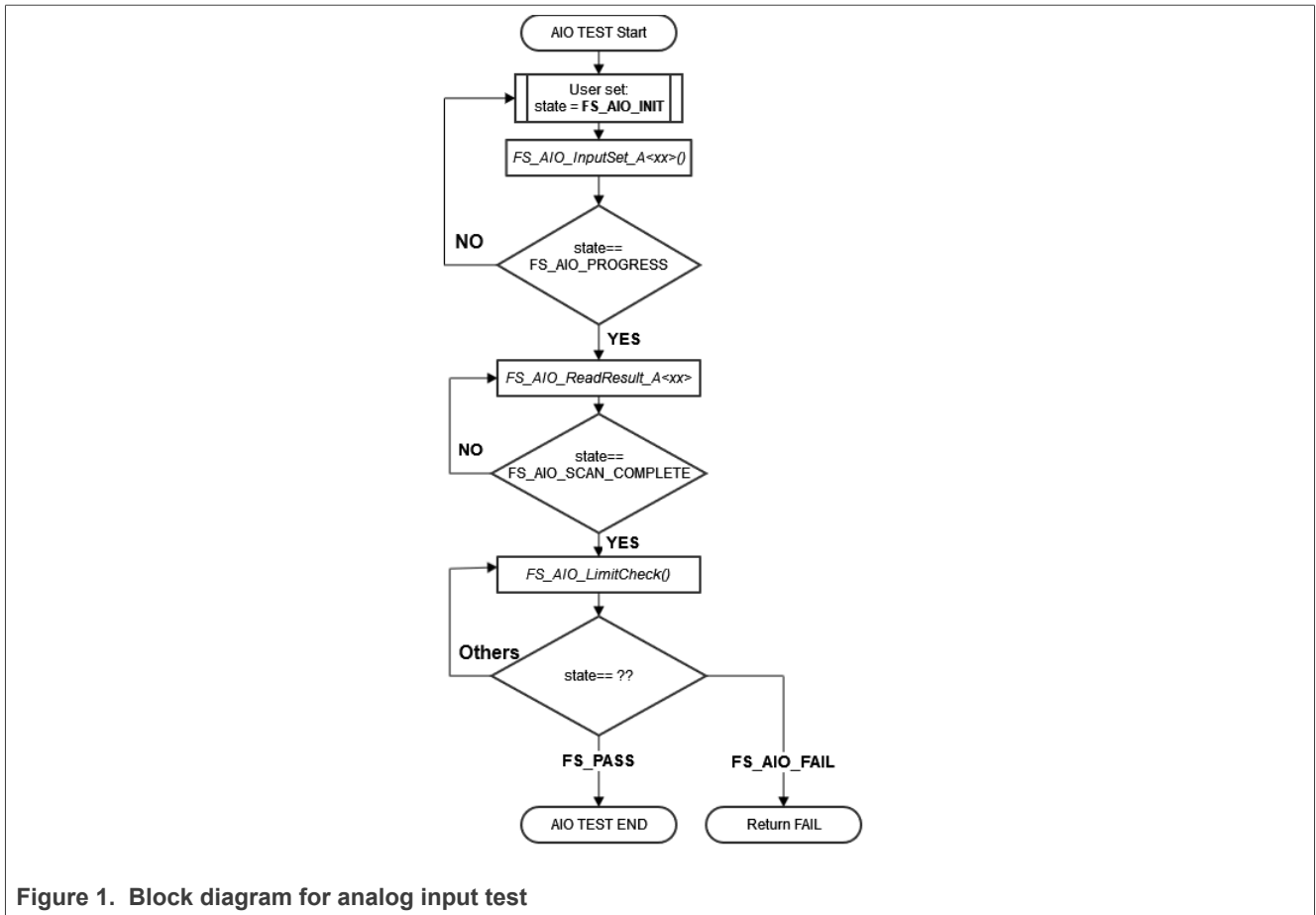


Figure 1. Block diagram for analog input test

The figure above shows the sequence of conversion and checks one channel. For the full ADC test, run this sequence with three channels: reference high, reference low, and bandgap voltage. This sequence is handled on the user application side, all functions from the library (with the **FS\_** prefix) are written as non-blocking.

### 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 16. 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.

All functions are written as non-blocking, each function checks if the state variable is set to the necessary state. If not, they return immediately.

Throughout all supported devices, the ADC module has a slightly different arrangement of the registers that are involved in the test. Therefore, a standalone function is created for each ADC module. See [Core self-test library – source code version](#) for the function dedicated for your device. Also the corresponding data type must be used with this selected function.

The analog input test is based on a conversion of three analog inputs with known voltage values and it checks if the converted values fit into the specified limits. Normally, the limits should be roughly 10 % around the desired reference values.

For easier implementation of the AIO test to the final application, the AIO test is divided to three independent cycles:

1. Conversion and check of low reference
2. Conversion and check of high reference
3. Conversion and check of bandgap reference (the middle range of voltage)

Each of this independent phase has its own "test instance" structure with the *fs\_aio\_test\_a<TYPE>\_t* data type. The defined types which cover all supported devices are in the *iec60730b\_aio.h* file. The selected type must correspond to the used device. The description of each type is in the corresponding function description below.

**The following functions are used to test the analog input:**

- *FS\_AIO\_InputSet\_A1, FS\_AIO\_InputSet\_A23, FS\_AIO\_InputSet\_A4, FS\_AIO\_InputSet\_A4\_2, FS\_AIO\_InputSet\_A5, FS\_AIO\_InputSet\_A6, FS\_AIO\_InputSet\_A7, FS\_AIO\_InputSet\_A8*
- *FS\_AIO\_ReadResult\_A1, FS\_AIO\_ReadResult\_A23, FS\_AIO\_ReadResult\_A4, FS\_AIO\_ReadResult\_A5, FS\_AIO\_ReadResult\_A6, FS\_AIO\_ReadResult\_A7, FS\_AIO\_ReadResult\_A8*
- *FS\_AIO\_LimitCheck*

The *FS\_AIO\_InputSet\_A<xx>* and *FS\_AIO\_ReadResult\_A<xx>* functions are related directly to the used ADC module.

The *FS\_AIO\_LimitCheck* function works only with the AIO test instance structure and are not related to the ADC HW.

Each test instance structure has a "state" variable. This variable controls the code flow. You can use only a part of the ADC check functions. For example, it is possible to use only "*FS\_AIO\_LimitCheck()*" and the HW part of the test must be done on the application side. In this case, it is necessary to ensure that the state flow is correctly handled. Before calling *FS\_AIO\_LimitCheck()* set the state to "FS\_AIO\_SCAN\_COMPLETE" and fill the "RawResult" variable.

The whole state flow is as follows:

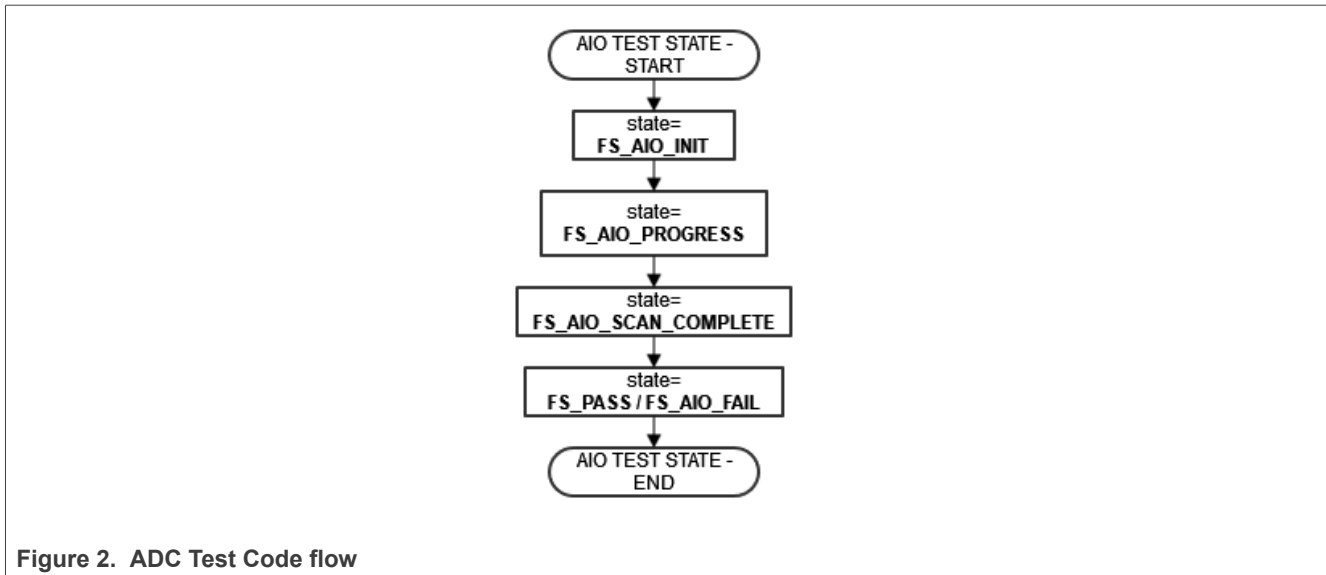


Figure 2. ADC Test Code flow

At the start, set the state variable to "FS\_AIO\_INIT". Test the items with this state. It can be used in function "FS\_AIO\_InputSet\_A<xx>", which sets the correct channel and trigger conversion of the ADC. After this function, set the variable to "FS\_AIO\_PROGRESS". In the progress state, call the "FS\_AIO\_ReadResult\_A<xx>" function, which, in case that the conversion is complete, stores the conversion to the RawResult variable in the test items structure and sets the state to "FS\_AIO\_SCAN\_COMPLETE". After this, call the FS\_AIO\_LimitCheck() function to check if RawResult is within Limits. This function sets the state variable to FS\_PASS or FS\_FAIL.

**Initialization of the test**

In some \*.c files, you must define a corresponding array variable:

**Testing the instance variables definition**

```

/*****
 *
 *
 *          STRUCTURE FOR AIO TEST
 *
 *
 *****/
#define TESTED_ADC ADC0
#define ADC_RESOLUTION 12
#define ADC_REFERENCE 3.06
#define ADC_BANDGAP_LEVEL 1.7
#define ADC_DEVIATION_PERCENT 10
#define ADC_MAX ((1 << (ADC_RESOLUTION)) - 1)
#define ADC_BANDGAP_LEVEL_RAW (((ADC_BANDGAP_LEVEL) * (ADC_MAX)) / (ADC_REFERENCE))
#define ADC_MIN_LIMIT(val) (uint16_t)((val) * (100 - ADC_DEVIATION_PERCENT)) / 100)
#define ADC_MAX_LIMIT(val) (uint16_t)((val) * (100 + ADC_DEVIATION_PERCENT)) / 100)
fs_aio_test_a23468_t aio_safety_test_item_VL =
{
    .AdcChannel = 30,
    .Limits.low = (uint32_t)ADC_MIN_LIMIT(0),
    .Limits.high = (uint32_t)ADC_MAX_LIMIT(60),
}
  
```

```

    .state = FS_AIO_INIT
};
fs_aio_test_a23468_t aio_safety_test_item_VH =
{
    .AdcChannel = 29,
    .Limits.low = (uint32_t)ADC_MIN_LIMIT(ADC_MAX),
    .Limits.high = (uint32_t)ADC_MAX_LIMIT(ADC_MAX),
    .state = FS_AIO_INIT
};
fs_aio_test_a23468_t aio_safety_test_item_BG =
{
    .AdcChannel = 27,
    .Limits.low = (uint32_t)ADC_MIN_LIMIT(ADC_BANDGAP_LEVEL_RAW),
    .Limits.high = (uint32_t)ADC_MAX_LIMIT(ADC_BANDGAP_LEVEL_RAW),
    .state = FS_AIO_INIT
};
/* NULL terminated array of pointers to fs_aio_test_a23468_t items for safety
AIO test */
fs_aio_test_a23468_t *g_aio_safety_test_items[] = {&aio_safety_test_item_VL,
                                                    &aio_safety_test_item_VH,
                                                    &aio_safety_test_item_BG,
                                                    NULL};

```

After the definition, all necessary variables and initialization of ADC HW can be called as a function for the AIO test:

### Test

```

for(uint8_t i=0;i<3;i++) /* 3 test items VL, VH and BG */
{
    static int index = 0; /* Iteration variable for going through all ADC test
items */
    psSafetyCommon->AIO_test_result =
    FS_AIO_LimitCheck(g_aio_safety_test_items[index]->RawResult,
    &(g_aio_safety_test_items[index]->Limits), &(g_aio_safety_test_items[index]-
>state));
    switch (psSafetyCommon->AIO_test_result)
    {
        case FS_AIO_INIT:
            FS_AIO_InputSet_A23(g_aio_safety_test_items[index], (fs_aio_a23_t
*)TESTED_ADC);
            break;
        case FS_AIO_PROGRESS:
            FS_AIO_ReadResult_A23(g_aio_safety_test_items[index], (fs_aio_a23_t
*)TESTED_ADC);
            break;
        case FS_PASS: /* successful execution of test, call the trigger function again
*/
            if( g_aio_safety_test_items[++index] == NULL)
            {
                index = 0; /* again first channel*/
            }
            g_aio_safety_test_items[index]->state = FS_AIO_INIT;
            break;
        default:
            __asm("NOP");
            break;
    }
    /* Necessary delay for conversion time */
    for (uint8_t y = 0; y < 20; y++){ __asm("nop");}
}

```

```
}

```

## 2.2.1 ADC type A1

The ADC type of the A1 covers at least the following device families: K32L3A6, LPC55xx, i.MX RT117x, and i.MX RT116x.

For this group of devices, the following functions are dedicated:

- *FS\_AIO\_InputSet\_A1*
- *FS\_AIO\_ReadResult\_A1*
- *FS\_AIO\_LimitCheck*

For this type of ADCs, it is necessary use these data types:

- *fs\_aio\_test\_a1\_t* - for the test instance
- *fs\_aio\_a1\_t* - for a pointer to the ADC peripheral

### 2.2.1.1 fs\_aio\_a1\_t

*fs\_aio\_a1\_t* is data type for accessing ADC module registers. This data type is defined in the *iec60730b\_types.h* file and supports the device families mentioned above.

### 2.2.1.2 fs\_aio\_test\_a1\_t

This structure is the base structure of the ADC test. This data type is defined in the *iec60730b\_aio.h* file.

Define this structure and fill it to use the ADC test.

```
typedef struct
{
    uint8_t   AdcChannel;
    uint16_t  commandBuffer;
    uint8_t   SideSelect; /* 0 = A side, 1 = B side*/
    uint8_t   softwareTriggerEvent;
    fs_aio_limits_t Limits;
    uint32_t  RawResult;
    FS_RESULT state;
} fs_aio_test_a1_t;
```

- *AdcChannel* - the number of the ADC channel
- *commandBuffer* - the index of CommandBuffer
- *SideSelect* - 0 A side, 1 B side
- *softwareTriggerEvent* - the index of the software trigger
- *Limits* - a structure with low and high limits for *AdcChannel*
- *RawResult* - a raw result of the ADC conversion of "*AdcChannel*"
- *state* - a state variable, it can have the value of a macro: *FS\_PASS*, *FS\_FAIL\_AIO*, *FS\_AIO\_INIT*, *FS\_AIO\_PROGRESS*, *FS\_AIO\_SCAN\_COMPLETE*

### 2.2.1.3 FS\_AIO\_InputSet\_A1()

This function executes the first part of the AIO test sequence. This function sets up the ADC input channel and 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. The function has effect only when the input state is "*FS\_AIO\_INIT*". It has no effect in other states.

**Function prototype:**

```
FS_RESULT FS_AIO_InputSet_A1(fs_aio_test_a1_t *pObj, fs_aio_a1_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:**

For information about the function performance, see [Core self-test library – source code version](#).

**2.2.1.4 FS\_AIO\_ReadResult\_A1()**

This function is tied to the ADC hardware. This function reads the converted analog value only if pObj->state == *FS\_AIO\_PROGRESS*. When the value is read, it is stored to "pObj->RawResult" and the "pObj->State" variable is set to "*FS\_AIO\_SCAN\_COMPLETE*". The function uses a non-blocking approach.

**Function prototype:**

```
FS_RESULT FS_AIO_ReadResult_A1(fs_aio_test_a1_t *pObj, fs_aio_a1_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\_SCAN\_COMPLETE* - The conversion value was successfully read and stored to the "RawResult" variable.

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](#).

**2.2.2 ADC type A23**

The ADC type A23 covers at least the following device families: KV1x, KV3x, KLxx, K32L2A, K32L2B, K22F, KW3x, and KE0x.

For this group of devices, the following functions are dedicated:

- *FS\_AIO\_InputSet\_A23*
- *FS\_AIO\_ReadResult\_A23*
- *FS\_AIO\_LimitCheck*

For this type of ADCs, it is necessary to use these data types:

- *fs\_aio\_test\_a23468\_t* - for the test instance

- `fs_aio_a23_t` - for a pointer to the ADC peripheral

### 2.2.2.1 `fs_aio_a23_t`

The "`fs_aio_a23_t`" data type serves for accessing ADC module registers. This data type is defined in the `iec60730b_types.h` file and it supports the device families mentioned above.

### 2.2.2.2 `fs_aio_test_a23468_t`

This structure is the base structure of the ADC test. This data type is defined in the `iec60730b_aio.h` file.

Define this structure and fill it to use the ADC test.

```
typedef struct
{
    uint8_t AdcChannel;
    fs_aio_limits_t Limits;
    uint32_t RawResult;
    FS_RESULT state;
} fs_aio_test_a23468_t;
```

- `AdcChannel` - the number of the ADC channel
- `Limits` - a structure with low and high limits for `AdcChannel`
- `RawResult` - a raw result of the ADC conversion of `AdcChannel`
- `state` - a state variable, it can have the value of a macro: `FS_PASS`, `FS_FAIL_AIO`, `FS_AIO_INIT`, `FS_AIO_PROGRESS`, `FS_AIO_SCAN_COMPLETE`

### 2.2.2.3 `FS_AIO_InputSet_A23()`

This function executes the first part of the AIO test sequence. This function sets up the ADC input channel and 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. The function has effect only when the input state is "`FS_AIO_INIT`". It has no effect in other states.

#### **Function prototype:**

```
FS_RESULT FS_AIO_InputSet_A23(fs_aio_test_a23468_t *pObj, fs_aio_a23_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:**

For information about the function performance, see [Core self-test library – source code version](#).

### 2.2.2.4 FS\_AIO\_ReadResult\_A23()

This function is tied to the ADC hardware. This function reads the converted analog value only if "pObj->state == FS\_AIO\_PROGRESS". When the value is read, it is stored to "pObj->RawResult" and the "pObj->State" variable is set to "FS\_AIO\_SCAN\_COMPLETE".

#### Function prototype:

```
FS_RESULT FS_AIO_ReadResult_A23(fs_aio_test_a23468_t *pObj, fs_aio_a23_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\_SCAN\_COMPLETE - The conversion value was successfully read and stored to the "RawResult" variable.

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](#).

## 2.2.3 ADC type A4

The ADC type A4 covers at least the following device families: KE1xZ and KE1xF.

For this group of devices, the following functions are dedicated:

- FS\_AIO\_InputSet\_A4
- FS\_AIO\_InputSet\_A4\_2 - dedicated function for MKE17Z9
- FS\_AIO\_ReadResult\_A4
- FS\_AIO\_LimitCheck

For this type of ADCs, it is necessary to use these data types:

- fs\_aio\_test\_a23468\_t - for the test instance
- fs\_aio\_a4\_t - for a pointer to the ADC peripheral

### 2.2.3.1 fs\_aio\_a4\_t

The "fs\_aio\_a4\_t" data type serves for accessing ADC module registers. This data type is defined in the *iec60730b\_types.h* file and it supports the device families mentioned above.

### 2.2.3.2 fs\_aio\_test\_a23468\_t

This structure is the base structure of the ADC test. This data type is defined in the *iec60730b\_aio.h* file.

Define this structure and fill it to use the ADC test.

```
typedef struct
{
    uint8_t AdcChannel;
    fs_aio_limits_t Limits;
    uint32_t RawResult;
    FS_RESULT state;
}
```

```
} fs_aio_test_a23468_t;
```

- AdcChannel - the number of the ADC channel
- Limits - a structure with low and high limits for AdcChannel
- RawResult - a raw result of the ADC conversion of AdcChannel
- state - a state variable, it can have the value of a macro: *FS\_PASS*, *FS\_FAIL\_AIO*, *FS\_AIO\_INIT*, *FS\_AIO\_PROGRESS*, *FS\_AIO\_SCAN\_COMPLETE*

### 2.2.3.3 FS\_AIO\_InputSet\_A4()

This function executes the first part of the AIO test sequence. This function sets up the ADC input channel and 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. The function has effect only when the input state is "FS\_AIO\_INIT". It has no effect in other states.

#### Function prototype:

```
FS_RESULT FS_AIO_InputSet_A4(fs_aio_test_a23468_t *pObj, fs_aio_a4_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:

For information about the function performance, see [Core self-test library – source code version](#).

#### Calling restriction :

For MKE17Z9, use the [Section "FS\\_AIO\\_InputSet\\_A4\\_2\(\)"](#) function.

### 2.2.3.4 FS\_AIO\_InputSet\_A4\_2()

This function executes the first part of the AIO test sequence. It is dedicated for MKE17Z9. This function sets up the ADC input channel and 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. The function has effect only when the input state is "FS\_AIO\_INIT". It has no effect in other states.

#### Function prototype:

```
FS_RESULT FS_AIO_InputSet_A4_2(fs_aio_test_a23468_t *pObj, fs_aio_a4_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:**

For information about the function performance, see [Core self-test library – source code version](#).

**Calling restriction :**

The function is dedicated only for MKE17Z9.

### 2.2.3.5 FS\_AIO\_ReadResult\_A4()

This function is tied to the ADC hardware. This function reads the converted analog value only if "pObj->state == FS\_AIO\_PROGRESS". When the value is read, it is stored to "pObj->RawResult" and the "pObj->State" variable is set to "FS\_AIO\_SCAN\_COMPLETE".

**Function prototype:**

```
FS_RESULT FS_AIO_ReadResult_A4(fs_aio_test_a23468_t *pObj, fs_aio_a4_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\_SCAN\_COMPLETE* - The conversion value was successfully read and stored to the "RawResult" variable.

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](#).

## 2.2.4 ADC type A6

The ADC type A6 covers at least the following device family: i.MXRT10xx.

For this group of devices, the following functions are dedicated:

- *FS\_AIO\_InputSet\_A6*
- *FS\_AIO\_ReadResult\_A6*
- *FS\_AIO\_LimitCheck*

For this type of ADCs, it is necessary to use these data types:

- *fs\_aio\_test\_a23468\_t* - for test instance
- *fs\_aio\_a6\_t* - for a pointer to the ADC peripheral

### 2.2.4.1 fs\_aio\_a6\_t

The "fs\_aio\_a6\_t" data type is used for accessing ADC module registers. This data type is defined in the *iec60730b\_types.h* file and it supports the device families mentioned above.

### 2.2.4.2 fs\_aio\_test\_a23468\_t

This structure is the base structure of the ADC test. This data type is defined in the *iec60730b\_aio.h* file.

Define this structure and fill it to use the ADC test.

```
typedef struct
{
    uint8_t AdcChannel;
    fs_aio_limits_t Limits;
    uint32_t RawResult;
    FS_RESULT state;
} fs_aio_test_a23468_t;
```

- AdcChannel - the number of the ADC channel
- Limits - a structure with low and high limits for AdcChannel
- RawResult - a raw result of the ADC conversion of AdcChannel
- state - a state variable, it can have the value of a macro: *FS\_PASS*, *FS\_FAIL\_AIO*, *FS\_AIO\_INIT*, *FS\_AIO\_PROGRESS*, *FS\_AIO\_SCAN\_COMPLETE*

### 2.2.4.3 FS\_AIO\_InputSet\_A6()

This function executes the first part of the AIO test sequence. This function sets up the ADC input channel and 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. The function has effect only when the input state is "*FS\_AIO\_INIT*". It has no effect in other states.

#### **Function prototype:**

```
FS_RESULT FS_AIO_InputSet_A6(fs_aio_test_a23468_t *pObj, fs_aio_a6_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:**

For information about the function performance, see [Core self-test library – source code version](#).

### 2.2.4.4 FS\_AIO\_ReadResult\_A6()

This function is tied to the ADC hardware. This function reads the converted analog value only if "pObj->state == *FS\_AIO\_PROGRESS*". When the value is read, it is stored to pObj->RawResult and the pObj->State variable is set to "*FS\_AIO\_SCAN\_COMPLETE*".

#### **Function prototype:**

```
FS_RESULT FS_AIO_ReadResult_A6(fs_aio_test_a23468_t* pObj, fs_aio_a6_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\_SCAN\_COMPLETE* - The conversion value was successfully read and stored to the "RawResult" variable.

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](#).

### 2.2.5 ADC type A5

The ADC type A5 covers at least the following device families: LPC51U68, LPC8xx, LPC540x, and LPC54S0x.

For this group of devices, the following functions are dedicated:

- *FS\_AIO\_InputSet\_A5*
- *FS\_AIO\_ReadResult\_A5*
- *FS\_AIO\_LimitCheck*

For this type of ADCs, it is necessary to use these data types:

- *fs\_aio\_test\_a5\_t* - for the test instance
- *fs\_aio\_a5\_t* - for a pointer to the ADC peripheral

#### 2.2.5.1 *fs\_aio\_a5\_t*

The "*fs\_aio\_a5\_t*" data type serves for accessing ADC module registers. This data type is defined in the *iec60730b\_types.h* file and it supports the device families mentioned above.

#### 2.2.5.2 *fs\_aio\_test\_a5\_t*

This structure is the base structure of the ADC test. This data type is defined in the *iec60730b\_aio.h* file.

Define this structure and fill it to use the ADC test.

```
typedef struct
{
    uint8_t AdcChannel;
    uint8_t sequence;
    fs_aio_limits_t Limits;
    uint32_t RawResult;
    FS_RESULT state;
} fs_aio_test_a5_t;
```

- *AdcChannel* - the number of the ADC channel
- *sequence* - the index of the used sequence
- *Limits* - a structure with low and high limits for *AdcChannel*
- *RawResult* - a raw result of the ADC conversion of *AdcChannel*
- *state* - a state variable, it can have the value of a macro: *FS\_PASS*, *FS\_FAIL\_AIO*, *FS\_AIO\_INIT*, *FS\_AIO\_PROGRESS*, *FS\_AIO\_SCAN\_COMPLETE*

#### 2.2.5.3 *FS\_AIO\_InputSet\_A5()*

This function executes the first part of the AIO test sequence. This function sets up the ADC input channel and 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. The function has effect only when the input state is "FS\_AIO\_INIT". It has no effect in other states.

**Function prototype:**

```
FS_RESULT FS_AIO_InputSet_A5(fs_aio_test_a5_t *pObj, fs_aio_a5_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:**

For information about the function performance, see [Core self-test library – source code version](#).

#### 2.2.5.4 FS\_AIO\_ReadResult\_A5()

This function is tied to the ADC hardware. This function reads the converted analog value only if "pObj->state == FS\_AIO\_PROGRESS". When the value is read, it is stored to "pObj->RawResult" and the "pObj->State" variable is set to "FS\_AIO\_SCAN\_COMPLETE".

**Function prototype:**

```
FS_RESULT FS_AIO_ReadResult_A5(fs_aio_test_a5_t* pObj, fs_aio_a5_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\_SCAN\_COMPLETE - The conversion value was successfully read and stored to the "RawResult" variable.

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](#).

#### 2.2.6 ADC type A7

The ADC type A7 covers at least the following device family: KV4x.

For this group of devices, the following functions are dedicated:

- FS\_AIO\_InputSet\_A7
- FS\_AIO\_ReadResult\_A7
- FS\_AIO\_LimitCheck

For this type of ADCs, it is necessary to use these data types:

- `fs_aio_test_a7_t` - for the test instance
- `fs_aio_a7_t` - for a pointer to the ADC peripheral

### 2.2.6.1 `fs_aio_a7_t`

The "`fs_aio_a7_t`" data type is used for accessing ADC module registers. This data type is defined in the `iec60730b_types.h` file and it supports the device families mentioned above.

### 2.2.6.2 `fs_aio_test_a7_t`

This structure is the base structure of the ADC test. This data type is defined in the `iec60730b_aio.h` file.

Define this structure and fill it to use the ADC test.

```
typedef struct
{
    uint8_t AdcChannel;
    uint8_t Sample;
    fs_aio_limits_t Limits;
    uint32_t RawResult;
    FS_RESULT state;
} fs_aio_test_a7_t;
```

- `AdcChannel` - the number of the ADC channel
- `Sample` - the number of the sample register
- `Limits` - a structure with low and high limits for `AdcChannel`
- `RawResult` - a raw result of the ADC conversion of `AdcChannel`
- `state` - a state variable, it can have the value of a macro: `FS_PASS`, `FS_FAIL_AIO`, `FS_AIO_INIT`, `FS_AIO_PROGRESS`, `FS_AIO_SCAN_COMPLETE`

### 2.2.6.3 `FS_AIO_InputSet_A7()`

This function executes the first part of the AIO test sequence. This function sets up the ADC input channel and 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. The function has effect only when the input state is "`FS_AIO_INIT`". It has no effect in other states.

#### **Function prototype:**

```
FS_RESULT FS_AIO_InputSet_A7(fs_aio_test_a7_t *pObj, fs_aio_a7_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:**

For information about the function performance, see [Core self-test library – source code version](#).

### 2.2.6.4 FS\_AIO\_ReadResult\_A7()

This function is tied to the ADC hardware. This function reads the converted analog value only if "pObj->state == FS\_AIO\_PROGRESS". When the value is read, it is stored to "pObj->RawResult" and the "pObj->State" variable is set to "FS\_AIO\_SCAN\_COMPLETE".

#### Function prototype:

```
FS_RESULT FS_AIO_ReadResult_A7(fs_aio_test_a7_t *pObj, fs_aio_a7_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\_SCAN\_COMPLETE* - The conversion value was successfully read and stored to the "RawResult" variable.

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](#).

## 2.2.7 ADC type A8

The ADC type A8 covers at least the following device family: MCXE24x.

For this group of devices, the following functions are dedicated:

- *FS\_AIO\_InputSet\_A8*
- *FS\_AIO\_ReadResult\_A8*
- *FS\_AIO\_LimitCheck*

For this type of ADCs, it is necessary to use these data types:

- *fs\_aio\_test\_a23468\_t* - for test instance
- *fs\_aio\_a8\_t* - for a pointer to the ADC peripheral

### 2.2.7.1 fs\_aio\_a8\_t

The "fs\_aio\_a8\_t" data type is used for accessing ADC module registers. This data type is defined in the *iec60730b\_types.h* file and it supports the device families mentioned above.

### 2.2.7.2 fs\_aio\_test\_a23468\_t

This structure is the base structure of the ADC test. This data type is defined in the *iec60730b\_aio.h* file.

Define this structure and fill it to use the ADC test.

```
typedef struct
{
    uint8_t AdcChannel;
    fs_aio_limits_t Limits;
    uint32_t RawResult;
    FS_RESULT state;
}
```

```
} fs_aio_test_a23468_t;
```

- `AdcChannel` - the number of the ADC channel
- `Limits` - a structure with low and high limits for `AdcChannel`
- `RawResult` - a raw result of the ADC conversion of `AdcChannel`
- `state` - a state variable, it can have the value of a macro: `FS_PASS`, `FS_FAIL_AIO`, `FS_AIO_INIT`, `FS_AIO_PROGRESS`, `FS_AIO_SCAN_COMPLETE`

### 2.2.7.3 FS\_AIO\_InputSet\_A8()

This function executes the first part of the AIO test sequence. This function sets up the ADC input channel and 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. The function has effect only when the input state is "`FS_AIO_INIT`". It has no effect in other states.

#### Function prototype:

```
FS_RESULT FS_AIO_InputSet_A8(fs_aio_test_a23468_t *pObj, fs_aio_a8_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:

For information about the function performance, see [Core self-test library – source code version](#).

### 2.2.7.4 FS\_AIO\_ReadResult\_A8()

This function is tied to the ADC hardware. This function reads the converted analog value only if "`pObj->state == FS_AIO_PROGRESS`". When the value is read, it is stored to `pObj->RawResult` and the `pObj->State` variable is set to "`FS_AIO_SCAN_COMPLETE`".

#### Function prototype:

```
FS_RESULT FS_AIO_ReadResult_A8(fs_aio_test_a23468_t* pObj, fs_aio_a8_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_SCAN_COMPLETE` - The conversion value was successfully read and stored to the "`RawResult`" variable.

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](#).

### 2.2.8 FS\_AIO\_LimitCheck()

This function executes the last part of the AIO test sequence and it is common for all ADC types. If the state is "FS\_AIO\_SCAN\_COMPLETE", the function checks if value from the "RawResult" input parameter is within the limits from the "pLimits" structure.

#### **Function prototype:**

```
FS_RESULT FS_AIO_LimitCheck(uint32_t RawResult, fs_aio_limits_t *pLimits, FS_RESULT *pState );
```

#### **Function inputs:**

*uint32\_t RawResult* - The input argument is the "RawResult" of the ADC conversion.

*\*pLimits* - The input argument is the pointer to the "fs\_aio\_limits\_t" structure with conversion limits.

*\*pState* - The input argument is the pointer to the "FS\_RESULT" variable.

#### **Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_FAIL\_AIO* - The input "RawResult" is not within the borders defined in "Limits".
- *FS\_PASS* - The input "RawResult" is in the border defined in "Limits".

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

#### **Function call example:**

The example of the function call is provided in [Section "Analog input/output test implementation"](#).

#### **Function performance:**

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

## 3 Clock test

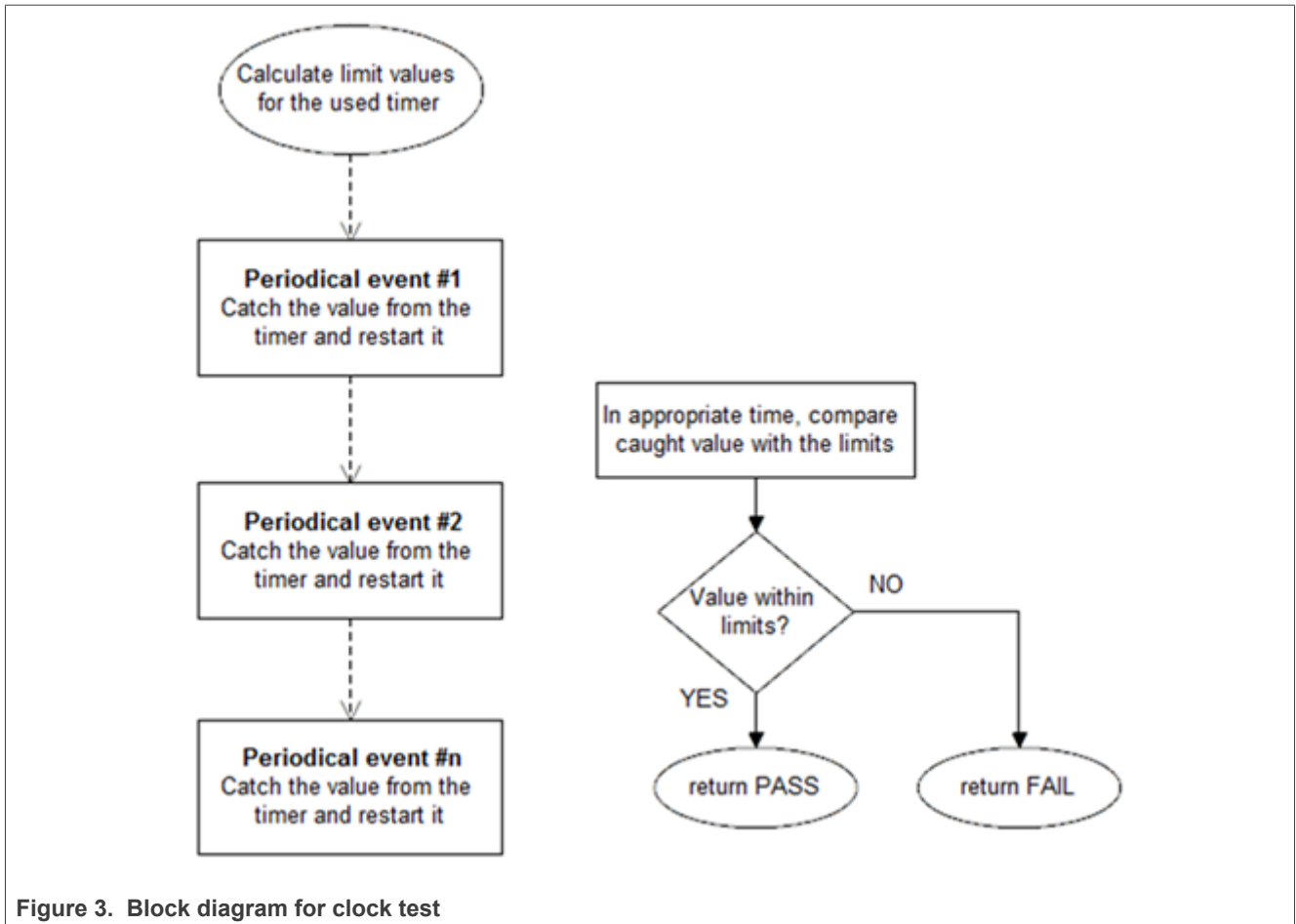
The clock test procedure tests the oscillators of the processor for the wrong frequency. The clock test can be performed once after the MCU reset and also during runtime.

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 3](#).



### 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 17. 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 in the *iec60730b\_clock.c* file and they are 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() / FS\_CLK\_STM()*
- *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, usually before the `while()` loop. The `FS_CLK_LPTMR()` (to choose the dedicated function for your device, see [Core self-test library – source code version](#)) function is then called within a periodic event. The `FS_CLK_Check()` function for evaluation can be called at any given time. When the test is in the initialization phase, the check function returns the “in progress” value. If the captured value from the reference counter is within the preset limits, the check function returns a pass value. If not, a defined fail value is returned.

The example of the test implementation is as follows:

```
#include "iec60730b.h"
FS_RESULT st;
unsigned long clockTestContext;
#define ISR_FREQUENCY (100)
#define CLOCK_TEST_TOLERANCE (10)
#define REF_TIMER_CLOCK_FREQUENCY (32e031)
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.

#### Function prototype:

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

#### Function inputs:

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

#### Function output:

*void*

#### Function performance:

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

### 3.2.2 FS\_CLK\_Check()

This function handles the clock test. It evaluates the captured value stored in the `testContext` variable with predefined limits. Until the first execution of the respective `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. Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#). 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\_RTC()**

This function is used only with the RTC module. Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#). 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](#).

### 3.2.5 FS\_CLK\_GPT()

This function is used only with the GPT module. Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#). 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\_CTIMER()

This function is used only with the CTimer module. Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#). 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(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](#).

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

This function is used only with the WKT module. Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#). 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](#).

### 3.2.8 FS\_CLK\_STM()

This function is used only with the STM module. Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#). This function reads the counter value from the timer and saves it into the TestContext variable. After that, it starts the STM again.

**Function prototype:**

```
void FS_CLK_STM(fs_stm_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](#).

## 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 18](#).

Table 18. 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\_InputExt\_RGPIO()* / *FS\_DIO\_InputExt\_MCX()*
- *FS\_DIO\_ShortToSupplySet()* / *FS\_DIO\_ShortToSupplySet\_IMXRT()* / *FS\_DIO\_ShortToSupplySet\_IMX8M()* / *FS\_DIO\_ShortToSupplySet\_LPC()* / *FS\_DIO\_ShortToSupplySet\_RGPIO()* / *FS\_DIO\_ShortToSupplySet\_MCX()*
- *FS\_DIO\_ShortToAdjSet()* / *FS\_DIO\_ShortToAdjSet\_IMXRT()* / *FS\_DIO\_ShortToAdjSet\_IMX8M()* / *FS\_DIO\_ShortToAdjSet\_LPC()* / *FS\_DIO\_ShortToAdjSet\_RGPIO()* / *FS\_DIO\_ShortToAdjSet\_MCX()*

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 [Figure 4](#):

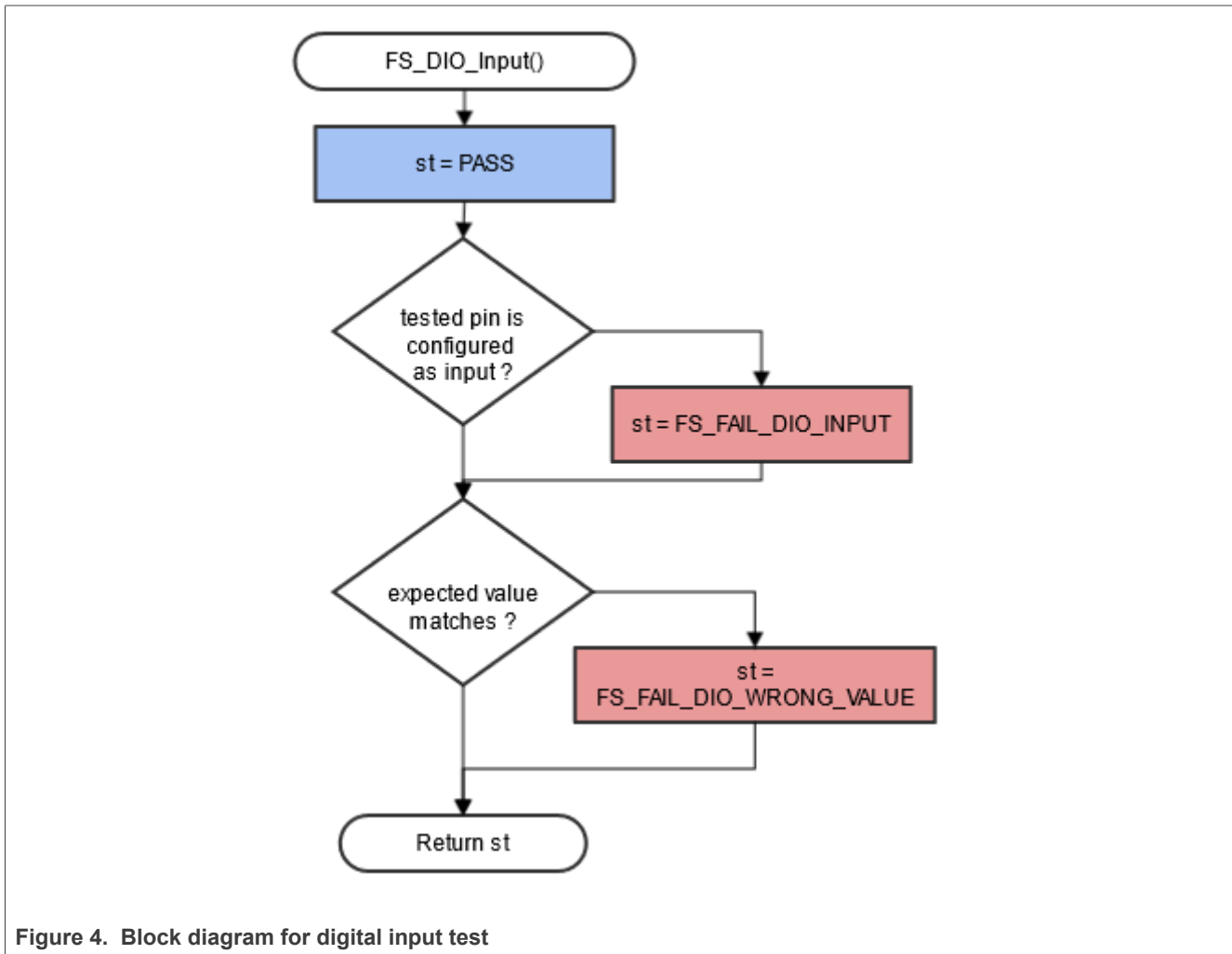


Figure 4. Block diagram for digital input test

**Function prototype:**

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

**Function inputs:**

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

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

**Function output:**

`typedef uint32_t FS_RESULT;`

- `FS_PASS`
- `FS_FAIL_DIO_INPUT`- The pin is not set as the input.
- `FS_FAIL_DIO_WRONG_VALUE` - The pin does not have the expected value.

The function always returns the **first** detected error.

**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.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

#### 4.2.2 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.

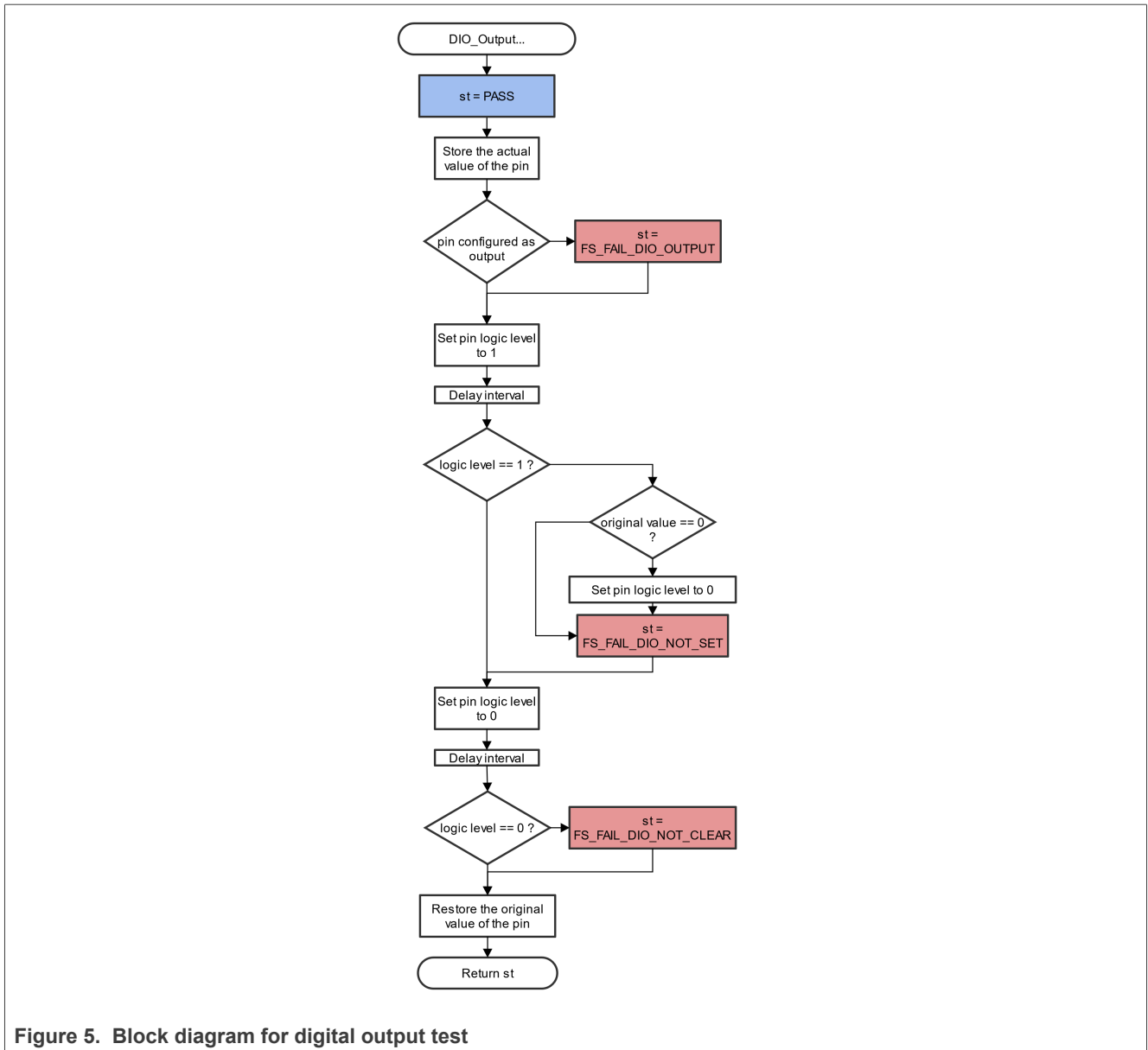


Figure 5. Block diagram for digital output test

**Function prototype:**

`FS_RESULT FS_DIO_Output(fs_dio_test_t *pTestedPin, uint32_t delay);`

**Function inputs:**

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

*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_OUTPUT` - The pin is not set as the output.
- `FS_FAIL_DIO_NOT_SET` - The pin cannot be set to logical 1.
- `FS_FAIL_DIO_NOT_CLEAR` - The pin cannot be cleared to logical 0.

The function always returns the **first** detected error.

**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.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

**4.2.3 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 [Figure 6](#). 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.

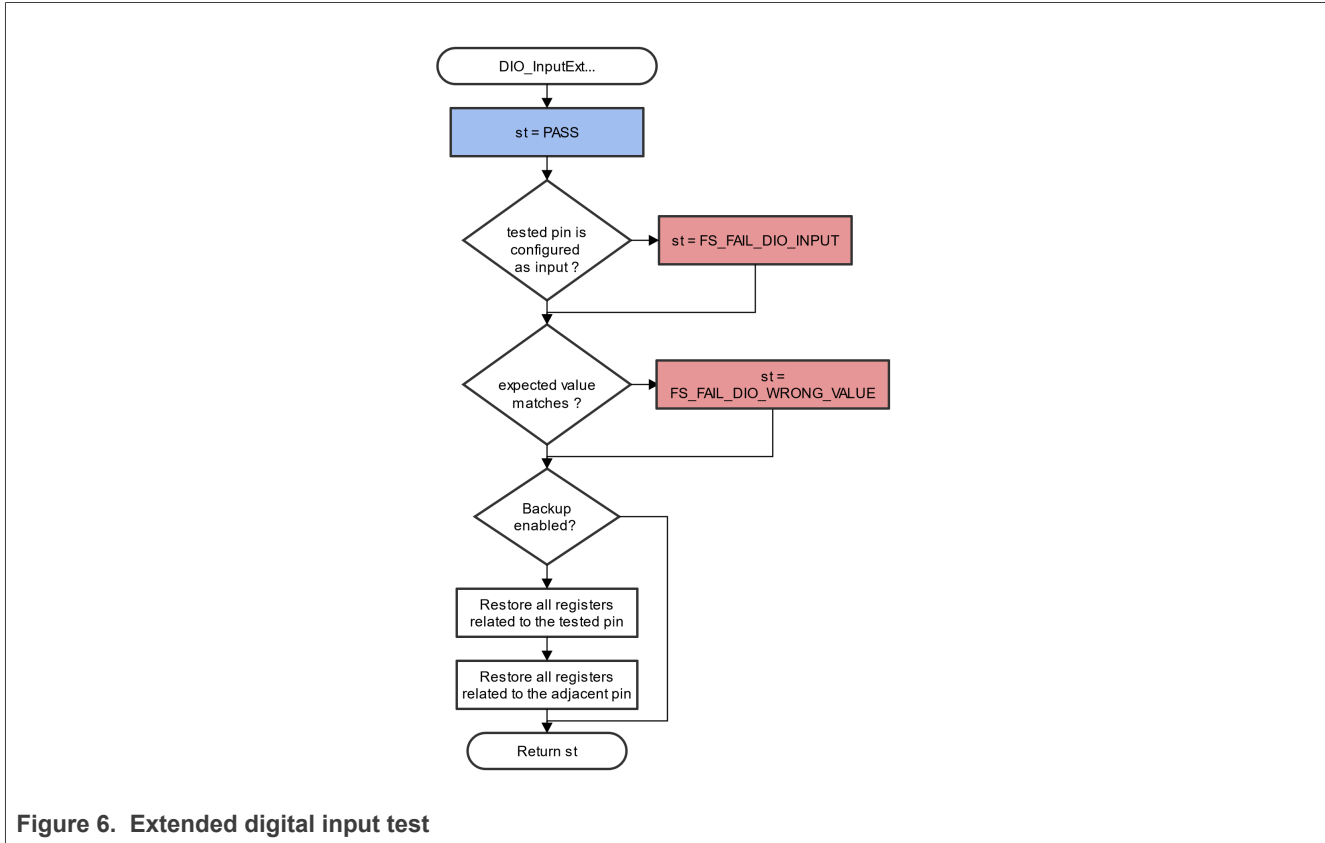


Figure 6. Extended digital input test

**Function prototype:**

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

**Function inputs:**

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

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

*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\_INPUT* - The pin is not set as the input.
- *FS\_FAIL\_DIO\_WRONG\_VALUE* - The pin does not have the expected value.

The function always returns the **first** detected error.

**Example of function call:**

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

**Function performance:**

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

**Calling restrictions:**

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

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

#### 4.2.4 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 7](#). 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.

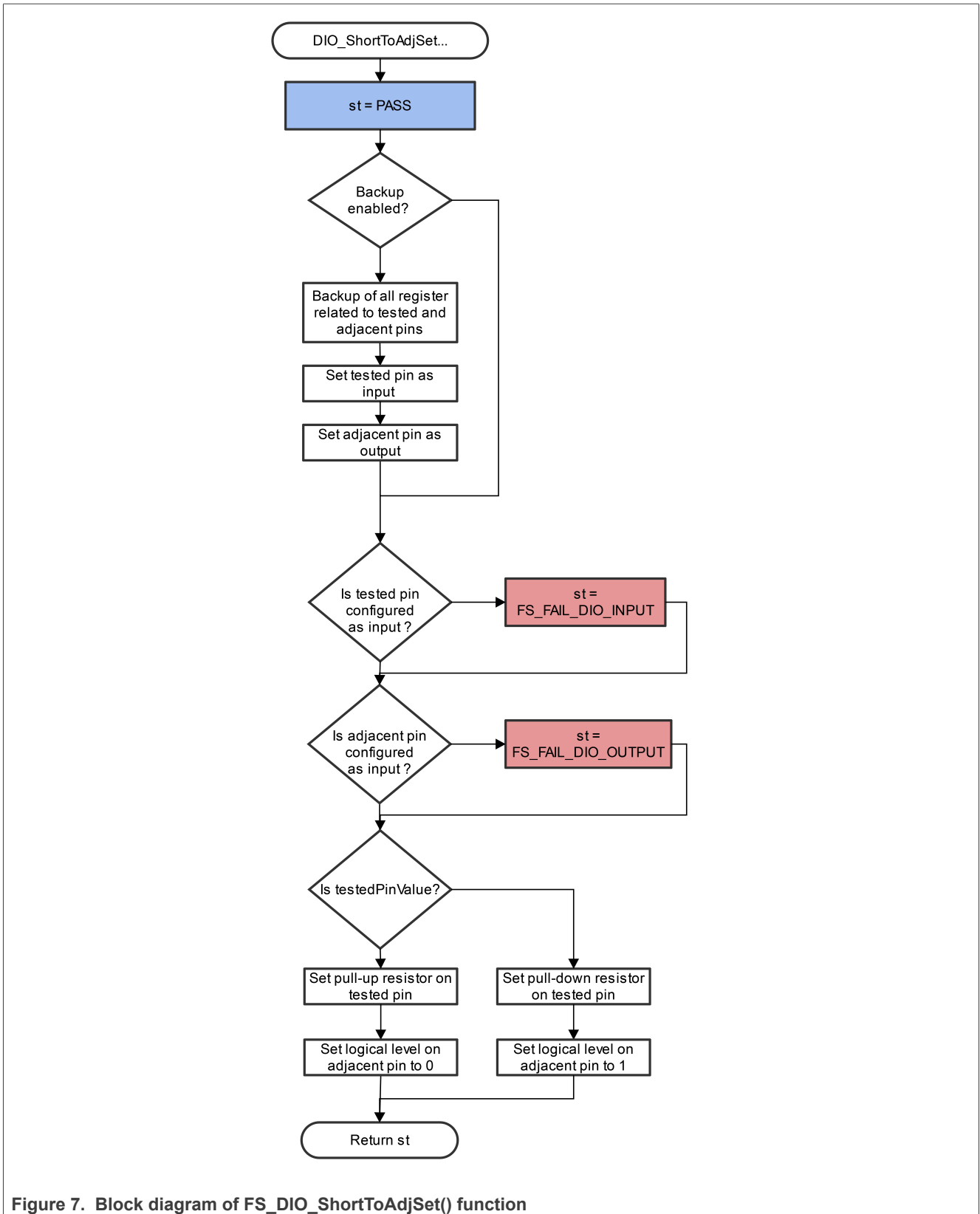


Figure 7. Block diagram of FS\_DIO\_ShortToAdjSet() function

**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 structure.

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

*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\_INPUT* - The tested pin is not set as the input.
- *FS\_FAIL\_DIO\_OUTPUT* - The adjacent pin is not set as the output.

Function always returns the **first** detected error.

**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 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.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

#### 4.2.5 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 8](#). 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).

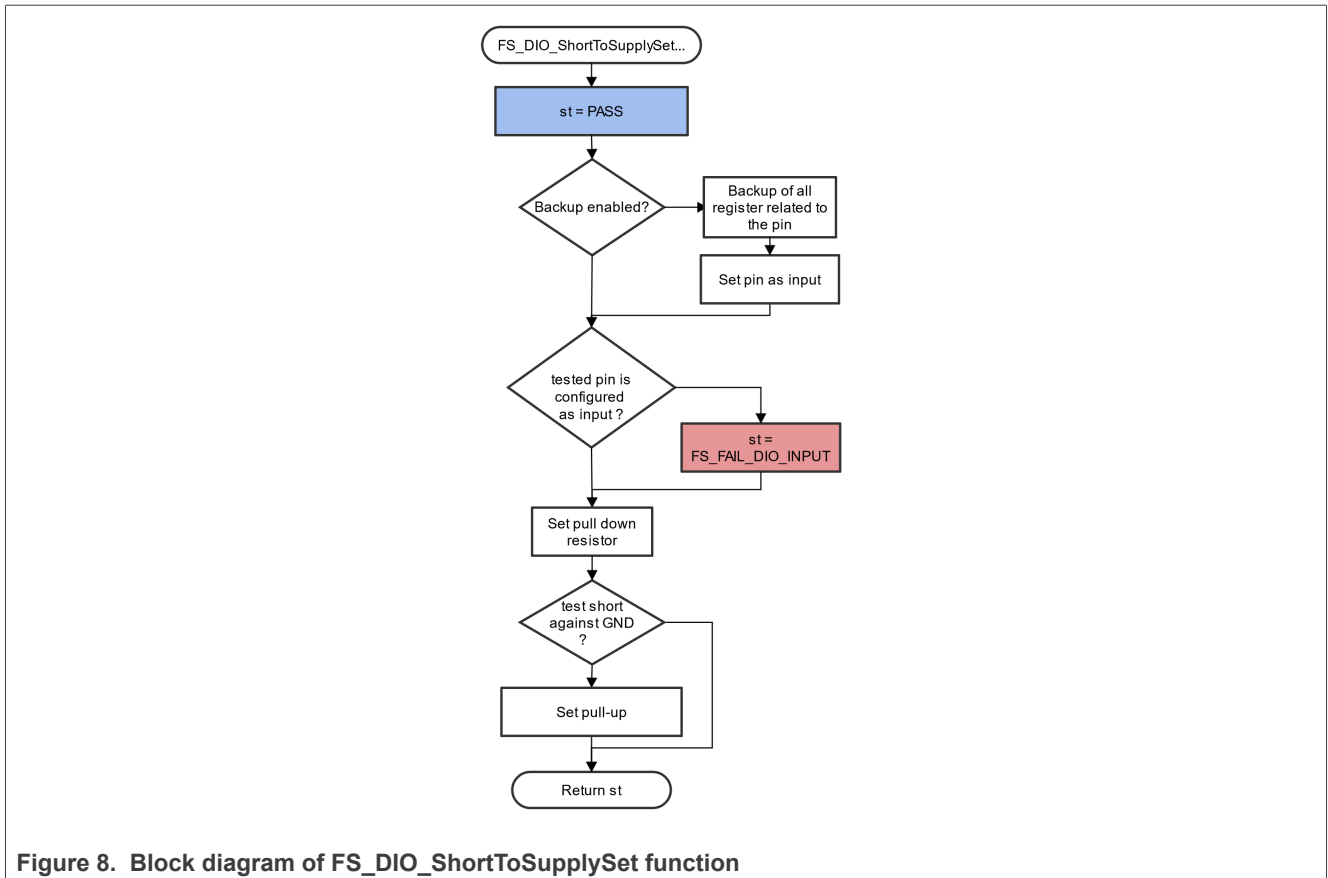


Figure 8. Block diagram of FS\_DIO\_ShortToSupplySet function

**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 structure.

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\_INPUT - The pin is not set as the input.

The function always returns the **first** detected error.

**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 the other way around.

```
#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 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.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

### 4.2.6 FS\_DIO\_InputExt\_MCX()

This is a modified version of the previously mentioned digital input test. 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 is shown in [Figure 6](#). 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.

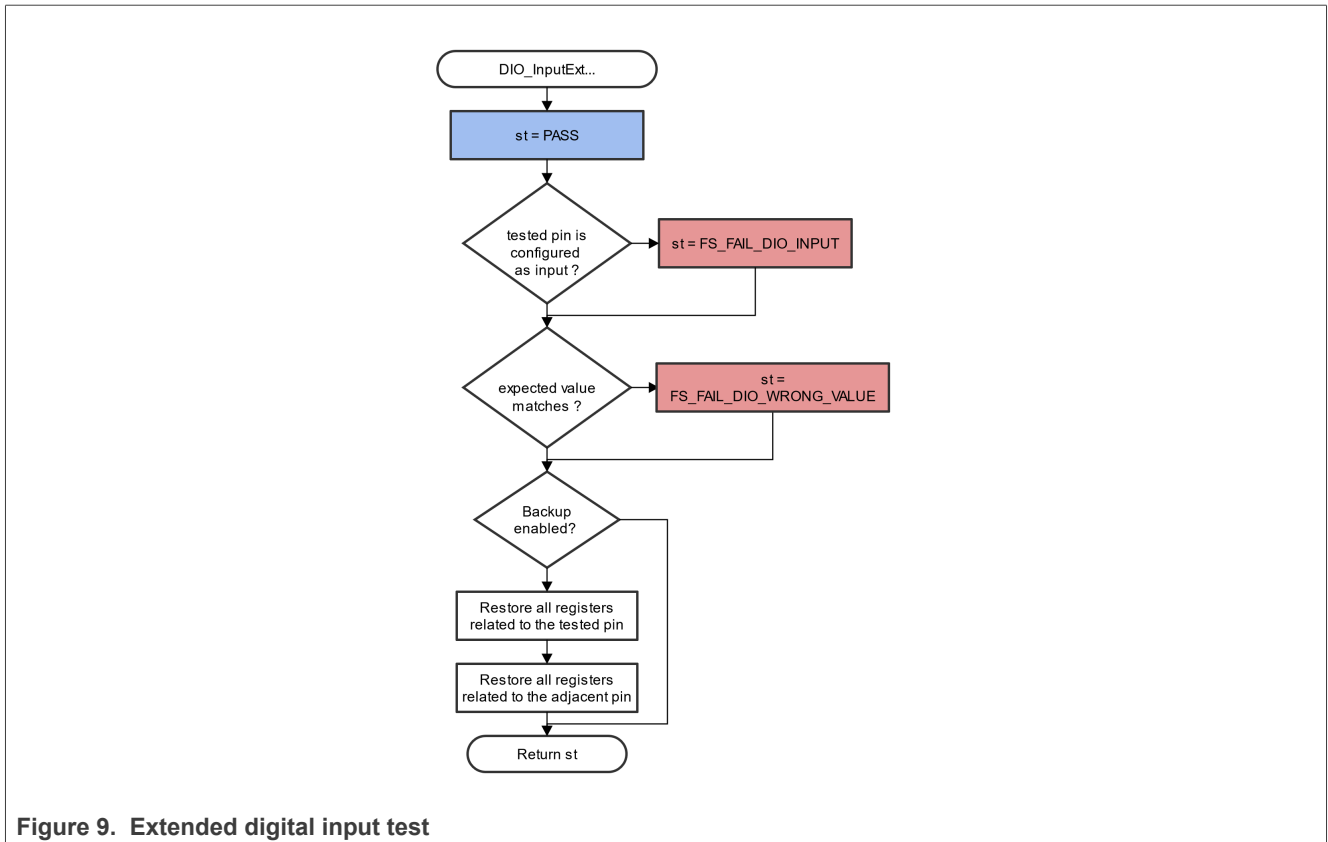


Figure 9. Extended digital input test

**Function prototype:**

```
FS_RESULT FS_DIO_InputExt_MCX(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 structure.

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

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\_WRONG\_VALUE - Different value on pin against the settings.
- FS\_FAIL\_DIO\_INPUT - The pin is not set as the input.

The function always returns the **first** detected error.

**Example of function call:**

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

**Function performance:**

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

**Calling restrictions:**

The 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`.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

#### 4.2.7 FS\_DIO\_ShortToAdjSet\_MCX()

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 [Block diagram of FS\\_DIO\\_ShortToAdjSet\\_MCX\(\) function](#). 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_MCX()` function is described in the respective section. Specify the tested pin and the adjacent pin for the input test function.

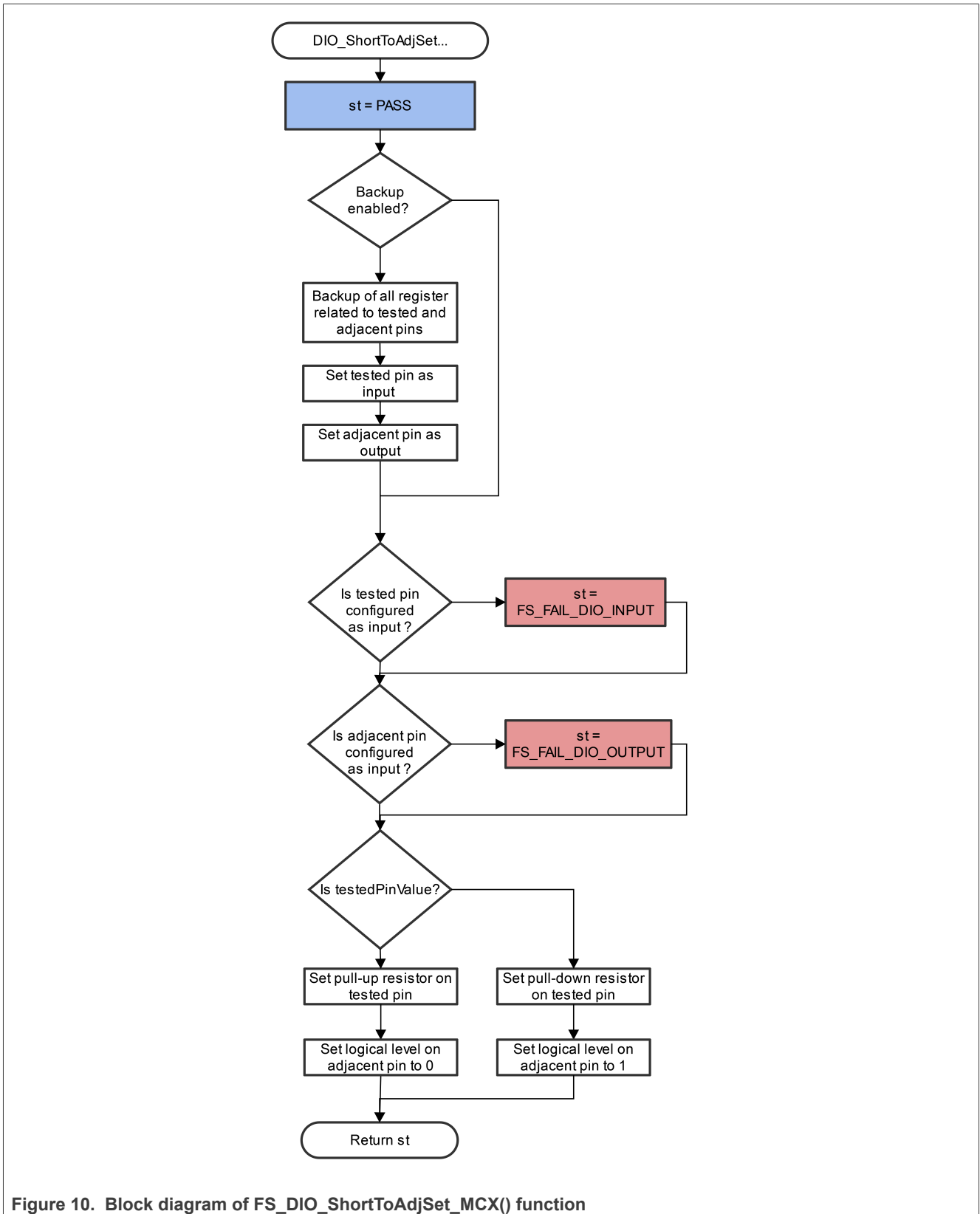


Figure 10. Block diagram of FS\_DIO\_ShortToAdjSet\_MCX() function

**Function prototype:**

```
FS_RESULT FS_DIO_ShortToAdjSet_MCX(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 structure.

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

*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\_INPUT* - The tested pin is not set as the input.
- *FS\_FAIL\_DIO\_OUTPUT* - The adjacent pin is not set as the output.

The function always returns the **first** detected error.

**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_MCX(&dio_safety_test_items[0], &dio_safety_test_items[1],
  LOGICAL_ONE, BACKUP_ENABLE);
dio_short_to_adj_test_result =FS_DIO_InputExt_MCX(&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 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\_MCX()* function is called for these pins.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

**4.2.8 FS\_DIO\_ShortToSupplySet\_MCX()**

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 [Block diagram of FS\\_DIO\\_ShortToSupplySet\\_MCX function](#). The second part of the test (result evaluation) is ensured by the *FS\_DIO\_InputExt\_MCX()* function that is described in the respective section. The main purpose of the *FS\_DIO\_InputExt\_MCX()* 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).

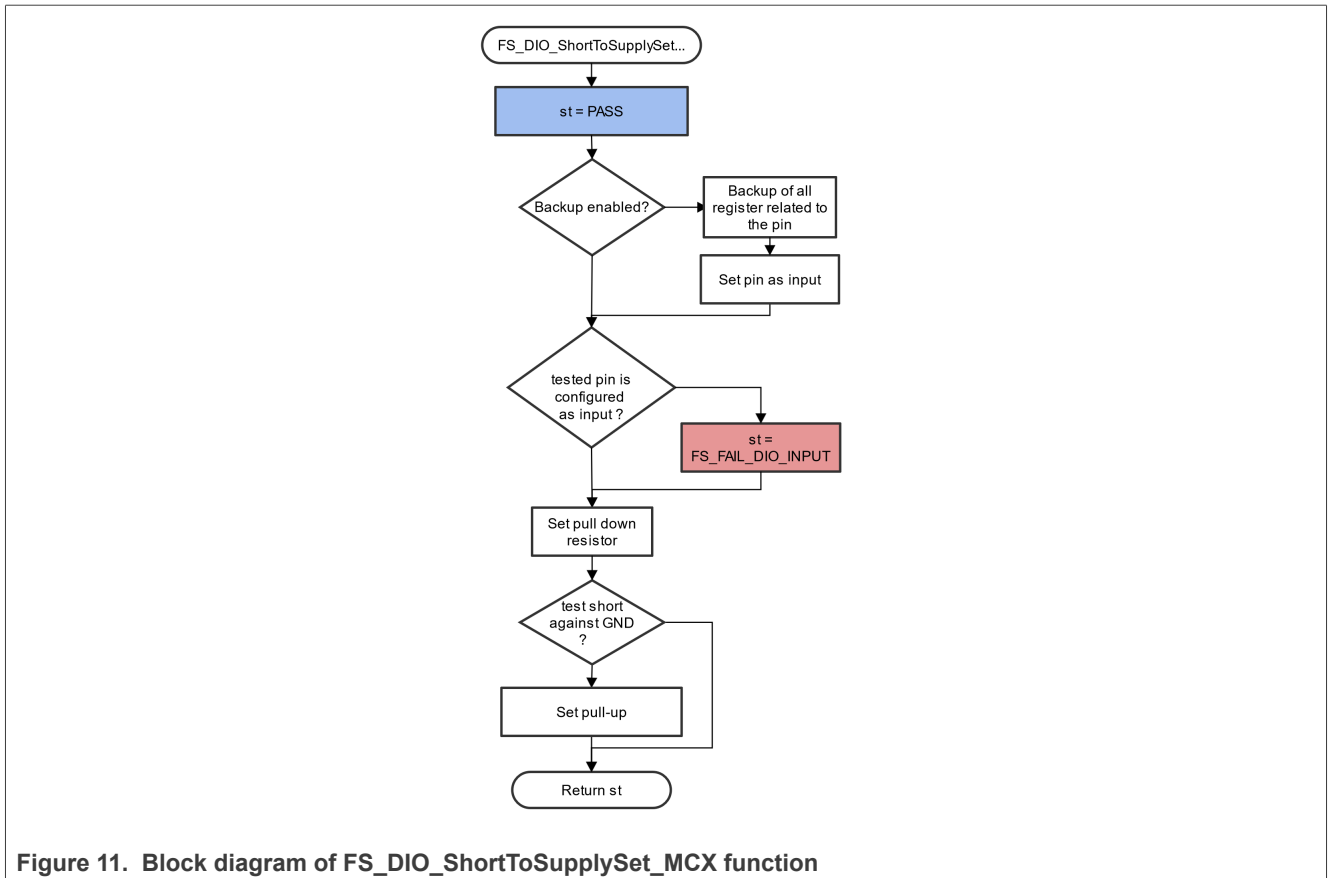


Figure 11. Block diagram of FS\_DIO\_ShortToSupplySet\_MCX function

**Function prototype:**

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

**Function inputs:**

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

*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\_INPUT* - The pin is not set as the input.

The function always returns the **first** detected error.

**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 the other way around.

```
#define DIO_SHORT_TO_GND_TEST 1
```

```
#define DIO_SHORT_TO_VDD_TEST 0
dio_short_to_vcc_test_result =
  FS_DIO_ShortToSupplySet_MCX(&dio_safety_test_items[0], DIO_SHORT_TO_GND_TEST,
  BACKUP_ENABLE);
dio_short_to_vcc_test_result = FS_DIO_InputExt_MCX(&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_MCX(&dio_safety_test_items[0], DIO_SHORT_TO_VDD_TEST,
  BACKUP_ENABLE);
dio_short_to_vcc_test_result = FS_DIO_InputExt_MCX(&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\_MCX()* function is called for the tested pin.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

### 4.2.9 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 12](#). 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.

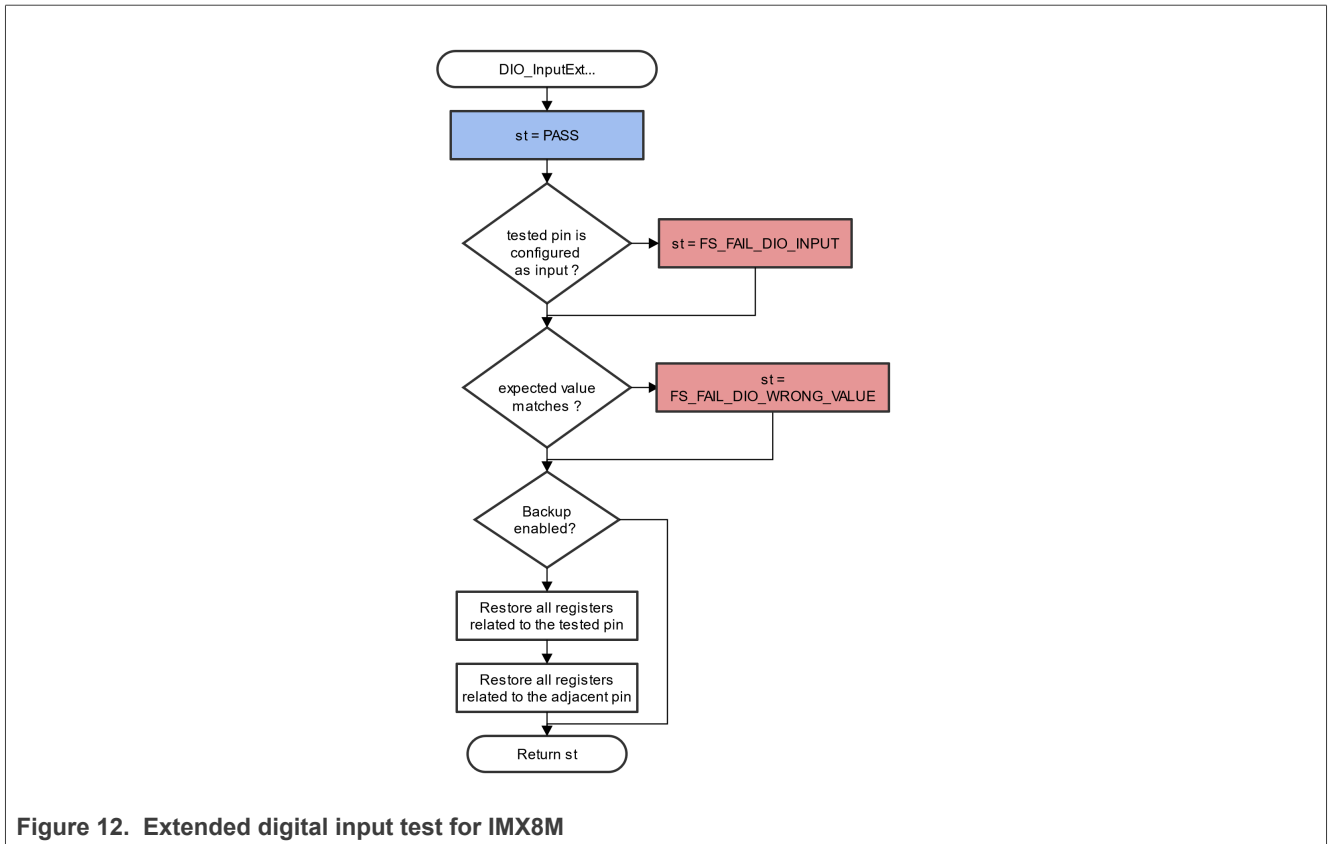


Figure 12. Extended digital input test for IMX8M

**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 structure.

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

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\_INPUT - The pin is not set as the input.
- FS\_FAIL\_DIO\_WRONG\_VALUE - The pin does not have the expected value.

The function always returns the **first** detected error.

**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:**

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 to enter the same input as for "TestedPin".

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

**4.2.10 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.

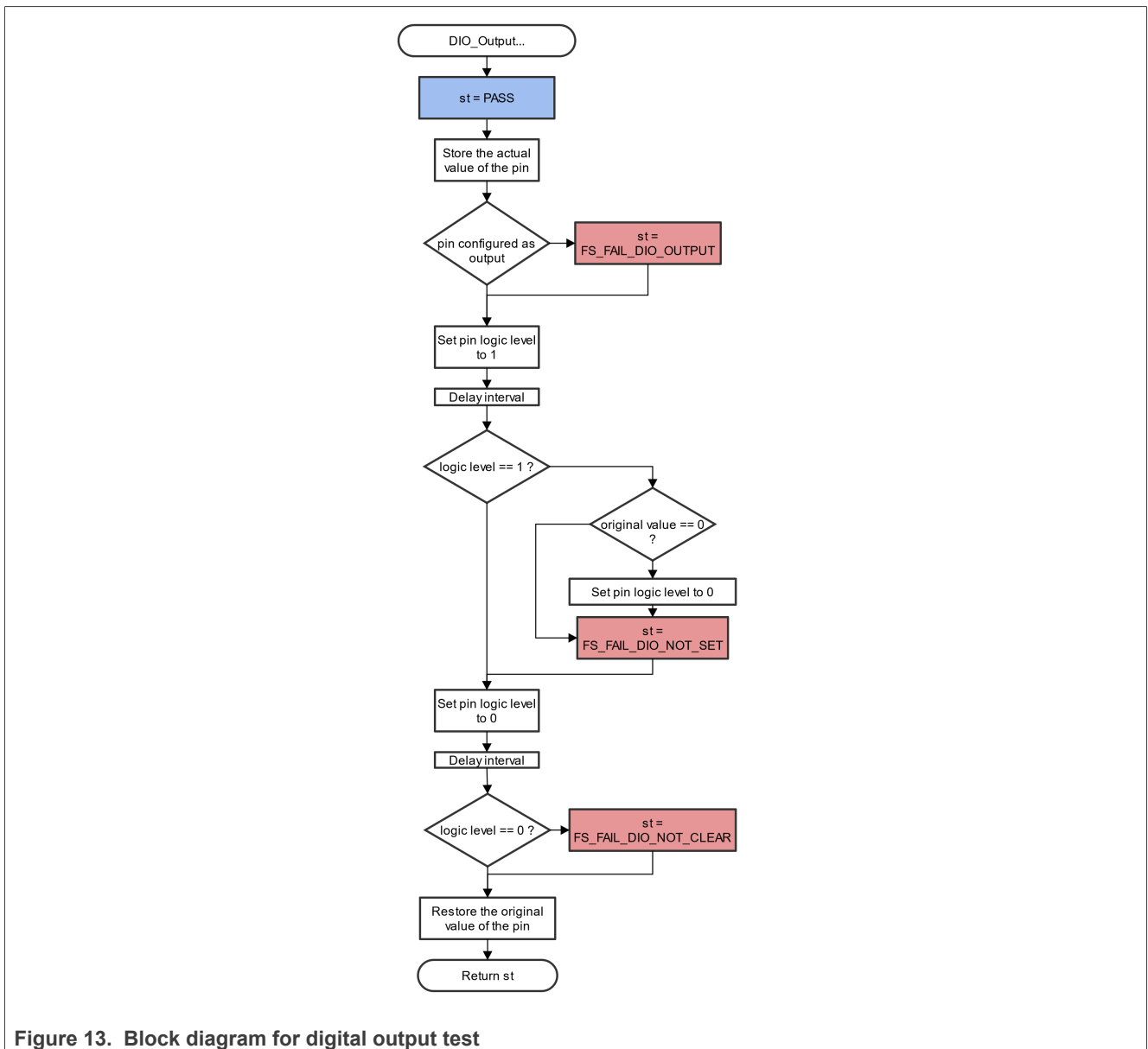


Figure 13. Block diagram for digital output test

**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 structure.

*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\_OUTPUT* - The pin is not set as the output.
- *FS\_FAIL\_DIO\_NOT\_SET* - The pin cannot be set to logical 1.
- *FS\_FAIL\_DIO\_NOT\_CLEAR* - The pin cannot be cleared to logical 0.

The function always returns the **first** detected error.

**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.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

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

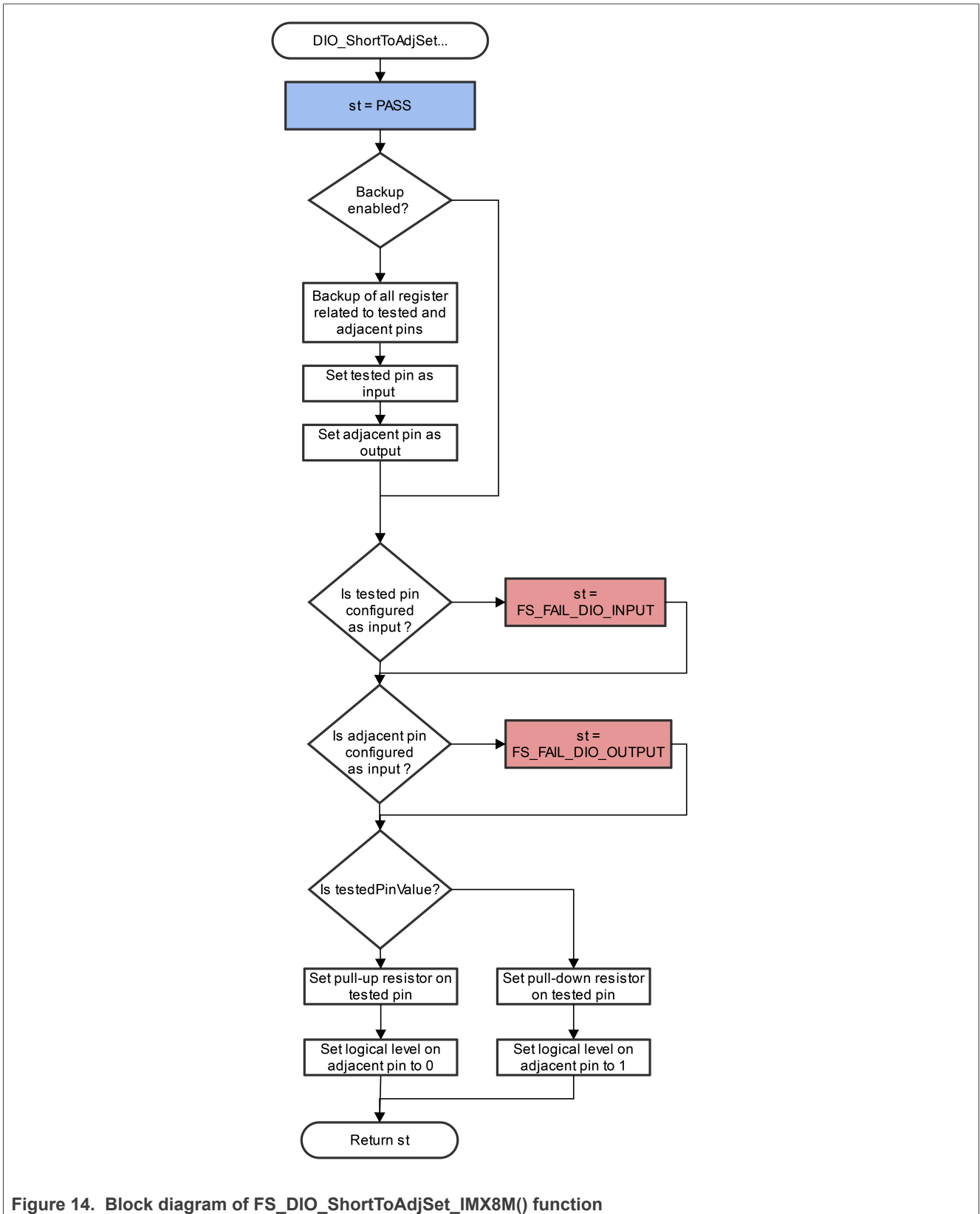


Figure 14. Block diagram of FS\_DIO\_ShortToAdjSet\_IMX8M() function

**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 structure.

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

*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\_INPUT* - The tested pin is not set as the input.
- *FS\_FAIL\_DIO\_OUTPUT* - The adjacent pin is not set as the output.

The function always returns the **first** detected error.

**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 is called for these pins.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

**4.2.12 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 15](#). 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).

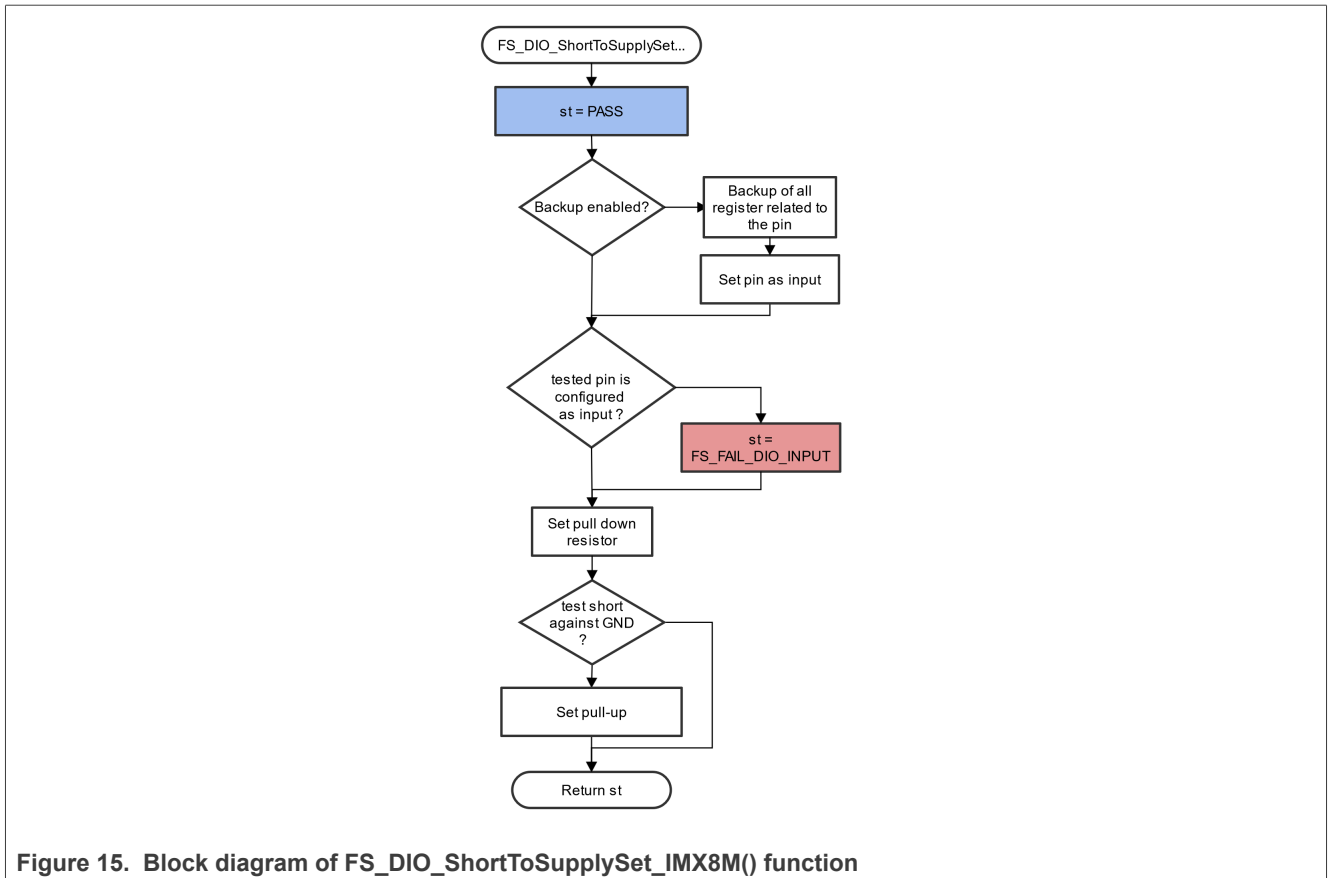


Figure 15. Block diagram of FS\_DIO\_ShortToSupplySet\_IMX8M() function

**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 structure.

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\_INPUT - The pin is not set as the input.

The function always returns the **first** detected error.

**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 the other way around).

```
#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.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

### 4.2.13 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 16](#). 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.

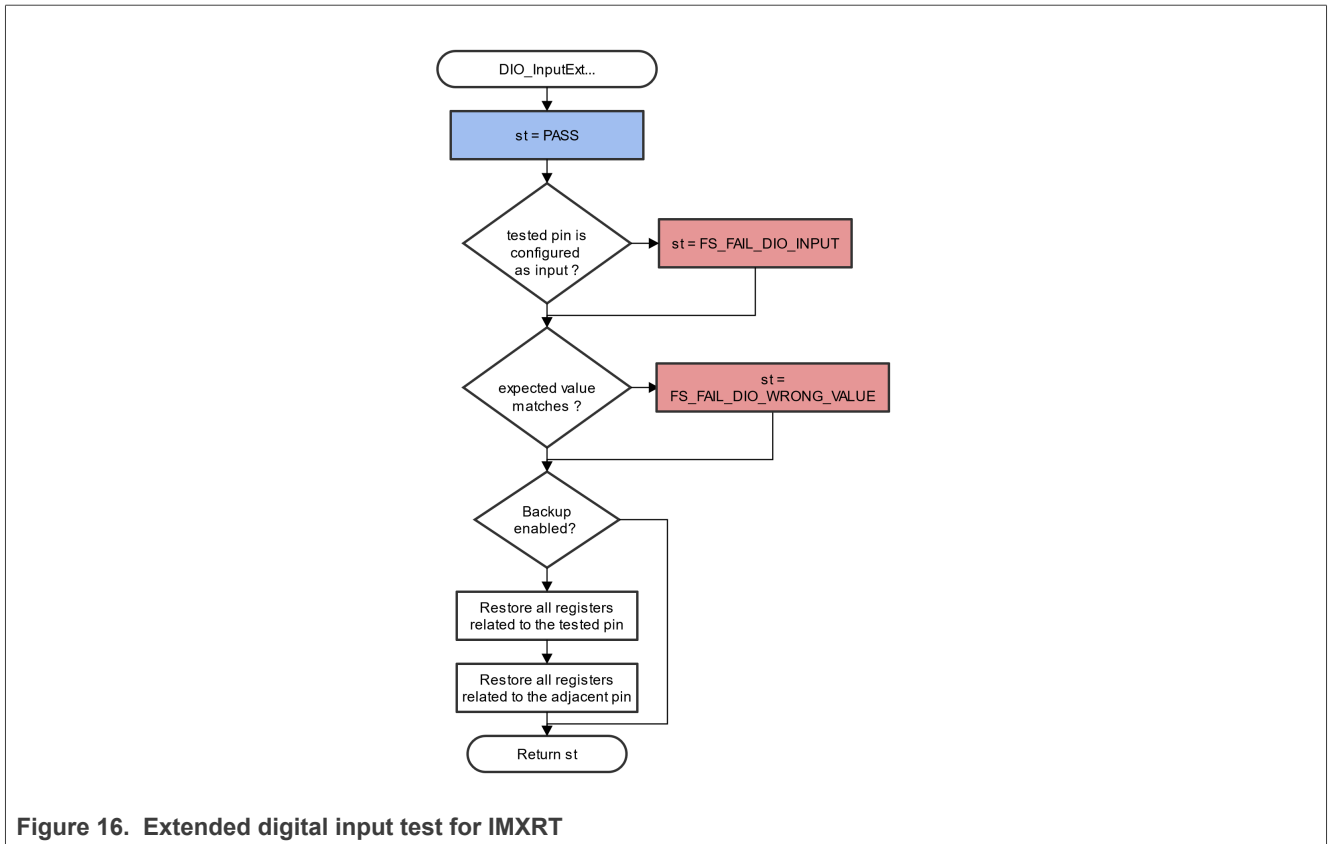


Figure 16. Extended digital input test for IMXRT

**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 structure.

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

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\_INPUT - The pin is not set as the input.
- FS\_FAIL\_DIO\_WRONG\_VALUE - The pin does not have the expected value.

The function always returns the **first** detected error.

**Example of function call:**

```
fs_dio_input_test_result = FS_DIO_InputExt_IMXRT(&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:**

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 to enter the same input as for "TestedPin".

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

**4.2.14 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.

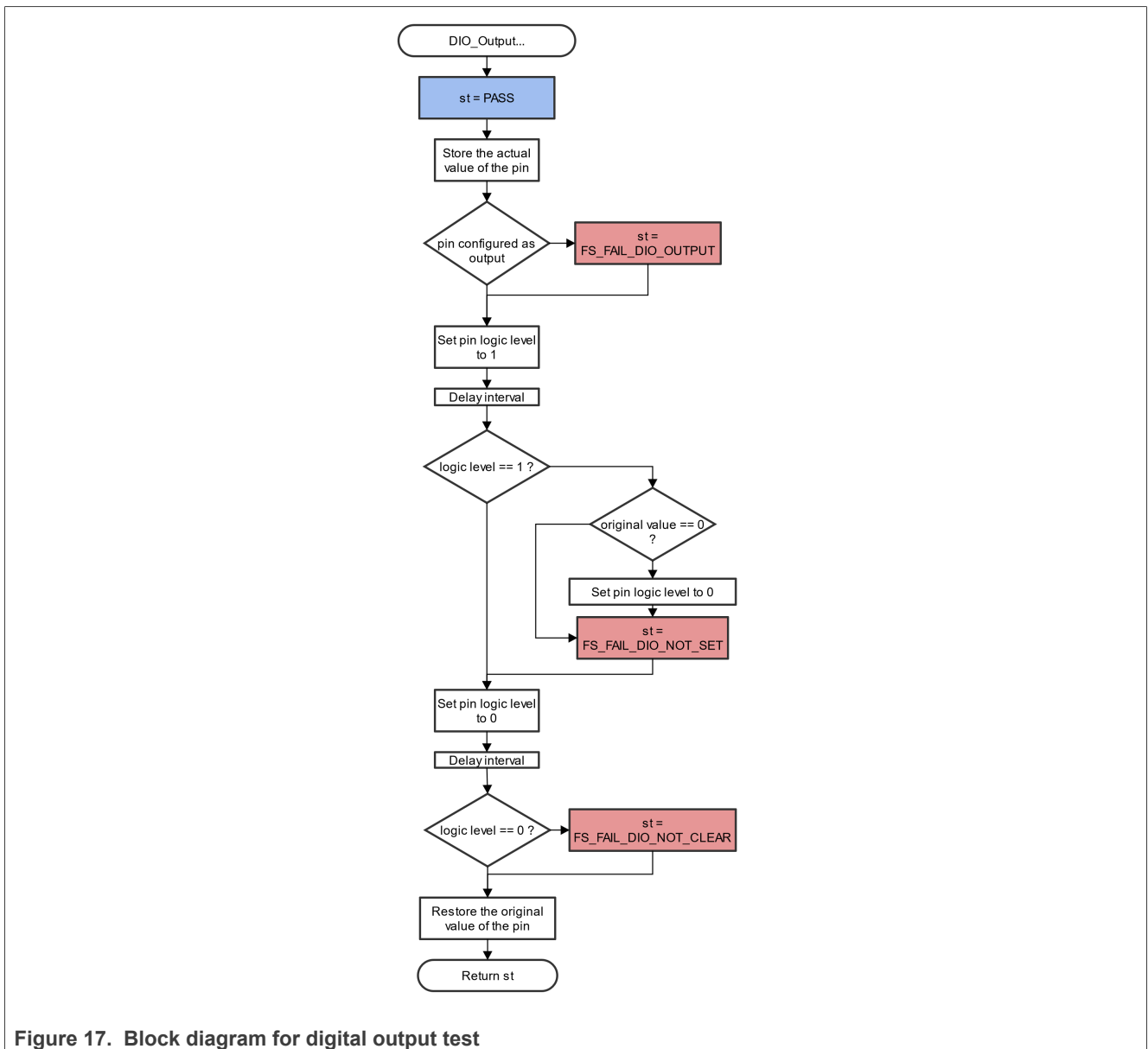


Figure 17. Block diagram for digital output test

**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 structure.

*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\_OUTPUT* - The pin is not set as the output.
- *FS\_FAIL\_DIO\_NOT\_SET* - The pin cannot be set to logical 1.
- *FS\_FAIL\_DIO\_NOT\_CLEAR* - The pin cannot be cleared to logical 0.

The function always returns the **first** detected error.

**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.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

**4.2.15 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 18](#). 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.

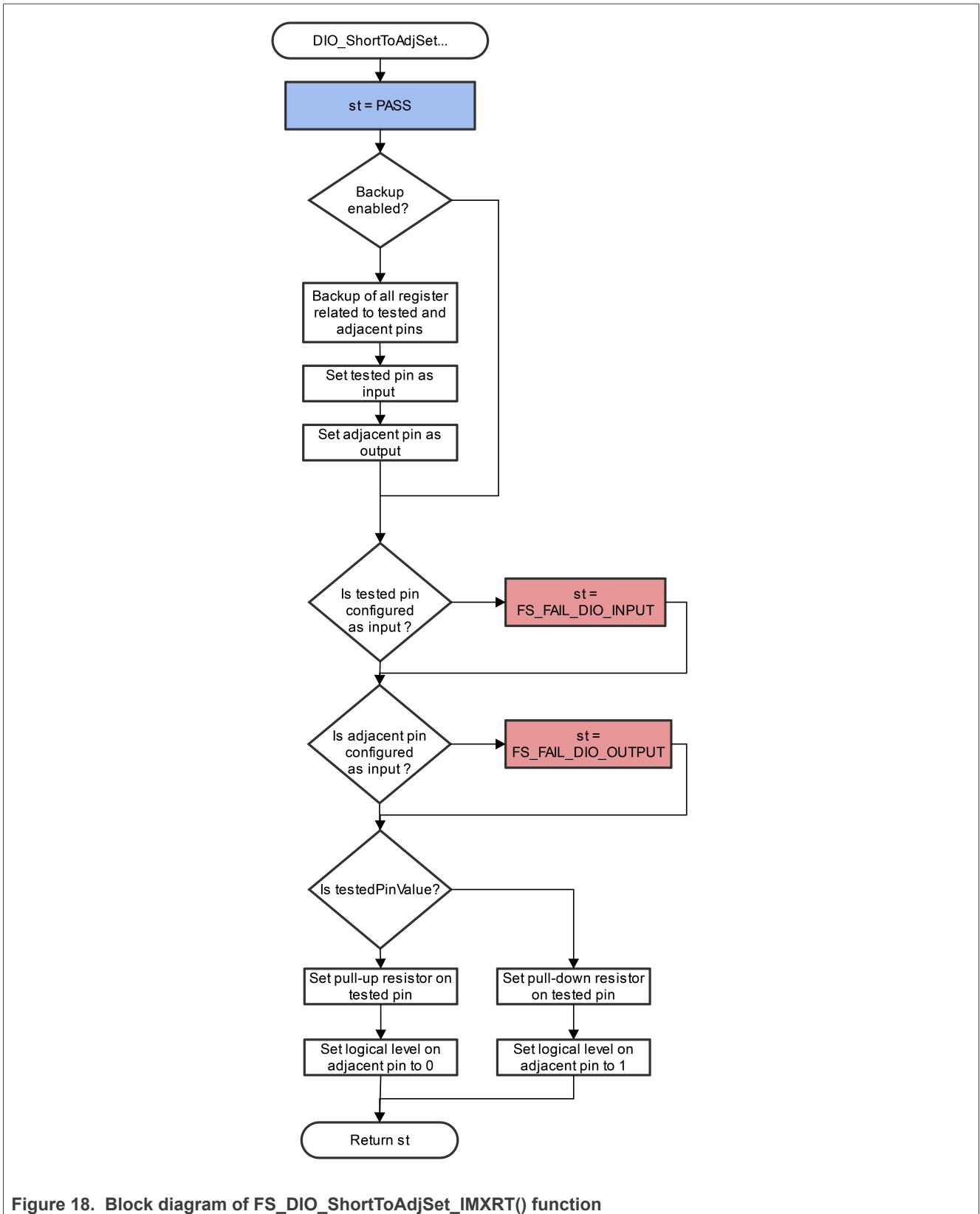


Figure 18. Block diagram of FS\_DIO\_ShortToAdjSet\_IMXRT() function

**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 structure.

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

*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\_INPUT* - The tested pin is not set as the input.
- *FS\_FAIL\_DIO\_OUTPUT* - The adjacent pin is not set as the output.

The function always returns the **first** detected error.

**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.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

**4.2.16 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 19](#). 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).

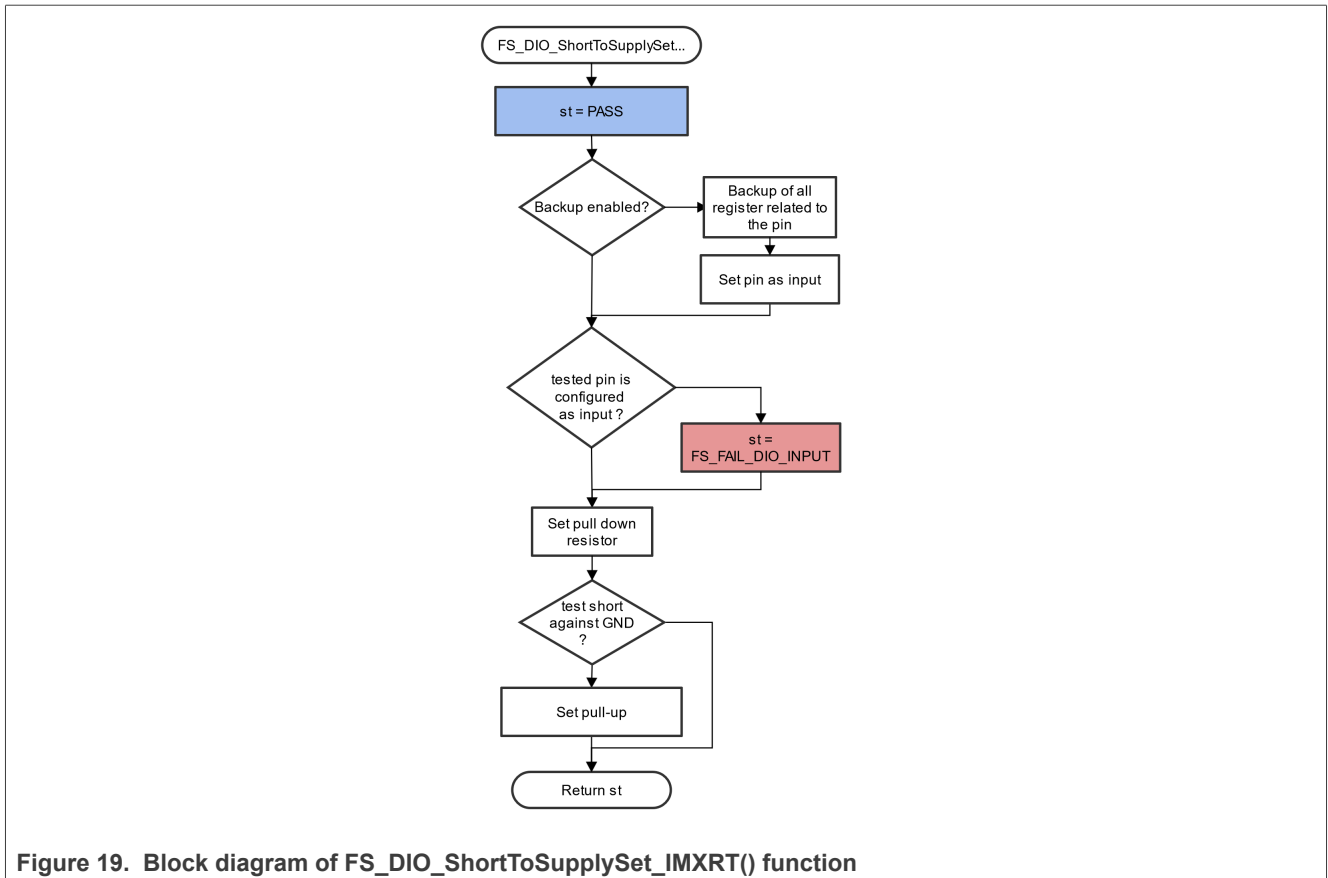


Figure 19. Block diagram of FS\_DIO\_ShortToSupplySet\_IMXRT() function

**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 structure.

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\_INPUT - The pin is not set as the input.

The function always returns the **first** detected error.

**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 the other way around).

```
#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.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

### 4.2.17 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 [Figure 20](#). 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.

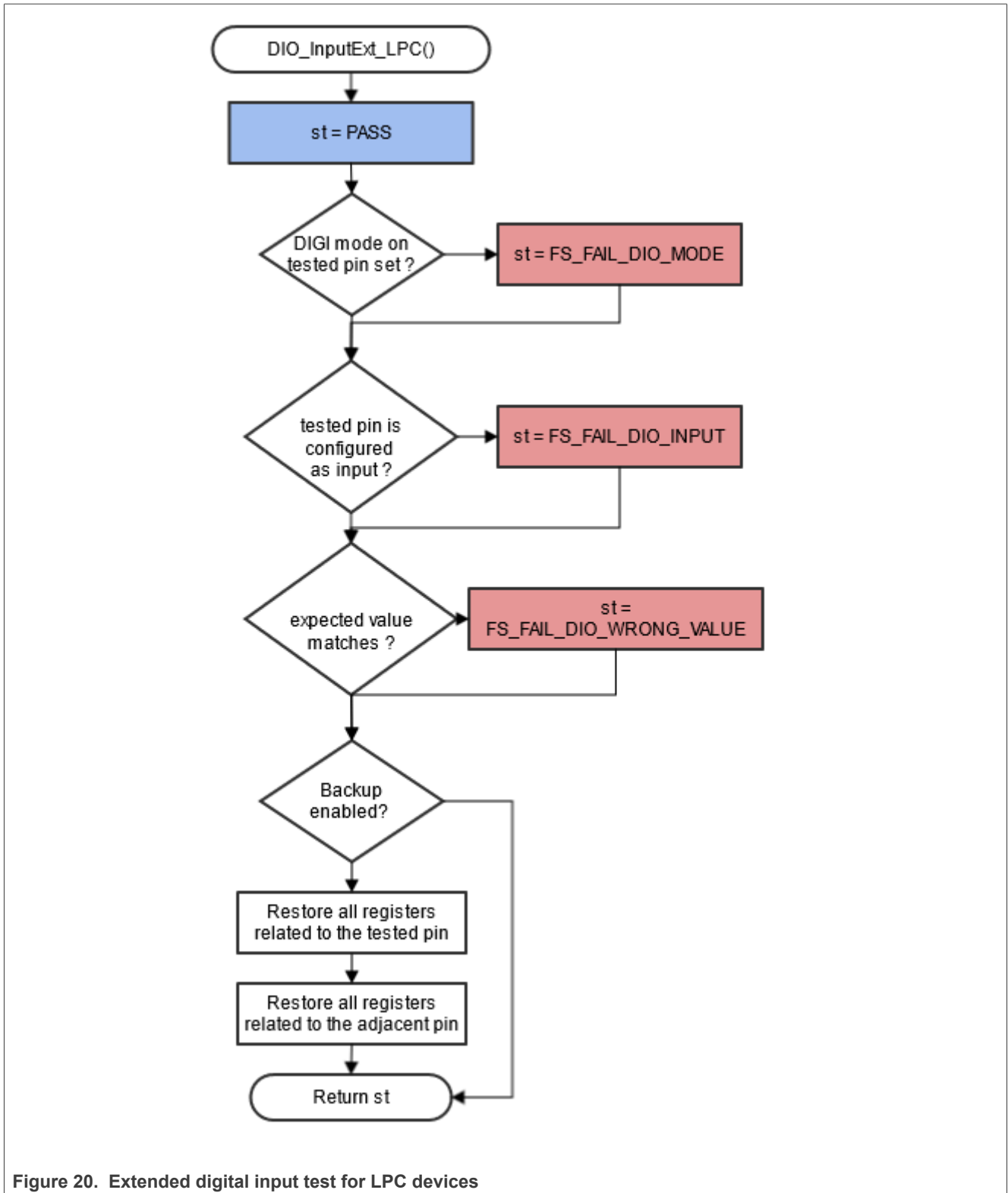


Figure 20. Extended digital input test for LPC devices

**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 structure.

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

*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\_INPUT* - The pin is not set as the input.
- *FS\_FAIL\_DIO\_WRONG\_VALUE* - The pin does not have the expected value.
- *FS\_FAIL\_DIO\_MODE* - The pin does not have the "digimode" set - only for a specific LPC device.

Function always returns the **first** detected error.

**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.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

#### 4.2.18 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.

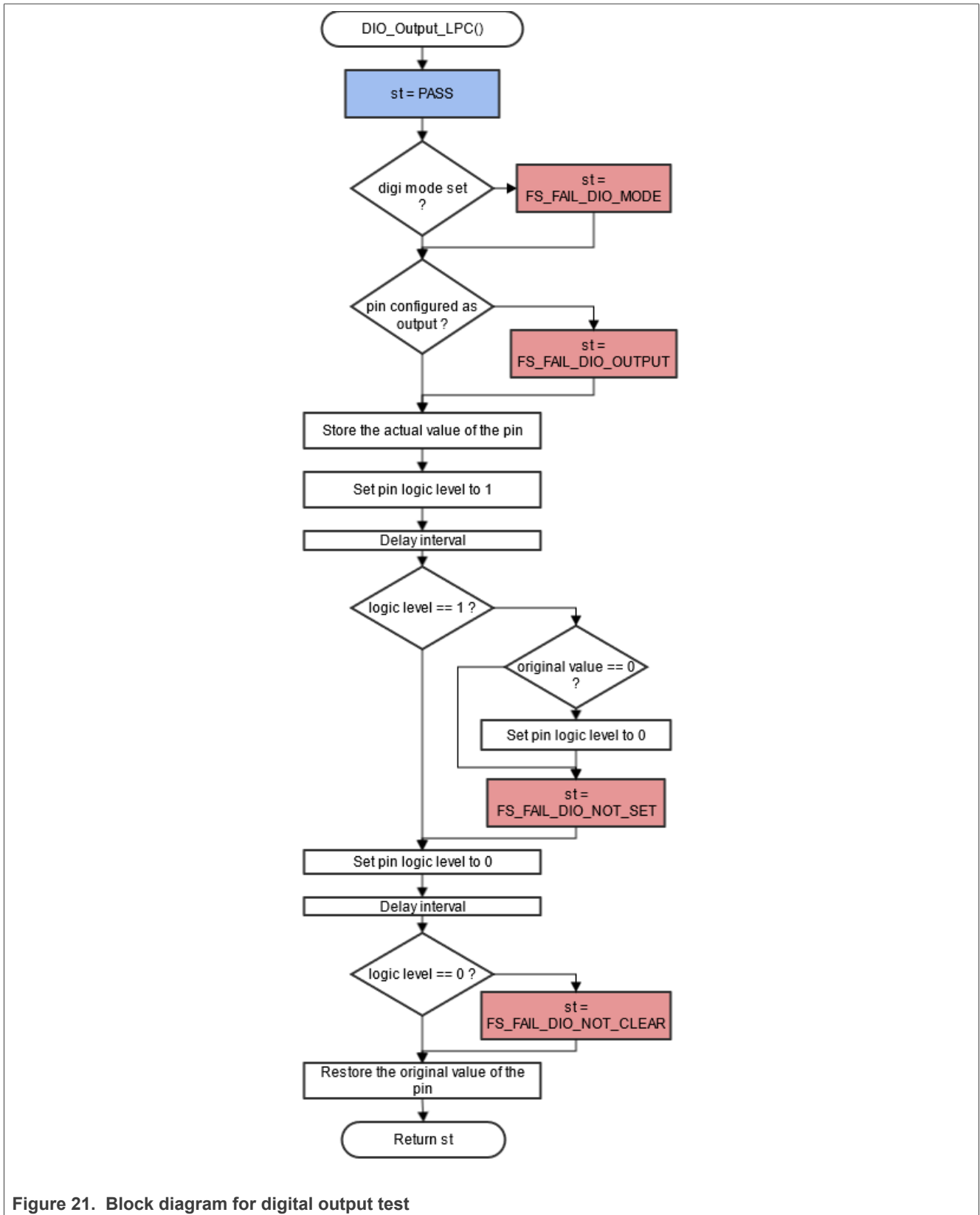


Figure 21. Block diagram for digital output test

**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 structure.

*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\_OUTPUT* - The pin is not set as the output.
- *FS\_FAIL\_DIO\_NOT\_SET* - The pin cannot be set to logical 1.
- *FS\_FAIL\_DIO\_NOT\_CLEAR* - The pin cannot be cleared to logical 0.
- *FS\_FAIL\_DIO\_MODE* - The pin does not have the "digimode" set - only for specific LPC devices.

The function always returns the **first** detected error.

**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.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

#### 4.2.19 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 22](#). 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.

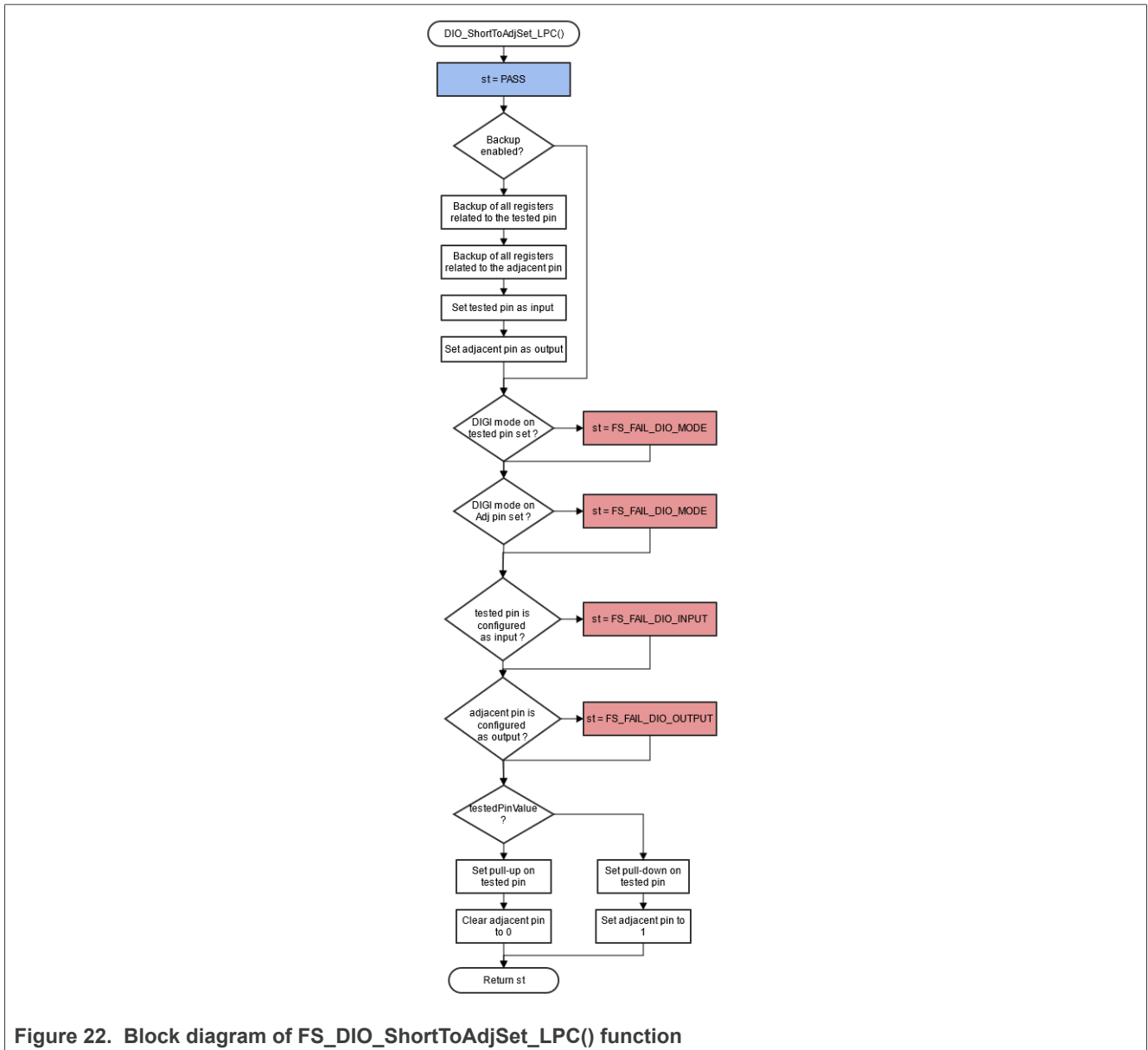


Figure 22. Block diagram of FS\_DIO\_ShortToAdjSet\_LPC() function

**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 structure.

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

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\_INPUT* - The tested pin is not set as the input.
- *FS\_FAIL\_DIO\_OUTPUT* - The adjacent pin is not set as the output.
- *FS\_FAIL\_DIO\_MODE* - The tested or adjacent pins do not have the "digimode" set - only for specific LPC devices.

The function always returns the **first** detected error.

#### **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 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 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 can manipulate neither the tested nor the adjacent pins until the *FS\_DIO\_InputExt\_LPC()* function is called for these pins.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

#### **4.2.20 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 ( $V_{cc}$ ,  $V_{dd}$ ) or between the tested pin and the hardware ground (GND). Its block diagram is shown in [Figure 23](#). 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).

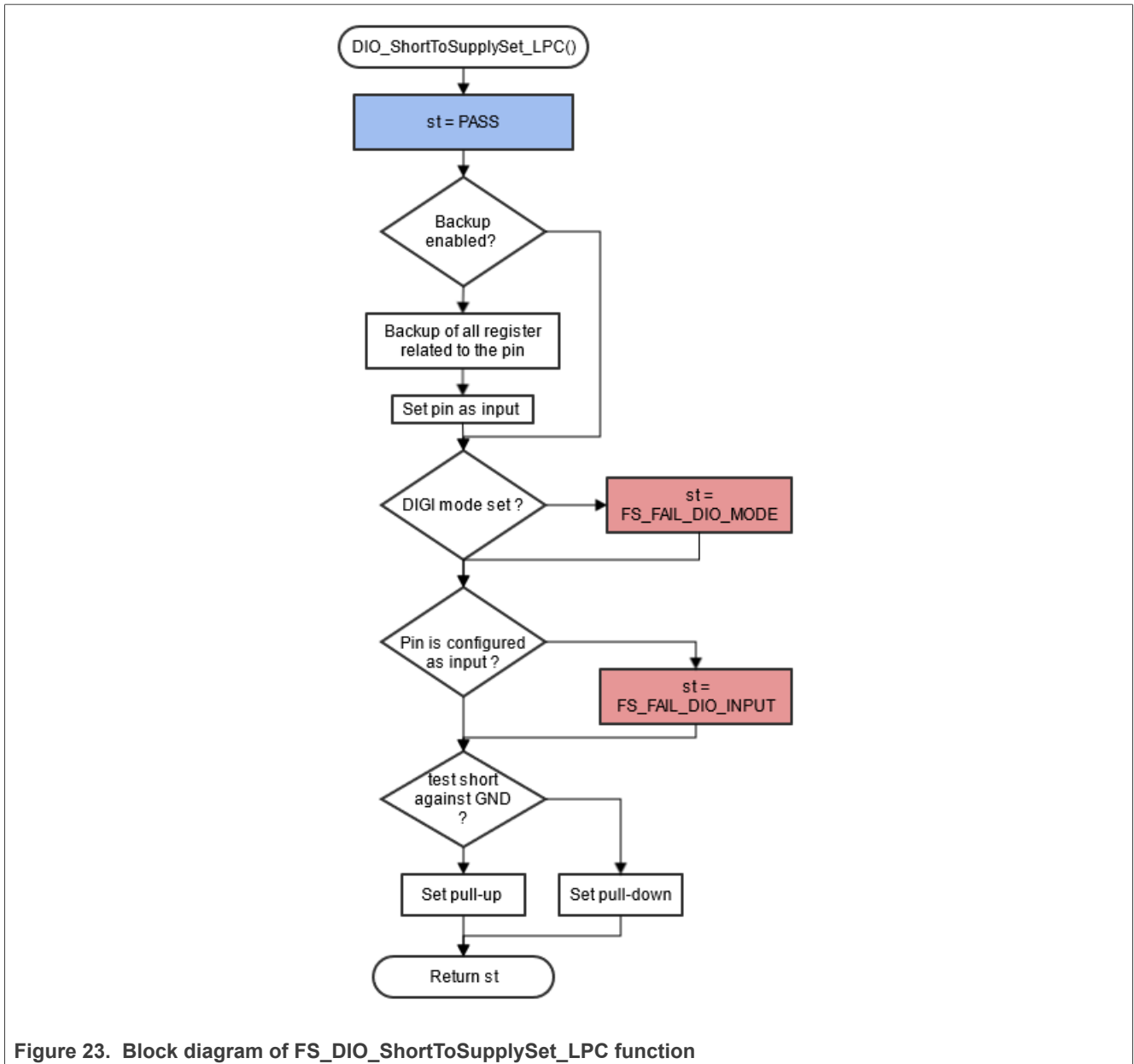


Figure 23. Block diagram of FS\_DIO\_ShortToSupplySet\_LPC function

**Function prototype:**

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

**Function inputs:**

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

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\_INPUT* - The pin is not set as the input.
- *FS\_FAIL\_DIO\_MODE* - The pin does not have the "digimode" set, only for specific LPC devices.

The function always returns the **first** detected error.

#### **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 the other way around).

```
#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.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

#### **4.2.21 FS\_DIO\_InputExt\_RGPIO()**

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 RGPIO 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\_RGPIO()* function is shown in [Figure 24](#). 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.

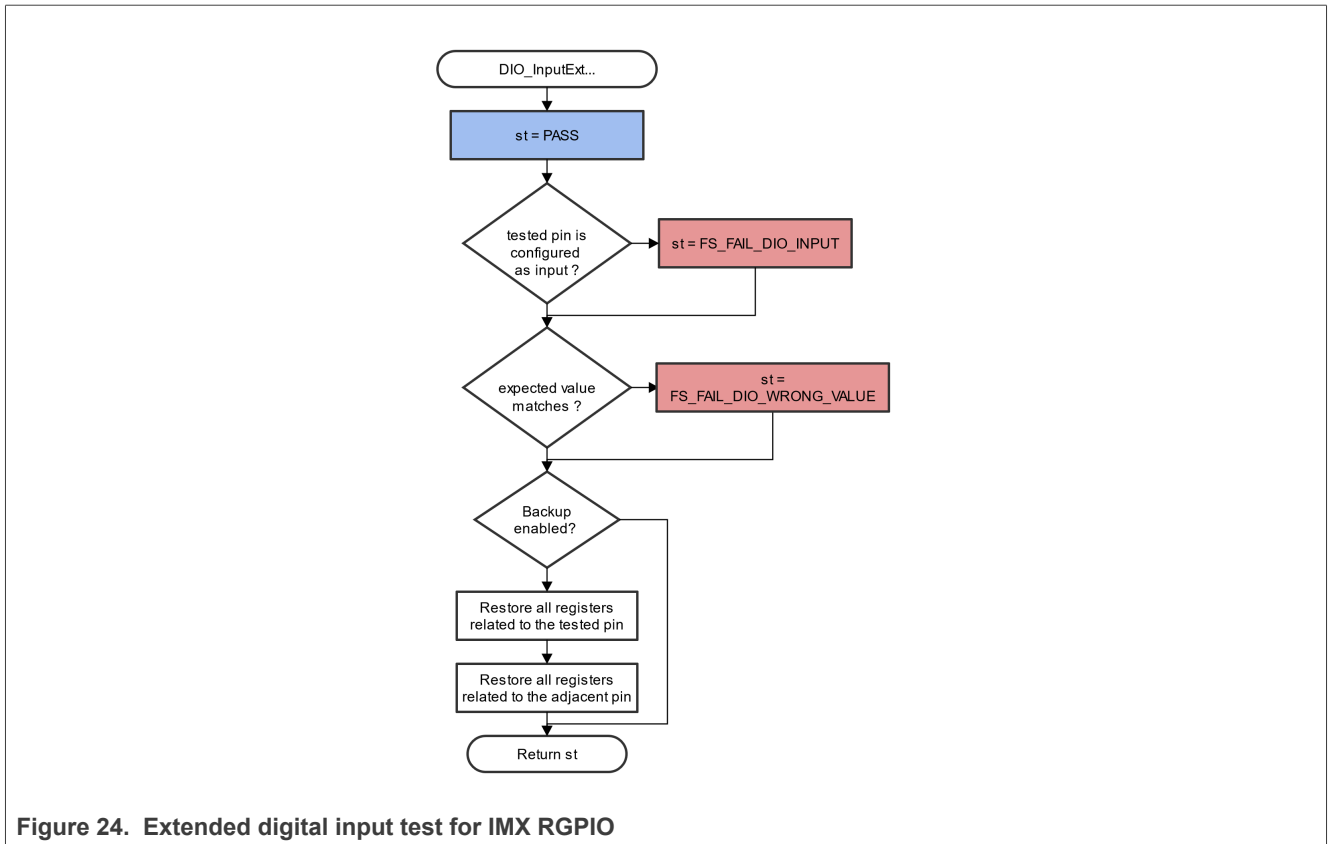


Figure 24. Extended digital input test for IMX RGPIO

**Function prototype:**

*FS\_DIO\_InputExt\_RGPIO(fs\_dio\_test\_rgpio\_t \*pTestedPin, fs\_dio\_test\_rgpio\_t \*pAdjPin, bool\_t testedPinValue, bool\_t backupEnable);*

**Function inputs:**

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

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

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\_INPUT - The pin is not set as the input.
- FS\_FAIL\_DIO\_WRONG\_VALUE - The pin does not have the expected value.

The function always returns the **first** detected error.

**Example of function call:**

```

fs_dio_input_test_result = FS_DIO_InputExt_RGPIO(&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:**

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

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

**4.2.22 FS\_DIO\_ShortToAdjSet\_RGPIO()**

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 25](#). 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\_RGPIO()* function is described in the respective chapter. Specify the tested pin and the adjacent pin for the input test function.

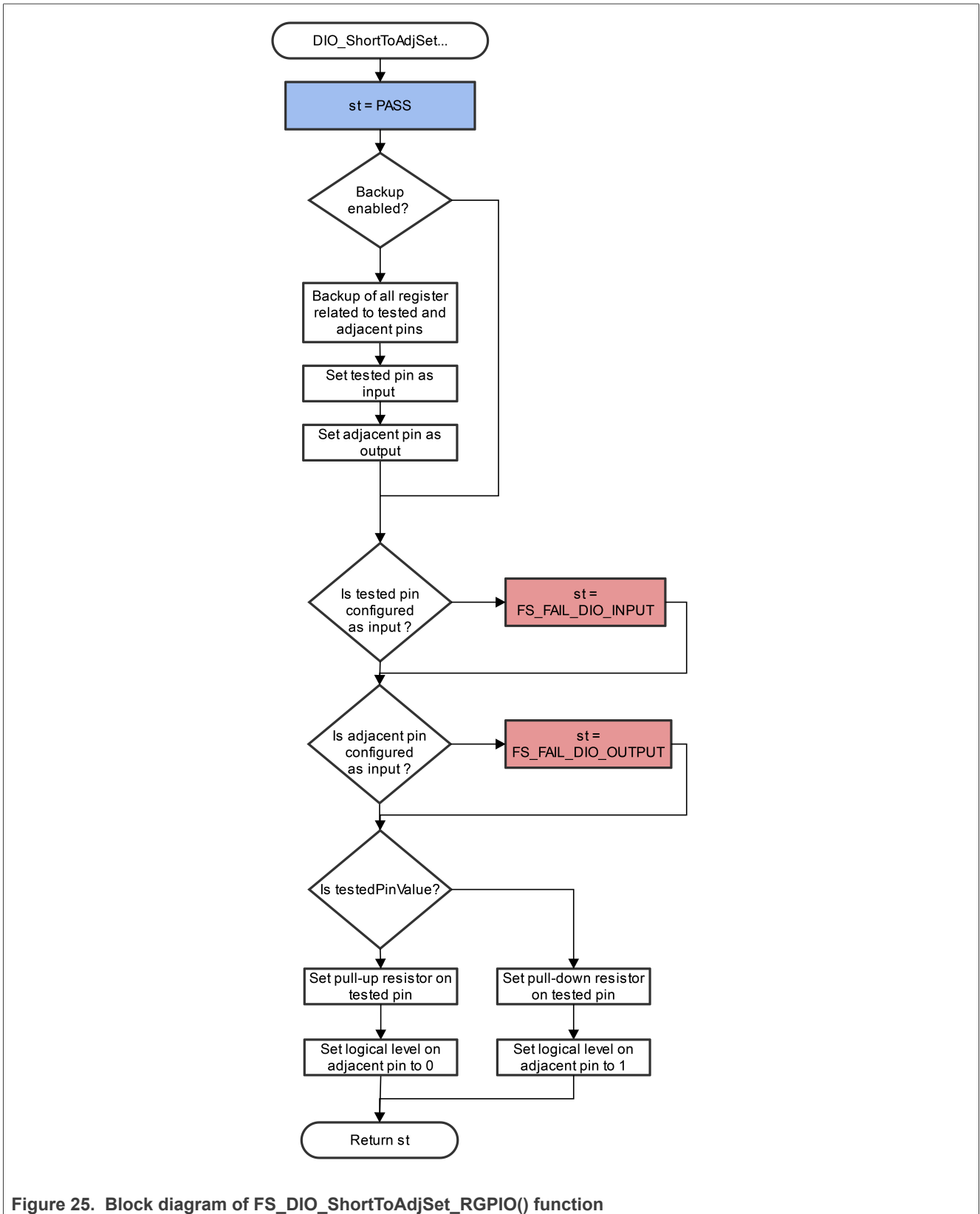


Figure 25. Block diagram of FS\_DIO\_ShortToAdjSet\_RGPIO() function

**Function prototype:**

```
FS_RESULT FS_DIO_ShortToAdjSet_RGPIO(fs_dio_test_rgpio_t *pTestedPin, fs_dio_test_rgpio_t *pAdjPin,
bool_t testedPinValue, bool_t backupEnable);
```

**Function inputs:**

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

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

*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\_INPUT* - The tested pin is not set as the input.
- *FS\_FAIL\_DIO\_OUTPUT* - The adjacent pin is not set as the output.

The function always returns the **first** detected error.

**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_RGPIO(&dio_safety_test_items[0],
  &dio_safety_test_items[1], LOGICAL_ONE, BACKUP_ENABLE);
dio_short_to_adj_test_result = FS_DIO_InputExt_RGPIO(&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 an RGPIO input and the adjacent pin must be configured as an RGPIO 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\_RGPIO()* function is called for these pins.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

**4.2.23 FS\_DIO\_ShortToSupplySet\_RGPIO()**

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 26](#). The second part of the test (result evaluation) is ensured by the *FS\_DIO\_InputExt\_RGPIO()* function described in the respective section. The main purpose of the *FS\_DIO\_InputExt\_RGPIO()* function is to set the pull-up or pull-down resistor connections on the tested pin. It also ensures that the pin is correctly configured and makes a backup of its settings (if needed).

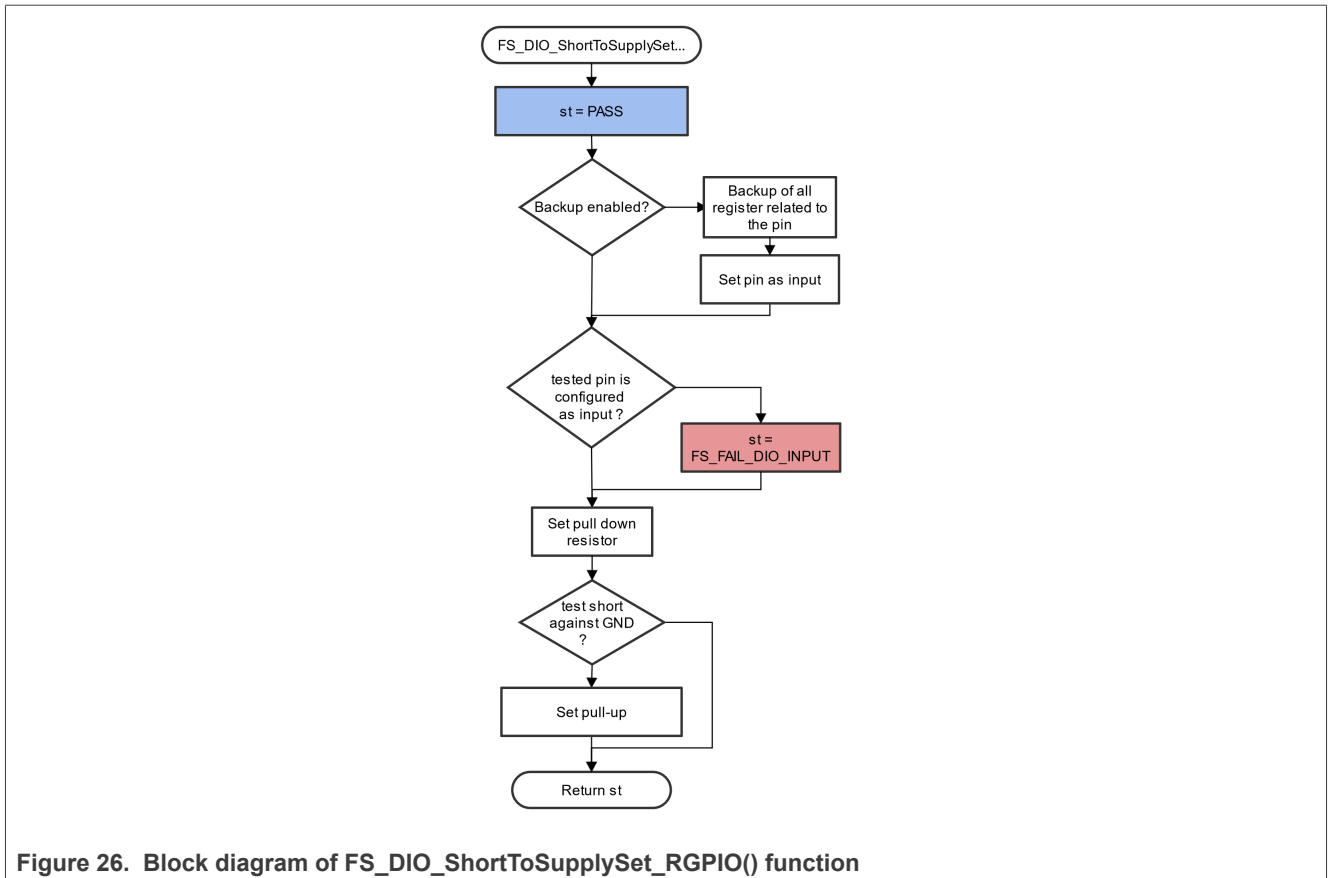


Figure 26. Block diagram of FS\_DIO\_ShortToSupplySet\_RGPIO() function

**Function prototype:**

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

**Function inputs:**

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

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\_INPUT - The pin is not set as the input.

The function always returns the **first** detected error.

**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 the other way around).

```
#define DIO_SHORT_TO_GND_TEST 1
```

```
#define DIO_SHORT_TO_VDD_TEST 0
dio_short_to_vcc_test_result =
  FS_DIO_ShortToSupplySet_RGPIO(&dio_safety_test_items[0], DIO_SHORT_TO_GND_TEST,
  BACKUP_ENABLE);
dio_short_to_vcc_test_result = FS_DIO_InputExt_RGPIO(&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_RGPIO(&dio_safety_test_items[0], DIO_SHORT_TO_VDD_TEST,
  BACKUP_ENABLE);
dio_short_to_vcc_test_result = FS_DIO_InputExt_RGPIO(&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 RGPIO 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\_RGPIO()* function is called for the tested pin.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

### 4.2.24 FS\_DIO\_InputExt\_SIUL2()

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 SIUL2 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\_SIUL2()* function is shown in [Figure 27](#). 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.

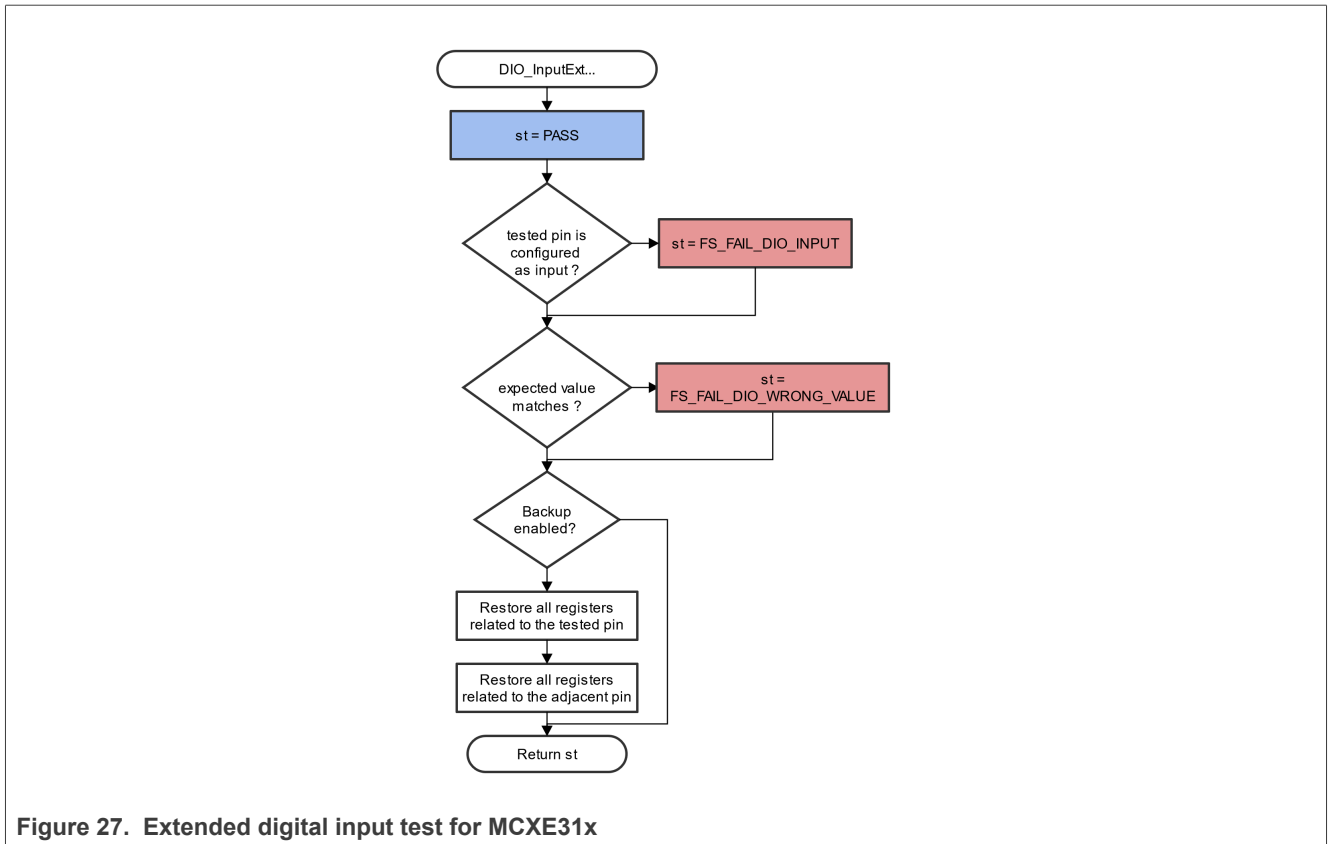


Figure 27. Extended digital input test for MCXE31x

**Function prototype:**

```
FS_RESULT FS_DIO_InputExt_SIUL2(fs_dio_test_siul2_t *pTestedPin, fs_dio_test_siul2_t *pAdjPin, bool_t testedPinValue, bool_t backupEnable);
```

**Function inputs:**

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

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

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\_INPUT - The pin is not set as the input.
- FS\_FAIL\_DIO\_WRONG\_VALUE - The pin does not have the expected value.

The function always returns the **first** detected error.

**Example of function call:**

```
fs_dio_input_test_result = FS_DIO_InputExt_SIUL2(&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:**

Configure the tested pin as an SIUL2 input before calling the function. Even if no adjacent pins are involved in the test, specify the "AdjacentPin" parameter. It is recommended to enter the same input as for "TestedPin".

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

#### 4.2.25 FS\_DIO\_ShortToAdjSet\_SIUL2()

This function ensures the conditions required 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 28](#). 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\_SIUL2()* function is described in the respective section. Specify the tested pin and the adjacent pin for the input test function.

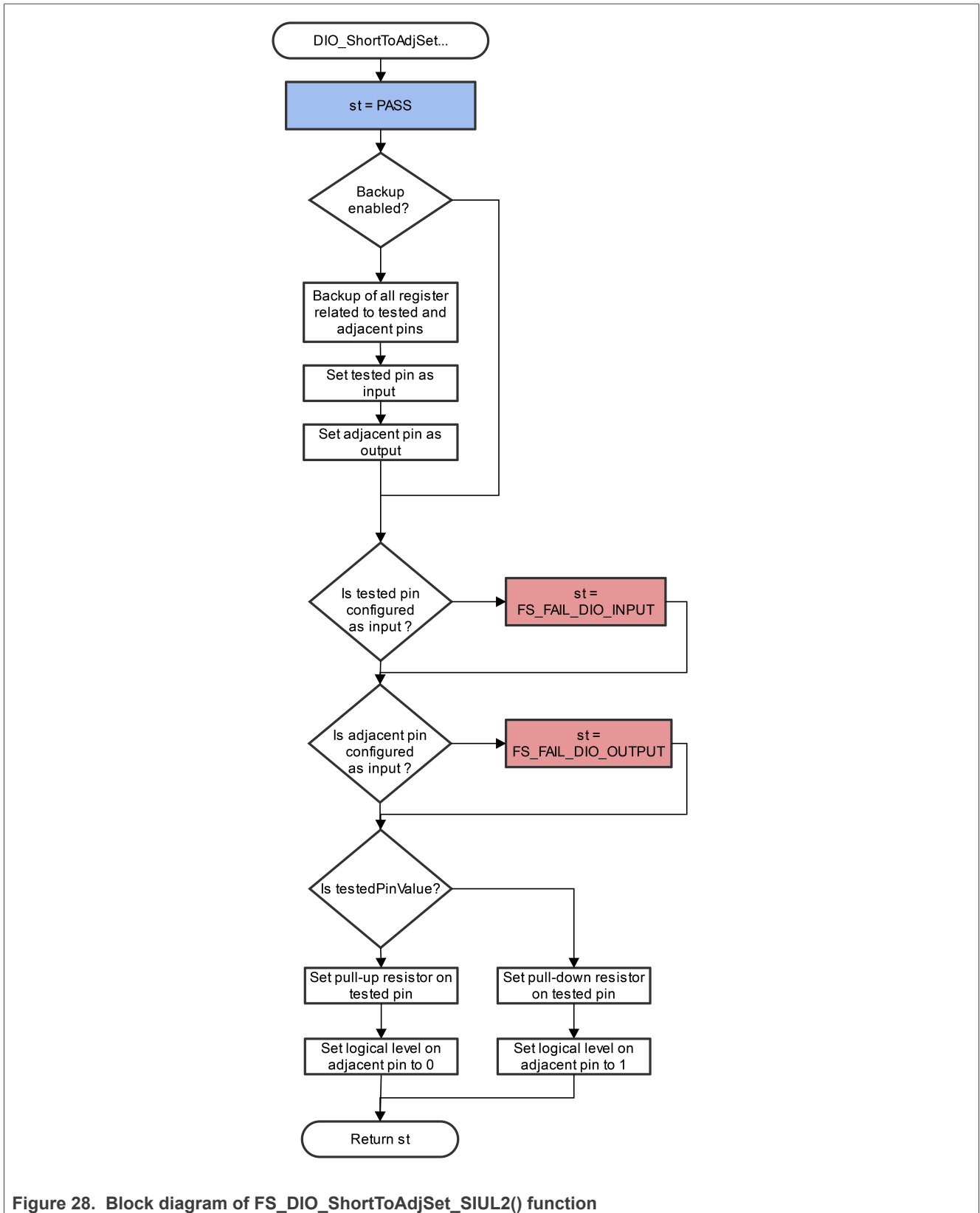


Figure 28. Block diagram of FS\_DIO\_ShortToAdjSet\_SIUL2() function

**Function prototype:**

```
FS_RESULT FS_DIO_ShortToAdjSet_SIUL2(fs_dio_test_siul2_t *pTestedPin, fs_dio_test_siul2_t *pAdjPin,
bool_t testedPinValue, bool_t backupEnable);
```

**Function inputs:**

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

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

*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\_INPUT* - The tested pin is not set as the input.
- *FS\_FAIL\_DIO\_OUTPUT* - The adjacent pin is not set as the output.

The function always returns the **first** detected error.

**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_SIUL2(&dio_safety_test_items[0],
  &dio_safety_test_items[1], LOGICAL_ONE, BACKUP_ENABLE);
dio_short_to_adj_test_result = FS_DIO_InputExt_SIUL2(&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 an SIUL2 input and the adjacent pin must be configured as an SIUL2 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\_SIUL2()* function is called for these pins.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

**4.2.26 FS\_DIO\_ShortToSupplySet\_SIUL2()**

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 29](#). The second part of the test (result evaluation) is ensured by the *FS\_DIO\_InputExt\_SIUL2()* function described in the respective section. The main purpose of the *FS\_DIO\_InputExt\_SIUL2()* function is to set the pull-up or pull-down resistor connections on the tested pin. It also ensures that the pin is correctly configured and makes a backup of its settings (if needed).

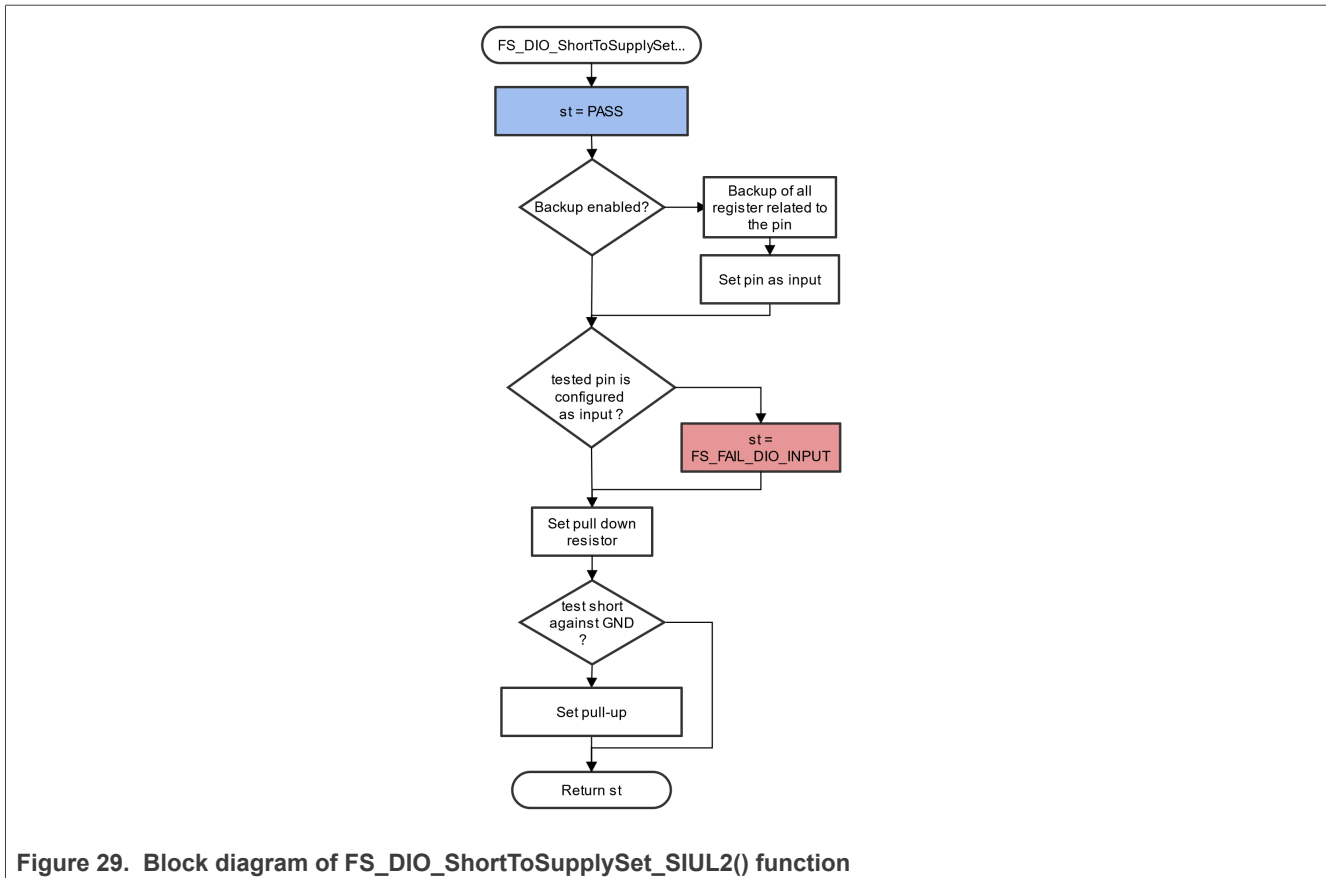


Figure 29. Block diagram of FS\_DIO\_ShortToSupplySet\_SIUL2() function

**Function prototype:**

*FS\_RESULT* FS\_DIO\_ShortToSupplySet\_SIUL2(*fs\_dio\_test\_siul2\_t* \*pTestedPin, *bool\_t* shortToVoltage, *bool\_t* backupEnable);

**Function inputs:**

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

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\_INPUT - The pin is not set as the input.

The function always returns the **first** detected error.

**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 the other way around).

```
#define DIO_SHORT_TO_GND_TEST 1
```

```
#define DIO_SHORT_TO_VDD_TEST 0
dio_short_to_vcc_test_result =
  FS_DIO_ShortToSupplySet_SIUL2(&dio_safety_test_items[0], DIO_SHORT_TO_GND_TEST,
  BACKUP_ENABLE);
dio_short_to_vcc_test_result = FS_DIO_InputExt_SIUL2(&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_SIUL2(&dio_safety_test_items[0], DIO_SHORT_TO_VDD_TEST,
  BACKUP_ENABLE);
dio_short_to_vcc_test_result = FS_DIO_InputExt_SIUL2(&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 SIUL2 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_SIUL2()` function is called for the tested pin.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

## 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 19](#).

Table 19. 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 parts of test functions for the flash memory are placed in *iec60730b\_cm4\_cm7\_flash.S* and they are written as assembler functions. The header file with the definitions and function prototypes is *iec60730b\_cm4\_cm7\_flash.h*. The rest of functions is placed in *iec60730b\_invariable\_memory.c* with the corresponding header file and they are written in the C language. The test functions use also the following header files: *iec60730b.h*, *asm\_mac\_common.h*, and *iec60730b\_types.h*. They are the common header files for the safety library.

The following functions are implemented in *iec60730b\_invariable\_memory.c*:

- *FS\_FLASH\_C\_HW16\_K()* / *FS\_FLASH\_C\_HW16\_K()* / *FS\_FLASH\_C\_HW32\_K()* / *FS\_CM4\_CM7\_FLASH\_HW32\_DCP()*

The following functions are implemented in *iec60730b\_cm4\_cm7\_flash.S*:

- *FS\_CM4\_CM7\_FLASH\_HW16()*
- *FS\_CM4\_CM7\_FLASH\_SW16()*
- *FS\_CM4\_CM7\_FLASH\_SW32()*

The hardware (\*\_HW) functions use 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`.

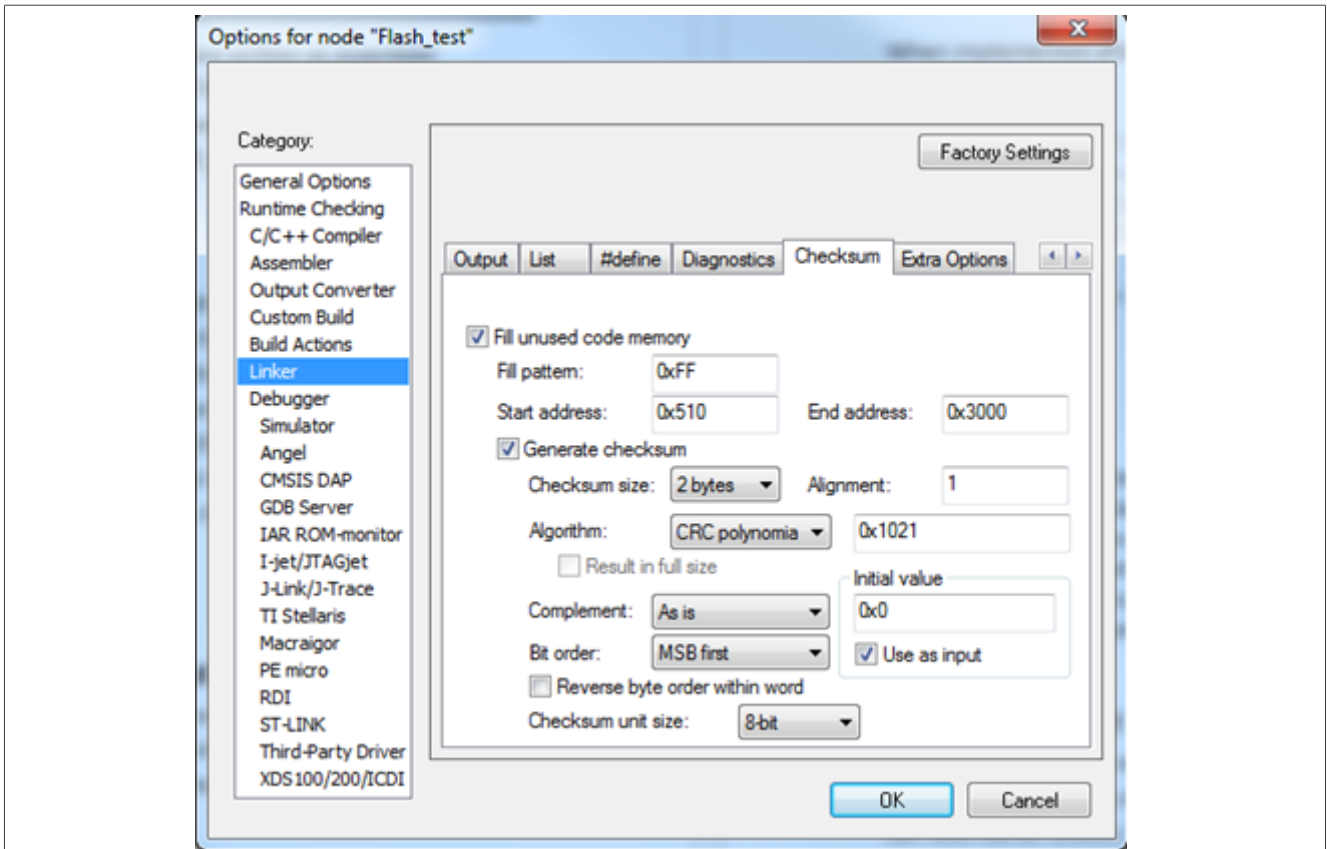


Figure 30. Checksum settings for linker

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_CM4_CM7_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_CM4_CM7_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\_FLASH\_C\_HW32\_K()

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

**Function prototype:**

```
FS_RESULT FS_FLASH_C_HW32_K(uint32_t startAddress, uint32_t size, FS_CRC_Type * moduleAddress,
uint32_t * crcVal);
```

**Function inputs:**

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

*size* - The size of the tested memory. It must be divisible by 4.

*moduleAddress* - The address of the CRC module.

*crcVal* - The pointer to the variable for the result and 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. The CRC module must be initialized to CRC-32 - normal 0x04C11DB7.

**Function output:**

*FS\_RESULT*

- FS\_FAIL\_FLASH\_NULL\_POINTER\_C - The *moduleAddress* or *crcVal* input parameters are NULL.
- FS\_FAIL\_FLASH\_MODULO\_C - The parameter *size* is not aligned to 4 bytes.
- FS\_FAIL\_FLASH\_SIZE\_C - The *size* input parameter is 0.

**Function performance:**

The function parameter was measured on FRDM-MCXXE31x with a clock frequency of 160 MHz.

The function size is 110 B.

The function duration depends on the defined block size. Several examples are shown in [Table 20](#):

Table 20. Duration of FS\_FLASH\_C\_HW32\_K() depending on tested block size

Block size (in bytes)	Execution time (approximately)
0x10	0.75 μs
0x20	0.95 μs
0x100	3.76 μs

**Calling restrictions:**

The CRC module must be correctly configured to calculate a normal 0x04C11DB7 CRC before calling this function. The function cannot be interrupted by a function that changes the content or setup of the hardware CRC module.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

### 5.2.5 FS\_FLASH\_C\_HW16\_K()

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

**Function prototype:**

```
FS_RESULT FS_FLASH_C_HW16_K(uint32_t startAddress, uint32_t size, FS_CRC_Type * moduleAddress,
uint16_t * crcVal);
```

**Function inputs:**

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

*size* - The size of the tested memory. It must be divisible by 4.

*moduleAddress* - The address of the CRC module.

*crcVal* - Pointer to the variable for the result and 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 (The CRC module must be initialize to: CRC-16-CCITT - normal 0x1021).

**Function output:**

*FS\_RESULT*

- FS\_FAIL\_FLASH\_NULL\_POINTER\_C - The *moduleAddress* or *crcVal* input parameters are NULL.
- FS\_FAIL\_FLASH\_MODULO\_C - The parameter *size* is not aligned to 4 bytes.
- FS\_FAIL\_FLASH\_SIZE\_C - The *size* input parameter is 0.

**Function performance:**

The function parameter was measured on LPC55S36 with a clock frequency of 150 MHz.

The function size is 96 B.

The function duration depends on the defined block size. Several examples are shown in [Table 21](#):

Table 21. Duration of FS\_FLASH\_C\_HW16\_K() depending on tested block size

Block size (in bytes)	Execution time (approximately)
0x10	1,6 µs
0x20	1,92 µs
0x100	6,68 µs

**Calling restrictions:**

The CRC module must be correctly configured to calculate normal 0x1021 CRC before calling this function. The function cannot be interrupted by a function that changes the content or setup of the hardware CRC module.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

### 5.2.6 FS\_FLASH\_C\_HW16\_L()

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

**Function prototype:**

```
FS_RESULT FS_FLASH_C_HW16_L(uint32_t startAddress, uint32_t size, FS_CRC_L_Type * 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* - Pointer to the variable for the result and 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 (CRC-16-CCITT - normal 0x1021).

**Function output:**

*FS\_RESULT*

- *FS\_FAIL\_FLASH\_NULL\_POINTER\_C* - The *moduleAddress* or *crcVal* input parameters are NULL.
- *FS\_FAIL\_FLASH\_SIZE\_C* - The *size* input parameter is 0.

**Function performance:**

The function parameter was measured on LPC54S018M with a clock frequency of 96 MHz.

The function size is 66 B.

The function duration depends on the defined block size. Several examples are shown in [Table 22](#):

**Table 22. Duration of FS\_FLASH\_C\_HW16\_L() depending on tested block size**

Block size (in bytes)	Execution time (approximately)
0x10	14,36 µs
0x20	18,04 µs
0x100	44,12 µs

**Calling restrictions:**

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

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

**5.2.7 FS\_CM4\_CM7\_FLASH\_HW32\_DCP()**

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

**Function prototype:**

*void FS\_CM4\_CM7\_FLASH\_HW32\_DCP(uint32\_t startAddress, uint32\_t size, uint32\_t moduleAddress, uint32\_t crcVal, fs\_flash\_dcp\_channels\_t channel, fs\_flash\_dcp\_state\_t \*psDCPState, uint32\_t tag);*

**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 starting 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).

*channel* - The DCP channel used for calculation.

*psDCPState* - The state and result structures of each DCP channel.

*tag* - Differentiates the calculation on the same channel.

**Function output:**

*uint32\_t* - The 32-bit CRC value of the memory range (CRC-32/MPEG-2 - 0x04C11DB7).

**Example of function call:**

```

/* CRC calculation of SAFETY_FLASH_BLOCK (channel should be available after
reset) */
do
{
    FS_CM4_CM7_FLASH_HW32_DCP(psFlashConfig->startAddress, psFlashConfig-
>size, (uint32_t)FLASH_USED_DCP,
                                psFlashConfig->startConditionSeed,
    g_dcpSafetyChannel, psFlashDCPState,
                                FLASH_DCP_TAG);

    /* Check error. */
    if (psFlashDCPState->CH3State == FS_FAIL_FLASH_DCP)
    {
        psSafetyCommon->safetyErrors |= FLASH_TEST_ERROR;
        SafetyErrorHandling(psSafetyCommon);
    }
} while (psFlashDCPState->CH3State == FS_FLASH_DCP_BUSY);

/* Store the result */
psSafetyCommon->FLASH_test_result = psFlashDCPState->CH3Result;

/* Check if result equals precomputed CRC value */
if (psSafetyCommon->FLASH_test_result != psFlashConfig->checksum)
{
    psSafetyCommon->safetyErrors |= FLASH_TEST_ERROR;
    SafetyErrorHandling(psSafetyCommon);
}

```

**Function performance:**

The function size is 448 bytes.<sup>4</sup>

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

**Table 23. Duration of FS\_CM4\_CM7\_FLASH\_HW32\_DCP() in dependence of tested block size**

Block size (bytes)	Clock cycles	Execution time (approximately)
0x10	57	2.375 µs
0x20	57	2.375 µs
0x50	67	2.791 µs
0x500	261	10.875 µs

**Calling restrictions:**

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

Multiple calculations with different tag number on the same channel are supported, but they must be placed in the same execution block - for example, channel 0 calculations in the SysTick ISR and channel 1 calculations in the while loop.

The calculated data block must be aligned to 4 bytes.

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

### 5.2.8 FS\_CM4\_CM7\_FLASH\_HW16()

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

**Function prototype:**

```
uint16_t FS_CM4_CM7_FLASH_HW16(uint32_t startAddress, uint32_t size, uint32_t moduleAddress, uint16_t crcVal);
```

**Function inputs:**

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

*size* - The size of the tested memory.

*moduleAddress* - The address of the CRC module.

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

**Function output:**

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

**Function performance:**

Function size is 44 bytes.<sup>1</sup>

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

**Table 24. Duration of FS\_CM4\_CM7\_FLASH\_HW16() in dependence of tested block size**

Block size (Bytes)	Clock cycles	Execution time (approximately)
0x10	208	2.6 µs
0x20	343	4.2 µs
0x50	745	9.3 µs

**Calling restrictions:**

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

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

### 5.2.9 FS\_CM4\_CM7\_FLASH\_SW16()

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

**Function prototype:**

```
uint16_t FS_CM4_CM7_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* - Has no effect. Just because of compatibility with HW 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:**

Function size is 54 bytes.<sup>1</sup>

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

Table 25. Duration of FS\_CM4\_CM7\_FLASH\_SW16() in dependence of tested block size

Block size (Bytes)	Clock cycles	Execution time (approximately)
0x10	1934	24.175 µs
0x20	3936	49.2 µs
0x50	9758	121.975 µs

**Calling restrictions:**

None.

**5.2.10 FS\_CM4\_CM7\_FLASH\_SW32()**

This function generates the 32-bit CRC value using software.

**Function prototype:**

*uint32\_t FS\_CM4\_CM7\_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* - Has no effect. Just because of compatibility with HW 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:**

Function size is 78 bytes.<sup>1</sup>

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

Table 26. Duration of FS\_CM4\_CM7\_FLASH\_SW32() in dependence of tested block size

Block size (Bytes)	Clock cycles	Execution time (approximately)
0x10	1795	22.438 µs
0x20	3631	45.388 µs
0x50	9030	112.875 µs

**Calling restrictions:**

None.

## 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:

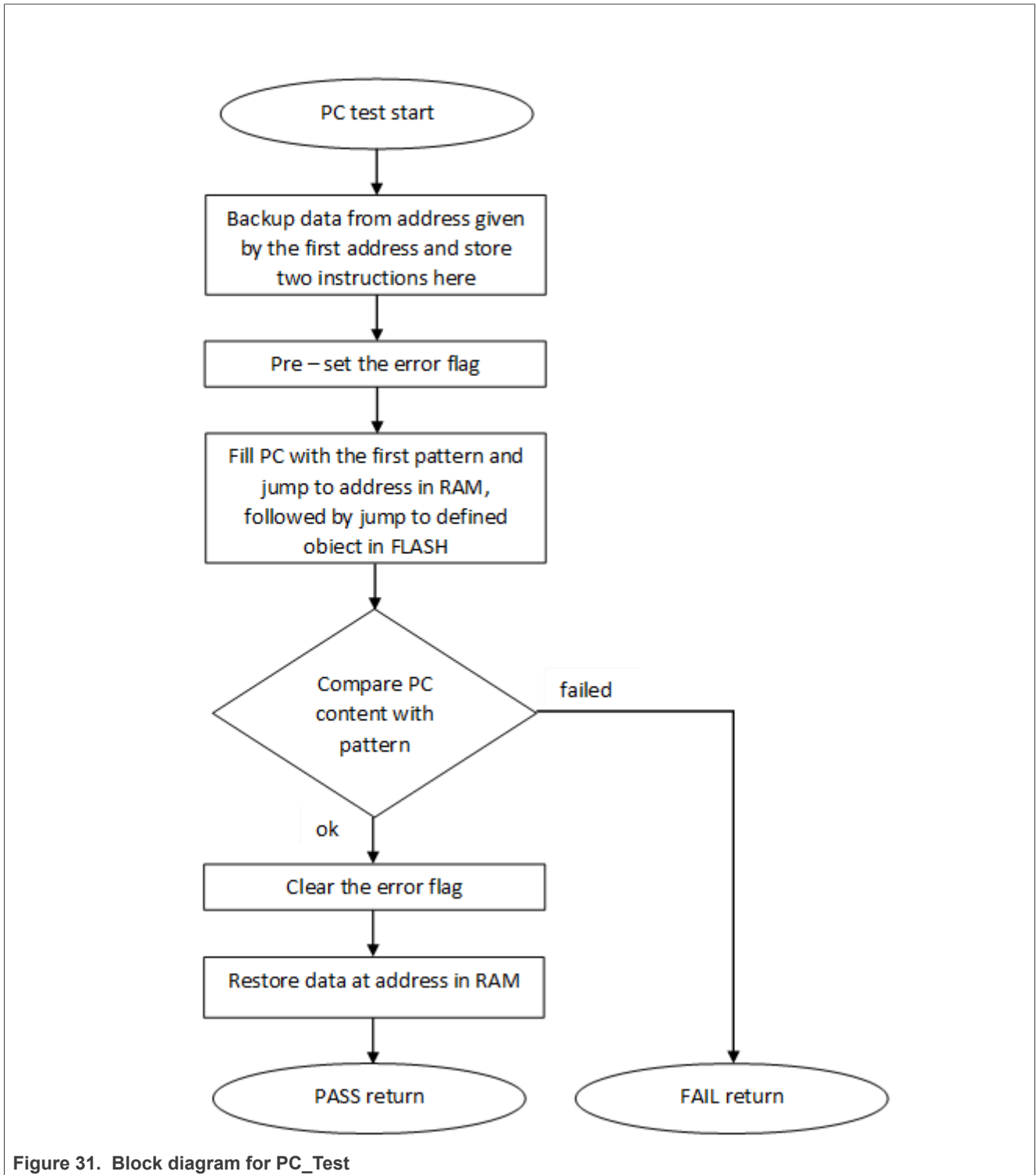


Figure 31. Block diagram for PC\_Test

### 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 27. 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\_cm4\_cm7\_pc.S* file and they are written as assembler functions. The header file with the test patterns and the function prototypes is *iec60730b\_cm4\_cm7\_pc.h*. The *iec60730b.h*, *asm\_mac\_common.h*, and *iec60730b\_types.h* are the common header files for the safety library. For the second test type, the *iec60730b\_cm4\_cm7\_pc\_object.S* file must be placed to an appropriate address in the flash memory.

### Implementation example of the PC test:

The only function that is handled in the application is as follows:

```
FS_CM4_CM7_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\_cm4\_cm7\_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_CM4_CM7_PC_Test(0x20000013, FS_PC_object, PC_TEST_FLAG);
if (FS_FAIL_PC == fs_pc_test_result)
    SafetyError();
```

### 6.2.1 FS\_CM4\_CM7\_PC\_Test()

The program counter register is tested according to [Figure 31](#).

#### Function prototype:

```
FS_RESULT FS_CM4_CM7_PC_Test(uint32_t pattern1, tFcn_pc pObjectFunction, uint32_t *flag);
```

#### Function inputs:

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

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

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

#### Function output:

```
typedef uint32_t FS_RESULT;
```

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

#### Function performance:

The function takes approximately 92 cycles (1.15  $\mu$ s).<sup>1</sup>

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

**Calling restrictions:**

This function cannot be interrupted.

### 6.2.2 FS\_PC\_Object()

This function is internally used by the *FS\_CM4\_CM7\_PC\_Test()* function. The function is used to perform the program counter test. It should be called only by the *FS\_CM4\_CM7\_PC\_Test()* function. It should be placed at a reliable address (by editing the linker file).

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

```
define symbol __PC_test_start__ = 0x00008FE0;
define symbol __PC_test_end__ = 0x00008FFF;
define region PC_region = mem:[from __PC_test_start__ to __PC_test_end__];
define block PC_TEST { section .text object iec60730b_cm4_cm7_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\_CM4\_CM7\_PC\_Test()* function. It's size is 16 bytes.

**Calling restrictions:**

This function is used to perform the PC test, it should be called only by the *FS\_CM4\_CM7\_PC\_Test()* function.

## 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:

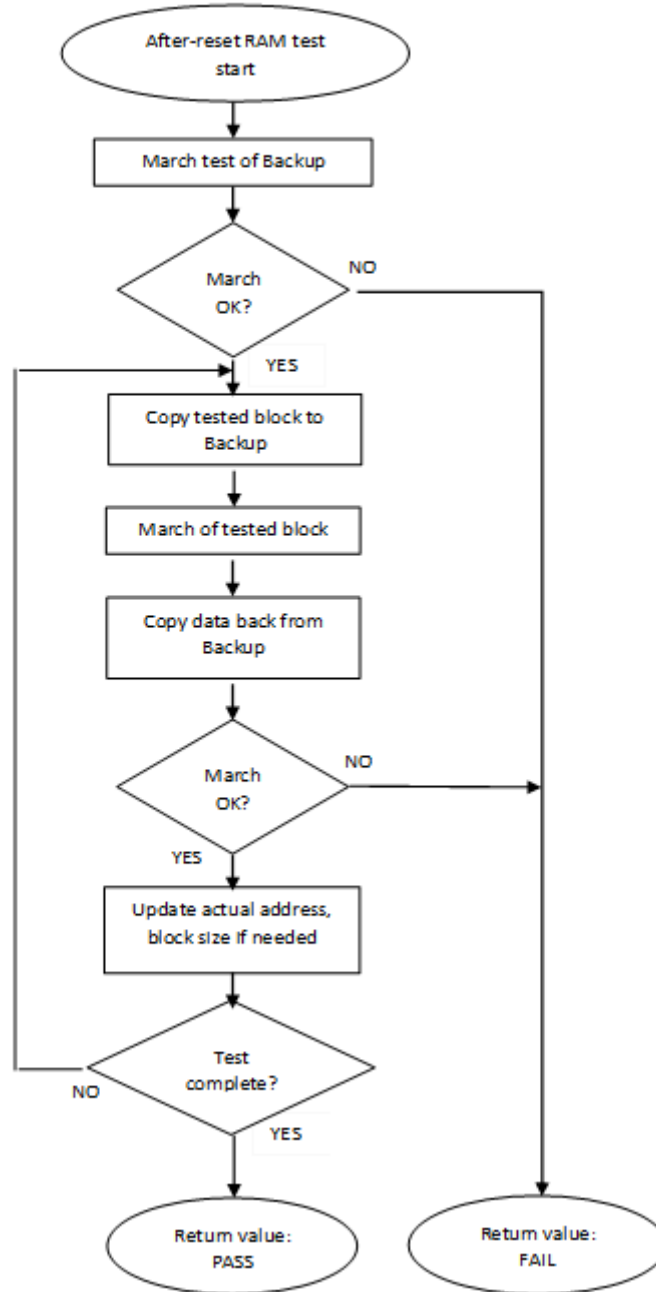


Figure 32. Block diagram for after-reset test of RAM

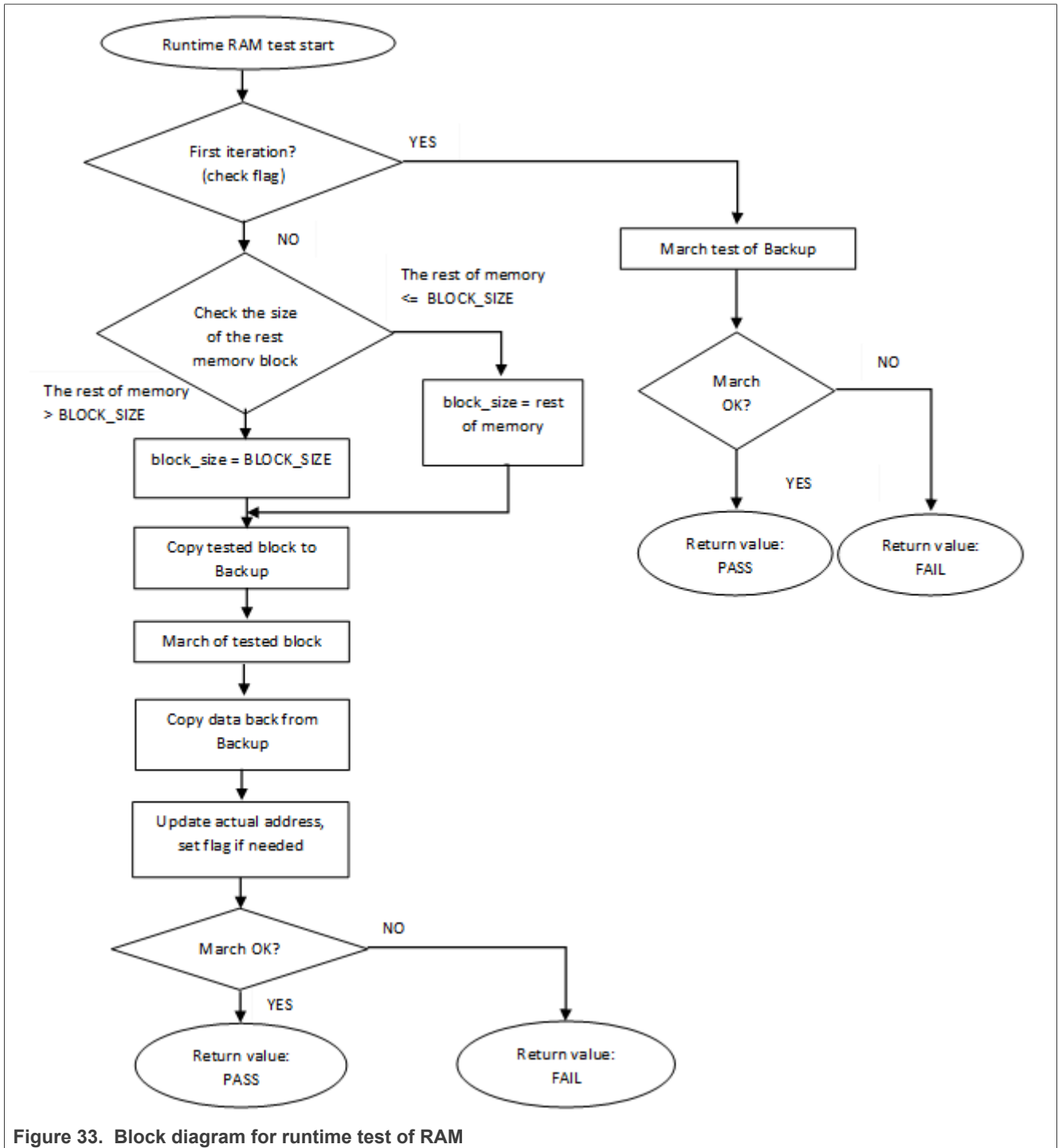


Figure 33. Block diagram for runtime test of RAM

### 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 28. 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\_cm4\_cm7\_ram.S* file and they are written as assembler functions. The header file with return values and function prototypes is *iec60730b\_cm4\_cm7\_ram.h*. The *iec60730b.h*, *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\_CM4\_CM7\_RAM\_AfterReset()*
- *FS\_CM4\_CM7\_RAM\_Runtime()*
- *FS\_CM4\_CM7\_RAM\_CopyToBackup()*
- *FS\_CM4\_CM7\_RAM\_CopyFromBackup()*
- *FS\_CM4\_CM7\_RAM\_SegmentMarchC()*
- *FS\_CM4\_CM7\_RAM\_SegmentMarchX()*

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

### 7.2.1 FS\_CM4\_CM7\_RAM\_AfterReset()

The after-reset test is done by the *FS\_CM0\_RAM\_AfterReset()* function. This function is called once after reset, when the execution time is not critical. Reserve free memory space for the backup area. The block size parameter cannot be larger than the size of the backup area. The function firstly checks the backup area. Then the loop begins. Blocks of memory are copied to the backup area and their locations are checked with the respective March test. The data is copied back to the original memory area and the actual address with the block size is updated. This is repeated until the last block of memory is tested. If a DC fault is detected, the function returns a fail pattern. The block diagram is shown in [.Figure 32](#)

Here is an example of the function call:

```
#include "iec60730b.h"
if (FS_FAIL_RAM == FS_CM4_CM7_RAM_AfterReset(startAddress, endAddress,
    blockSize, backupAddress, FS_CM4_CM7_RAM_SegmentMarchC))
    SafetyError();
```

#### Function prototype:

*FS\_RESULT FS\_CM4\_CM7\_RAM\_AfterReset(uint32\_t startAddress, uint32\_t endAddress, uint32\_t blockSize, uint32\_t backupAddress, tFcn pMarchType);*

#### Function inputs:

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

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

*blockSize* - The tested block size.

*backupAddress* - The address of the backup area.

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

**Function output:**

`typedef uint32_t FS_RESULT;`

- `FS_PASS`
- `FS_FAIL_RAM`

**Function performance:**

The function size is 98 B.<sup>1</sup>

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

**Table 29. FS\_CM4\_CM7\_RAM\_AfterReset duration**

Memory size(Bytes)	Block size(Bytes)	Cycles - March X	Cycles – March C
0x100	0x20	3831	5238
0x100	0x40	3842	5360
0x100	0x80	4095	5882
0x200	0x20	7310	9926
0x200	0x40	6839	9534
0x200	0x80	7346	10298
0x400	0x20	13791	18838
0x400	0x40	13574	18638
0x400	0x80	13095	18382

**Calling restrictions:**

This function is used once after the MCU reset, when the execution time is not critical.

It cannot be interrupted.

The backup area must be of at least the same size as the tested block size defined by the "block\_size" parameter.

**7.2.2 FS\_CM4\_CM7\_RAM\_Runtime()**

The runtime test is done by the `FS_CM4_CM7_RAM_Runtime()` function. Reserve free memory space for the backup area. The block size parameter cannot be larger than the size of the backup area. During the first call, the function checks the backup area. After the call, blocks of memory are processed in a sequence. They are copied to the backup area and their locations are checked using the respective March test. The data is copied back to the original memory area and the actual address and the block size are updated. This is repeated until the last block of memory is tested. If a DC fault is detected, the function returns a fail pattern. The block diagram is shown above. The example of a function call is as follows:

```
#include "iec60730_b.h"
if(FS_RAM_FAIL == FS_RESULT FS_CM4_CM7_RAM_Runtime(startAddress, endAddress,
    &actualAddress, blockSize, backupAddress, FS_CM4_CM7_RAM_SegmentMarchX)
    SafetyError();
```

**Function prototype:**

`FS_RESULT FS_CM4_CM7_RAM_Runtime(uint32_t startAddress, uint32_t endAddress, uint32_t *pActualAddress, uint32_t blockSize, uint32_t backupAddress, tFcn pMarchType);`

**Function inputs:**

- startAddress* - The first address of the tested RAM area.
- 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.<sup>1</sup>  
 The execution time depends on the block size and it is different for the March C and March X methods.<sup>1</sup>

Table 30. FS\_CM4\_CM7\_RAM\_Runtime duration

| Block size(Bytes) | Cycles - March X | Cycles - March C |
|-------------------|------------------|------------------|
| 0x4               | 202              | 187              |
| 0x8               | 250              | 298              |
| 0x20              | 532              | 688              |
| 0x40              | 908              | 1208             |

**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\_CM4\_CM7\_RAM\_CopyFromBackup()**

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

**Function prototype:**

```
void FS_CM4_CM7_RAM_CopyFromBackup(uint32_t startAddress, uint32_t blockSize, uint32_t backupAddress);
```

**Function inputs:**

- startAddress* - The first address of the destination.
- blockSize* - The size of the memory block.
- backupAddress* - The address of the backup area.

**Function output:**

*void*

**Function performance:**

The function size is 16 B.<sup>1</sup>

#### 7.2.4 FS\_CM4\_CM7\_RAM\_CopyToBackup()

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

**Function prototype:**

```
void FS_CM4_CM7_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 16 B.<sup>1</sup>

#### 7.2.5 FS\_CM4\_CM7\_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_CM4_CM7_RAM_SegmentMarchC(uint32_t startAddress, uint32_t blockSize);
```

**Function inputs:**

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

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

**Function output:**

```
typedef uint32_t FS_RESULT;
```

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

**Function performance:**

The function size is 118 B.<sup>1</sup>

#### 7.2.6 FS\_CM4\_CM7\_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_CM4_CM7_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 98 B.<sup>1</sup>

## 8 CPU register test

The CPU register test procedure tests all of the CM4/CM7 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:

- SP main
- SP process

Special registers:

- APSR
- CONTROL
- PRIMASK
- FAULTMASK
- BASEPRI

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.

For the PRIMASK and CONTROL and FAULTMASK and BASEPRI 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:

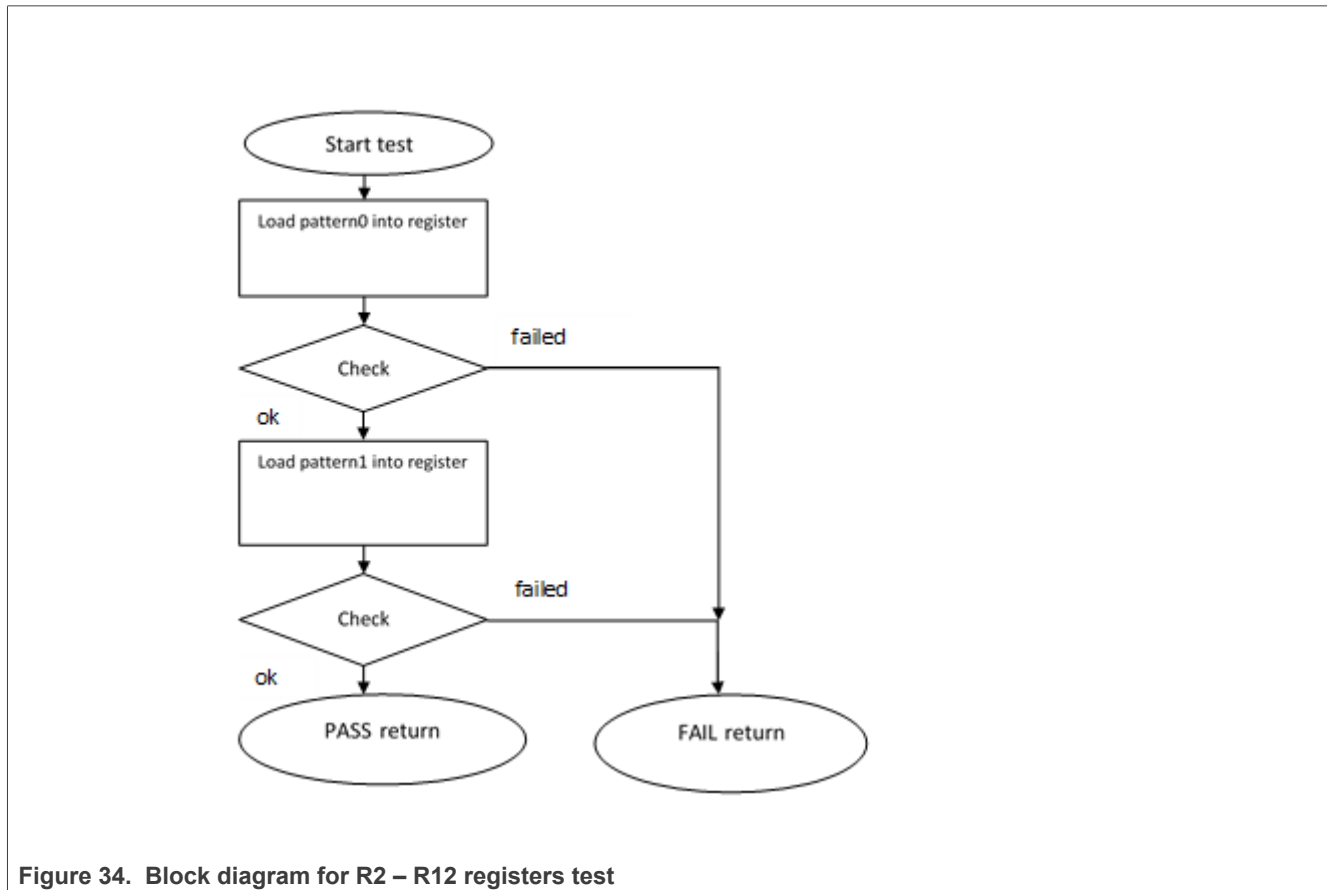


Figure 34. Block diagram for R2 – R12 registers test

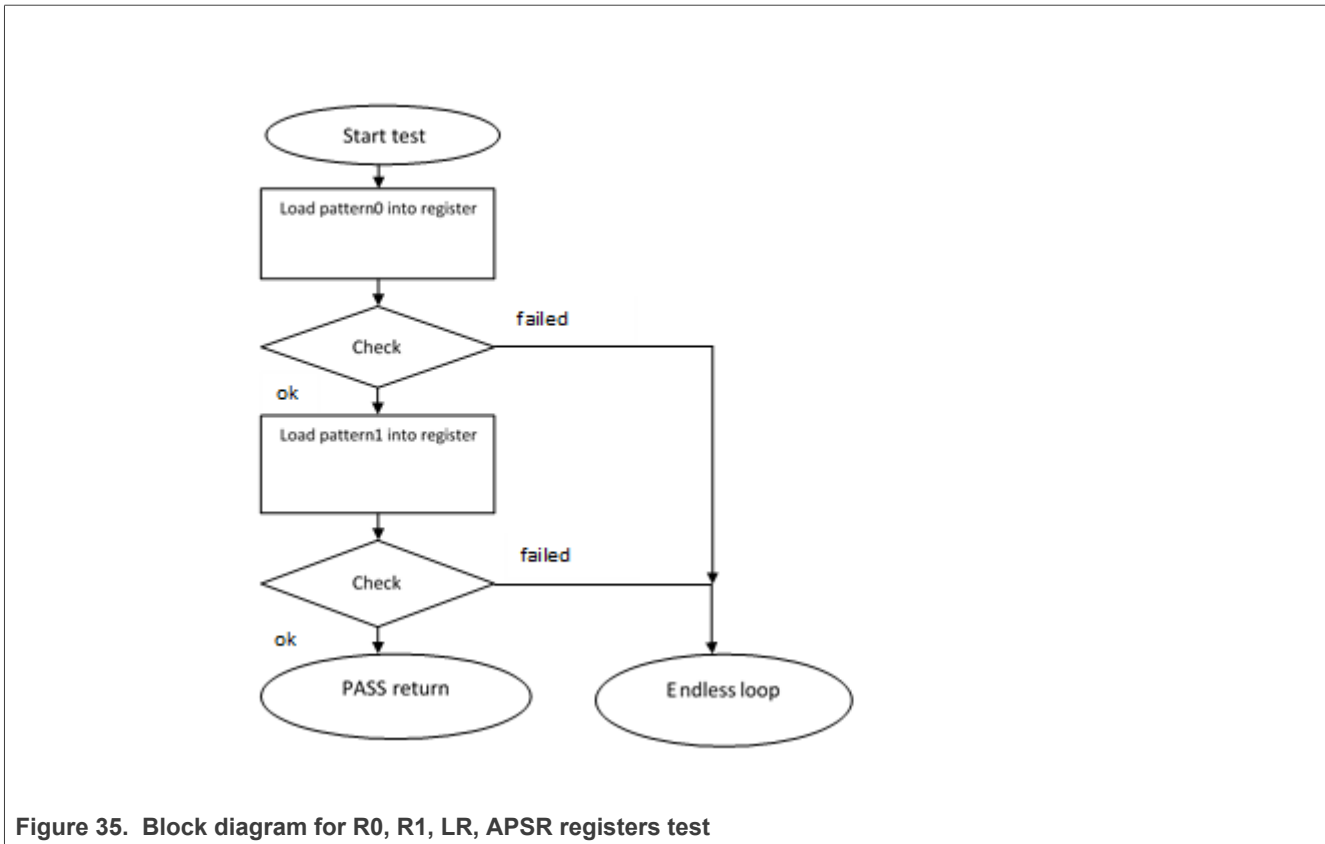


Figure 35. Block diagram for R0, R1, LR, APSR registers test

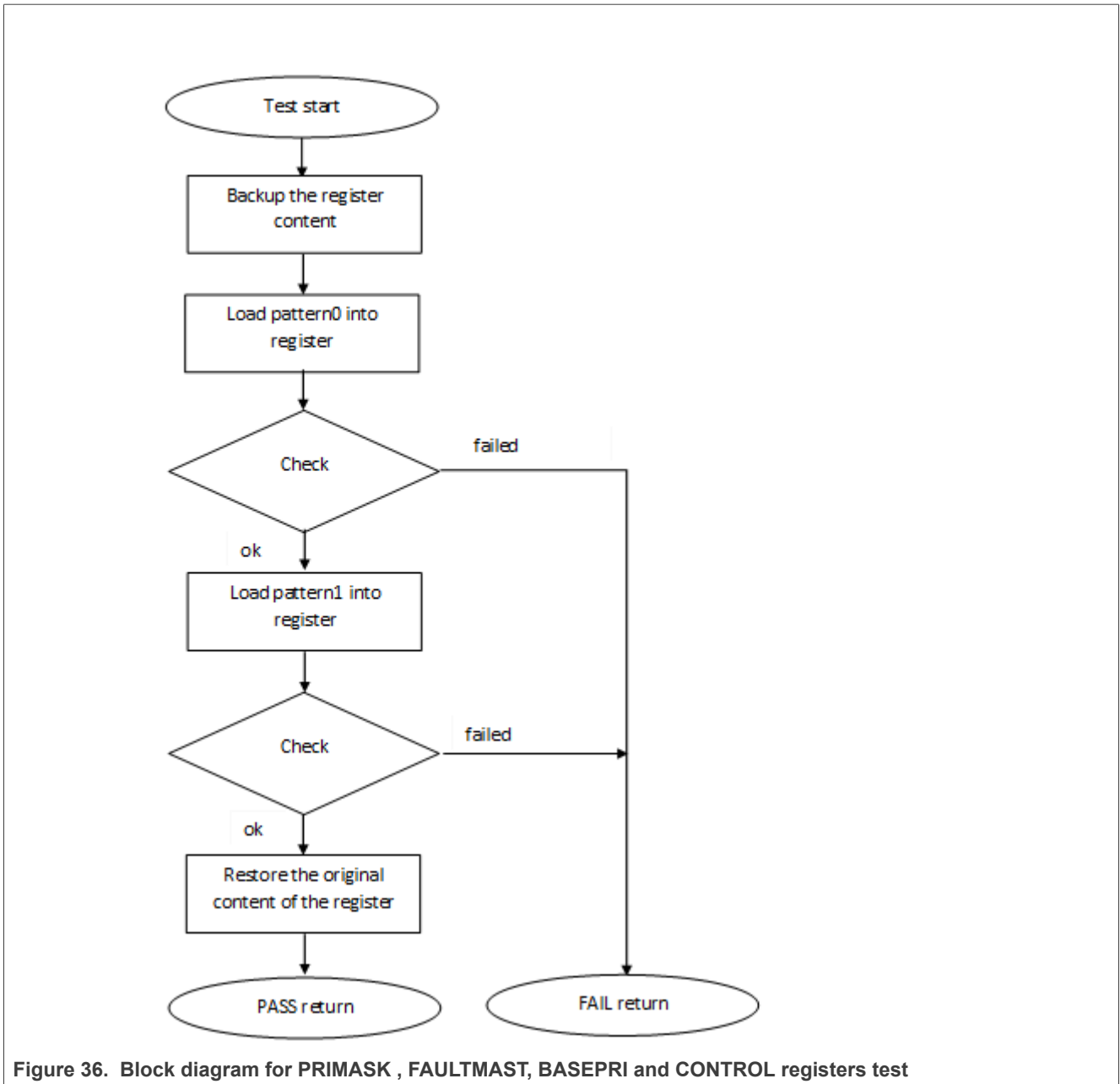


Figure 36. Block diagram for PRIMASK , FAULTMAST, BASEPRI and CONTROL registers test

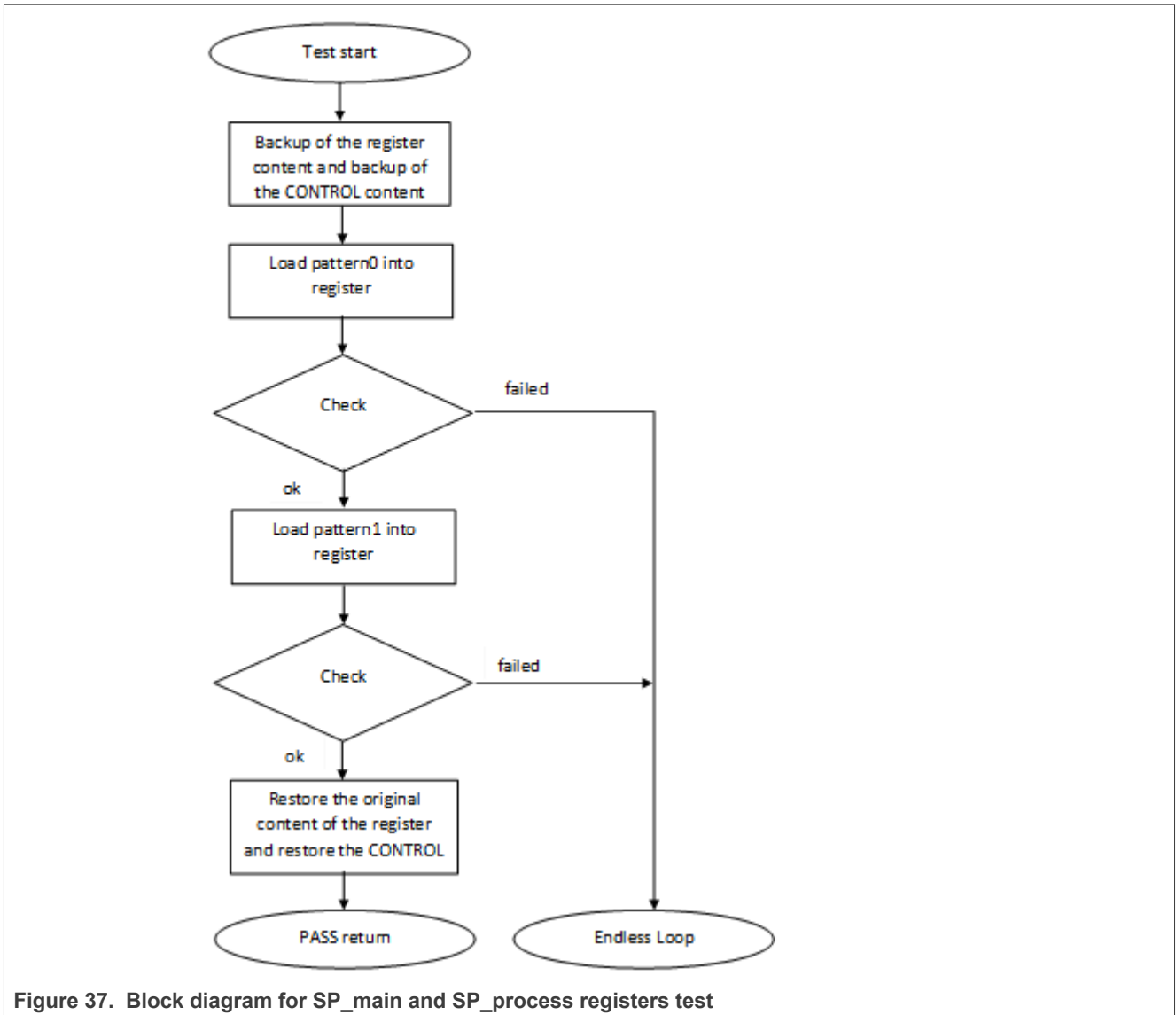


Figure 37. Block diagram for SP\_main and SP\_process registers test

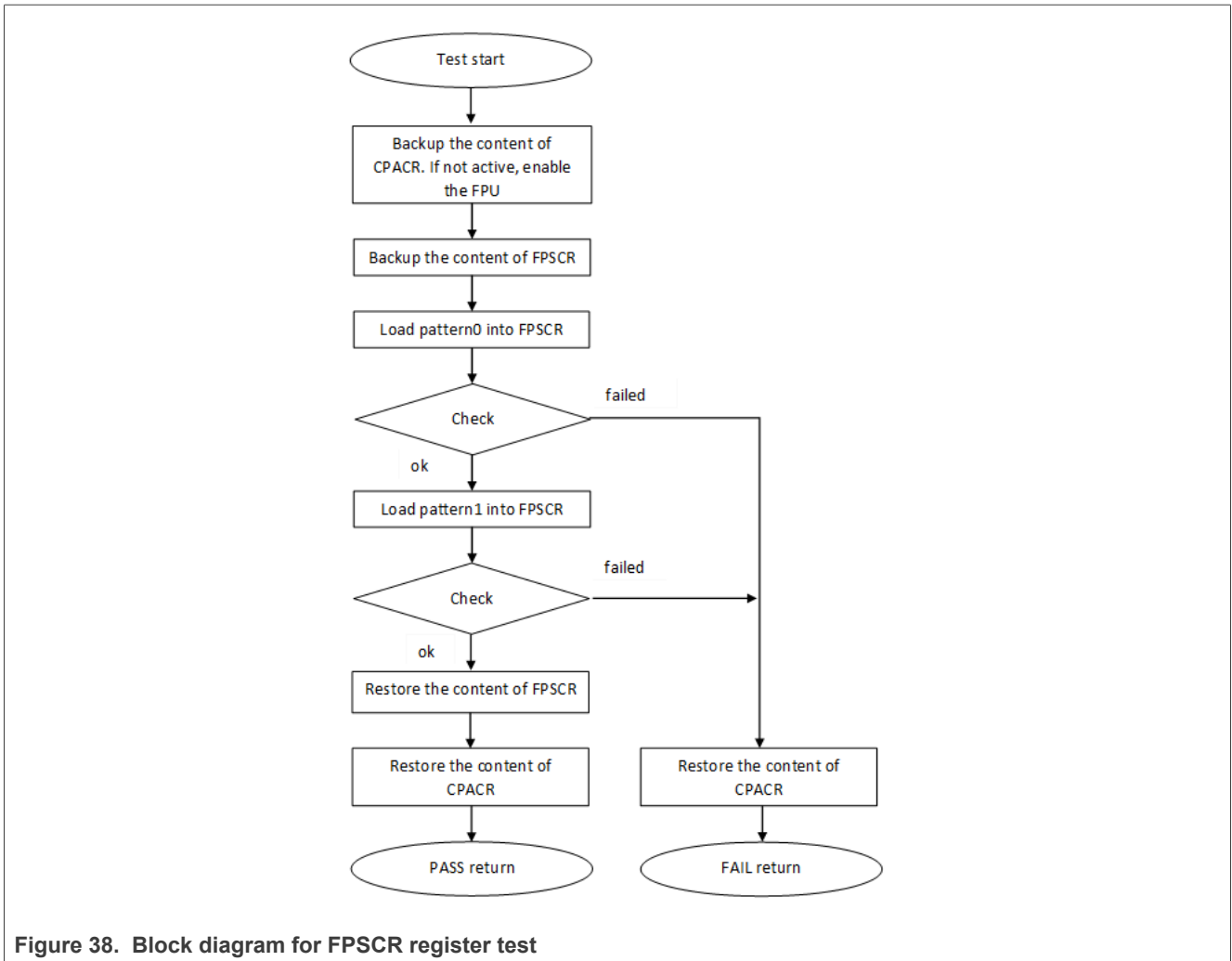
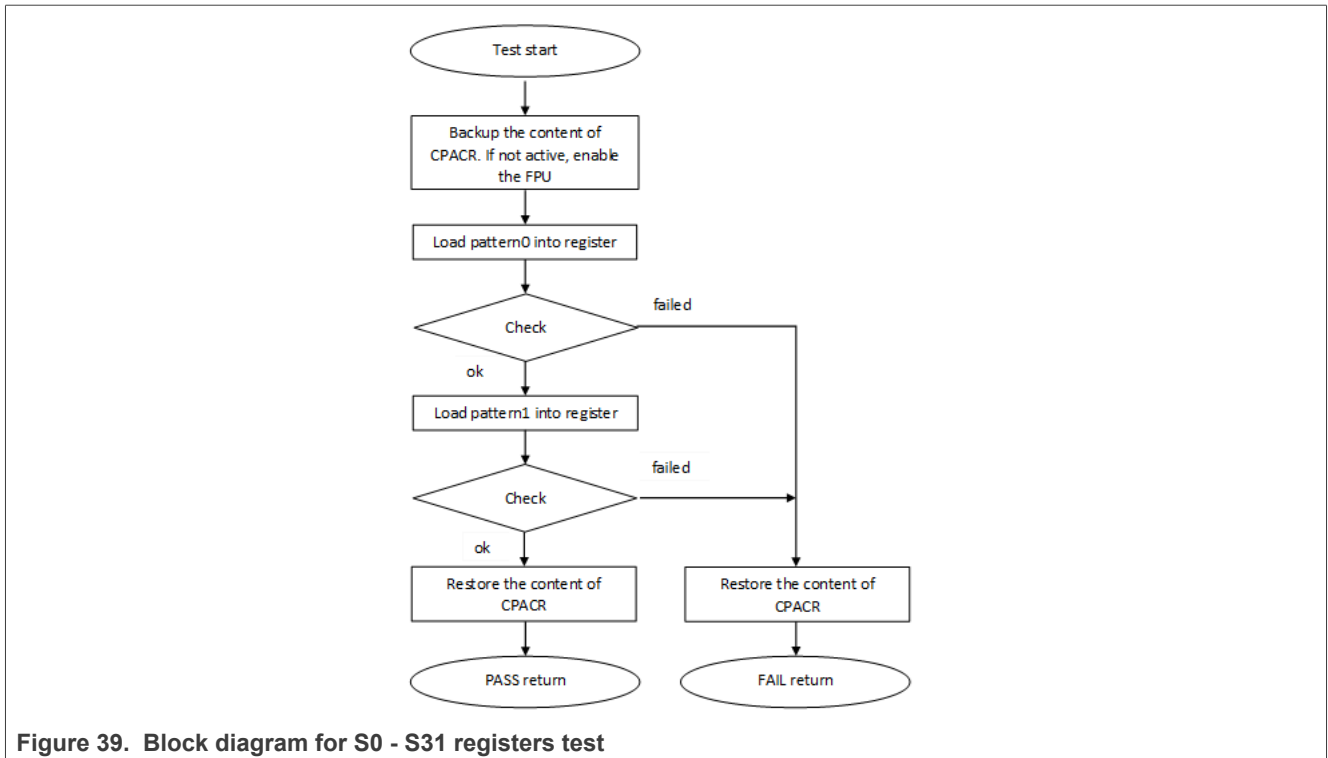


Figure 38. Block diagram for FPSCR register test



### 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 31. 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\_cm4\_cm7\_reg.S* file and they are written as assembler functions. For devices containing the FPU, *iec60730b\_cm4\_cm7\_reg\_fpu.S* is an additional file with tests of FPU-related registers.

The header file with the return values and function prototypes is *iec60730b\_cm4\_cm7\_reg.h*.

The *iec60730b.h*, *asm\_mac\_common.h*, and *iec60730b\_types.h* files are the common header files for the safety library.

The following functions are called to test the corresponding registers:

- *FS\_CM4\_CM7\_CPU\_Register()*
- *FS\_CM4\_CM7\_CPU\_NonStackedRegister()*
- *FS\_CM4\_CM7\_CPU\_Primask()*
- *FS\_CM4\_CM7\_CPU\_SPmain()*
- *FS\_CM4\_CM7\_CPU\_SPprocess()*
- *FS\_CM4\_CM7\_CPU\_Control()*

- *FS\_CM4\_CM7\_CPU\_Special()*
- *FS\_CM4\_CM7\_CPU\_Special8PriorityLevels()*

When the device has an FPU, the following functions are placed in the *iec60730b\_cm4\_cm7\_reg\_fpu.S* file:

- *FS\_CM4\_CM7\_CPU\_ControlFpu()*
- *FS\_CM4\_CM7\_CPU\_Float1()*
- *FS\_CM4\_CM7\_CPU\_Float2()*

Error detection is recognized by a 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_CM4_CM7_CPU_Register())
    SafetyError();
```

### 8.2.1 FS\_CM4\_CM7\_CPU\_Control()

This function tests the CONTROL register according to the [Figure 36](#).

**Function prototype:**

```
FS_RESULT FS_CM4_CM7_CPU_Control(void);
```

**Test pattern:**

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

**Function inputs:**

```
void
```

**Function output:**

```
typedef uint32_t FS_RESULT;
```

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

**Function performance:**

The function takes approximately 30 cycles, including the result comparison (0.375  $\mu$ s).<sup>1</sup>

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

**Calling restrictions:**

This function cannot be interrupted and it must be called in the thread mode (not in the handler mode).

### 8.2.2 FS\_CM4\_CM7\_CPU\_ControlFpu()

This function tests the CONTROL register according to the [Figure 36](#).

**Function prototype:**

*FS\_RESULT FS\_CM4\_CM7\_CPU\_ControlFpu(void);*

**Test pattern:**

*CONTROL: 0x00000000, 0x00000002, 0x00000004*

**Function inputs:**

*void*

**Function output:**

*typedef uint32\_t FS\_RESULT;*

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

**Function performance:**

The function takes approximately 52 cycles, including the result comparison (0.65  $\mu$ s).<sup>1</sup>

The function size is 62 B.<sup>1</sup>

**Calling restrictions:**

This function cannot be interrupted and it must be called in the thread mode (not in the handler mode).

This function should be used for devices with FPU, as a replace of the *FS\_CM4\_CM7\_CPU\_Control()* function.

### 8.2.3 FS\_CM4\_CM7\_CPU\_Float1()

This function checks the FPSCR and S0-S15 registers according to the [Figure 38](#) to the [Figure 39](#). 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\_CM4\_CM7\_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 286 cycles (3.575  $\mu$ s).<sup>1</sup>

The function size is 476 B.<sup>1</sup>

**Calling restrictions:**

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

### 8.2.4 FS\_CM4\_CM7\_CPU\_Float2()

This function checks the S16-S31 registers according to the [Figure 39](#). 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_CM4_CM7_CPU_Float2(void);
```

**Test patterns for respective registers:**

S0-S15: 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 270 cycles (3.375  $\mu$ s). <sup>1</sup>

The function size is 470 B. <sup>1</sup>

**Calling restrictions:**

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

### 8.2.5 FS\_CM4\_CM7\_CPU\_NonStackedRegister()

This function tests the R8, R9, R10, and R11 CPU registers. Each register is tested according to the [Figure 34](#).

**Function prototype:**

```
FS_RESULT FS_CM4_CM7_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 70 cycles, including the result comparison (0.875  $\mu$ s). <sup>1</sup>

The function size is 80 B. <sup>1</sup>

**Calling restrictions:**

None.

### 8.2.6 FS\_CM4\_CM7\_CPU\_Primask()

This function tests the PRIMASK register according to the [Figure 36](#).

**Function prototype:**

```
FS_RESULT FS_CM4_CM7_CPU_Primask(void);
```

**Test pattern:**

```
PRIMASK: 0x00000001, 0x00000000
```

**Function inputs:**

void

**Function output:**

```
typedef uint32_t FS_RESULT;
```

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

**Function performance:**

The function takes approximately 221 cycles, including the result comparison (2.763 µs). <sup>1</sup>

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

**Calling restrictions:**

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

### 8.2.7 FS\_CM4\_CM7\_CPU\_Register()

This function tests the R0-R7, R12, LR, and APSR CPU registers in a sequence. Each register is tested according to the [Figure 34](#) to the [Figure 35](#).

**Function prototype:**

```
FS_RESULT FS_CM4_CM7_CPU_Register(void);
```

**Test patterns for respective registers:**

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

```
APSR: 0x50000000, 0xA0000000
```

**Function inputs:**

void

**Function output:**

```
typedef uint32_t FS_RESULT;
```

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

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

**Function performance:**

Function duration is approximately 172 cycles including result comparison (2.15 µs) <sup>1</sup>

Function size is 204 bytes.<sup>1</sup>

**Calling restrictions:**

None.

### 8.2.8 FS\_CM4\_CM7\_CPU\_Special()

This function tests the BASEPRI and FAULTMASK registers according to the [Figure 36](#).

**Function prototype:**

```
FS_RESULT FS_CM4_CM7_CPU_Special(void);
```

**Test pattern:**

BASEPRI: 0xA0, 0x50

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 61 cycles (0.763 µs).<sup>1</sup>

The function size is 104 B.<sup>1</sup>

**Calling restrictions:**

None.

### 8.2.9 FS\_CM4\_CM7\_CPU\_Special8PriorityLevels()

This function tests the BASEPRI and FAULTMASK registers according to the [Figure 36](#).

**Function prototype:**

```
FS_RESULT FS_CM4_CM7_CPU_Special8PriorityLevels(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 53 cycles (1.104  $\mu$ s). <sup>6</sup>

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

**Calling restrictions:**

For devices with eight priority levels for interrupts.

**8.2.10 FS\_CM4\_CM7\_CPU\_SPmain()**

This function tests the SP\_main register according to the [Figure 37](#).

**Function prototype:**

```
FS_RESULT FS_CM4_CM7_CPU_SPmain(void);
```

**Test pattern:**

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

**Function inputs:**

*void*

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*

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

**Function performance:**

The function takes approximately 59 cycles, including the result comparison (0.738  $\mu$ s) <sup>1</sup>

The function size is 58 B. <sup>1</sup>

**Calling restrictions:**

This function cannot be interrupted.

**8.2.11 FS\_CM4\_CM7\_CPU\_SPprocess()**

This function checks SP\_process register according to the [Figure 37](#).

**Function prototype:**

```
FS_RESULT FS_CM4_CM7_CPU_SPprocess(void);
```

**Test pattern:**

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

**Function inputs:**

*void*

**Function output:**

```
typedef uint32_t FS_RESULT;
```

- *FS\_PASS*

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

**Function performance:**

The function takes approximately 51 cycles, including the result comparison (0.638  $\mu$ s)<sup>1</sup>

The function size is 58 B.<sup>1</sup>

**Calling restrictions:**

This function cannot be interrupted.

## 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__ = 0x1FFFFFFC10;
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_4_cm7_stack.S` file and they are written as assembler functions. The header file with the return values and the function prototypes is `iec60730b_4_cm7_stack.h`. The `iec60730b.h`, `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\_CM4\_CM7\_STACK\_Init

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

##### **Example of initialization:**

```

#include "iec60730b.h"

extern unsigned long STACK_TEST_P_2;
extern unsigned long STACK_TEST_P_3;
const unsigned long stack_test_first_address = (unsigned long) &STACK_TEST_P_2;
const unsigned long stack_test_second_address = (unsigned long) &STACK_TEST_P_3;
const unsigned long stack_test_pattern = 0x77777777;
const unsigned long stack_test_block_size = 0x10;

```

##### **Function prototype:**

```
void FS_CM4_CM7_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 (e.g. 0x77777777).

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

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

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

**Function output:**

*void*

**Function performance:**

The function takes approximately 86 cycles for a block size of 0x10. (1.075  $\mu$ s)<sup>1</sup>

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

**Calling restrictions:**

None.

### 9.3.2 FS\_CM4\_CM7\_STACK\_Test

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

**Function prototype:**

```
FS_RESULT FS_CM4_CM7_STACK_Test(uint32_t stackTestPattern, uint32_t firstAddress, uint32_t secondAddress, uint32_t blockSize);
```

**Function inputs:**

*stackTestPattern* - The test pattern (e.g. 0x77777777).

*firstAddress* - The first address of block in front of stack.

*secondAddress* - The first address of block behind the stack.

*blockSize* - The block size.

**Function output:**

```
typedef uint32_t FS_RESULT;
```

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

**Function performance:**

The function for block size 0x10 takes approximately 117 cycles (1.463  $\mu$ s).<sup>1</sup>

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

**Calling restrictions:**

None.

## 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:

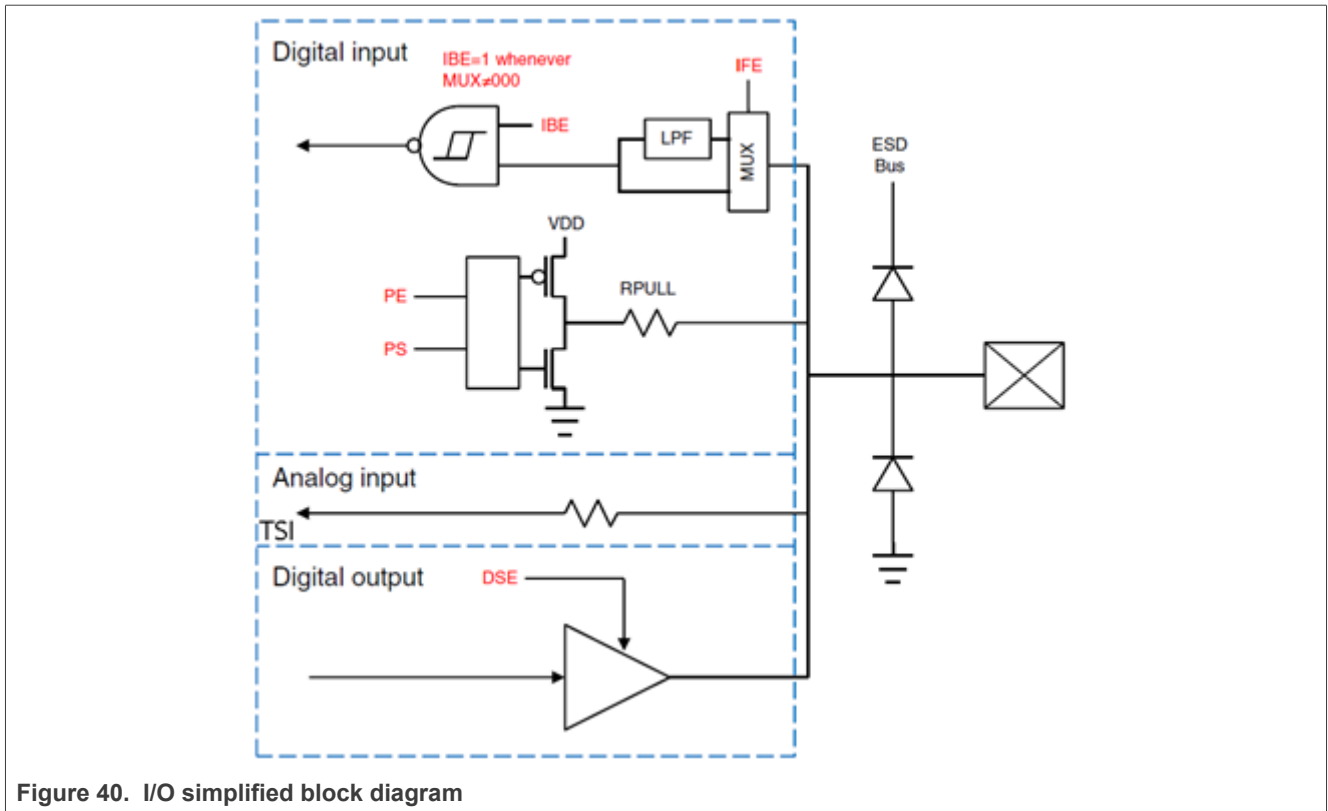


Figure 40. I/O simplified block diagram

### 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 [Section "Digital input/output test"](#))

### 10.2 TSI input test

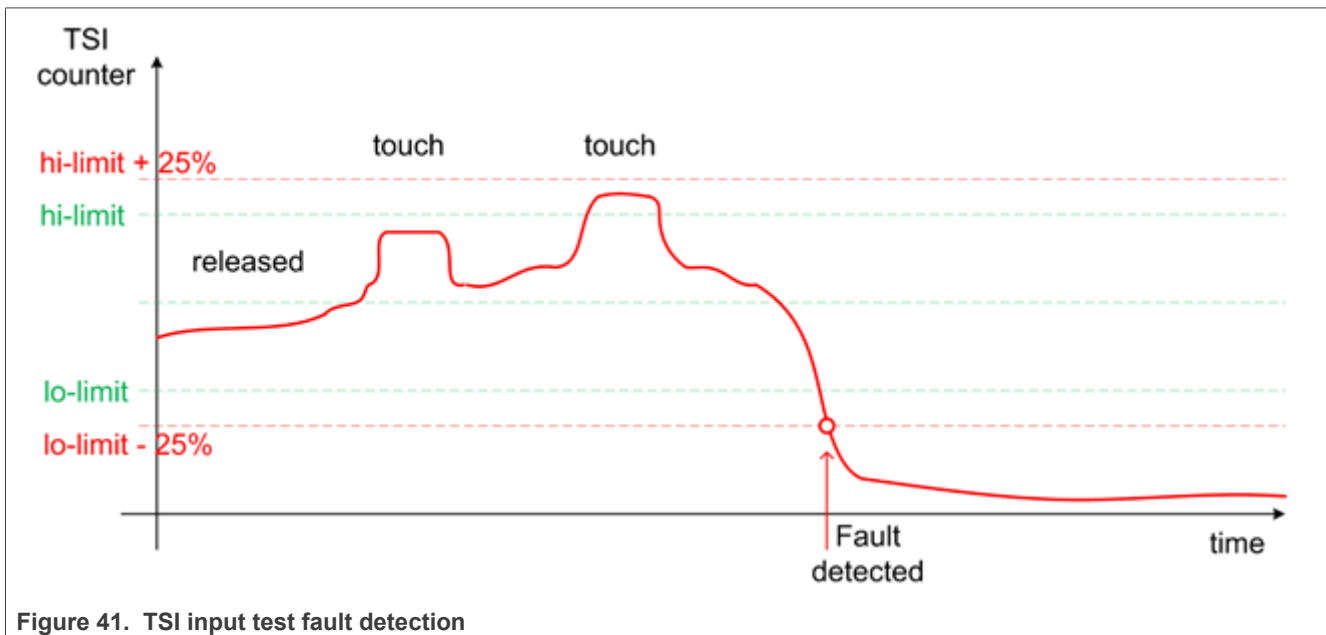
This test is responsible for checking the typical conversion results of the individual TSI channels. When the touch-sensing electrode is released (not touched), the typical conversion result is given by the intrinsic (parasitic) capacitance load connected externally to the TSI channel. The intrinsic 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.



**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:

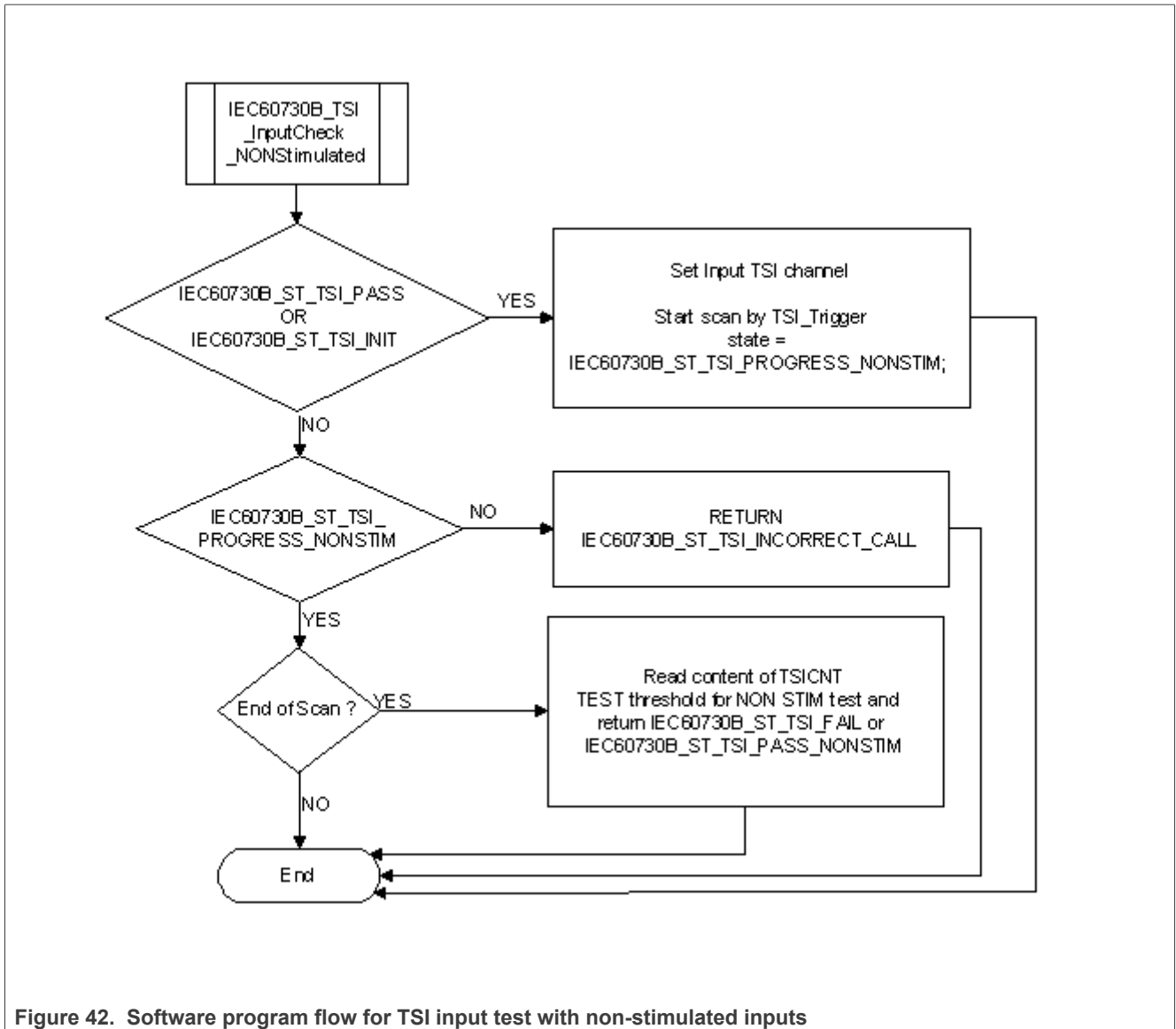


Figure 42. Software program flow for TSI input test with non-stimulated inputs

### 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.

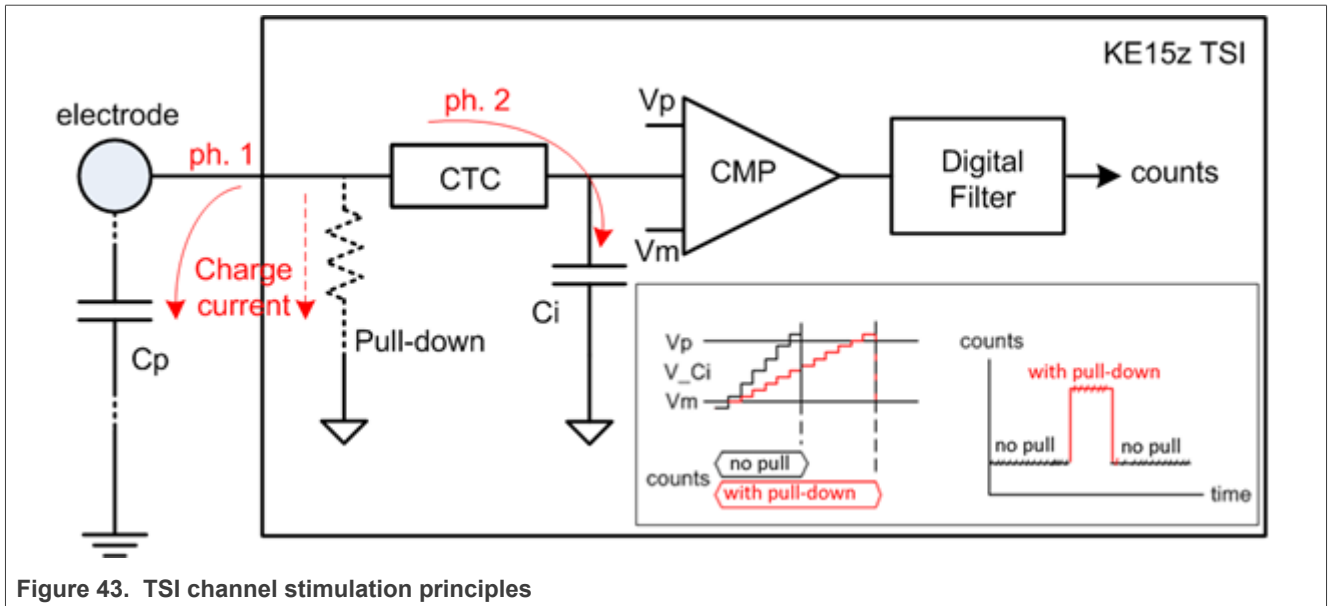


Figure 43. TSI channel stimulation principles

### 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.

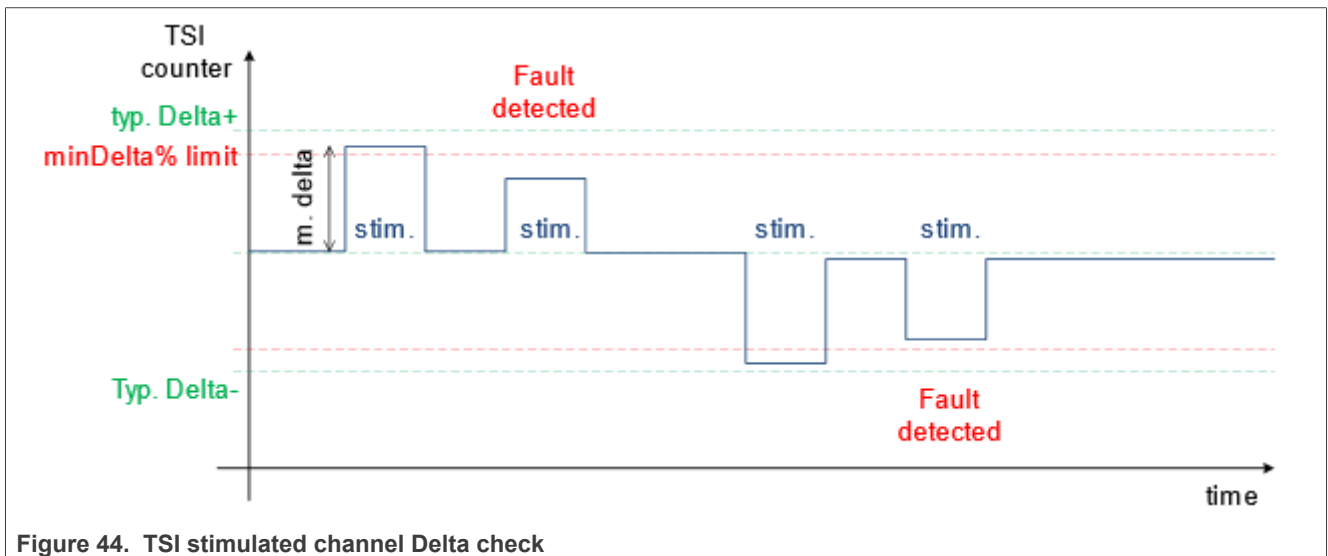


Figure 44. TSI stimulated channel Delta check

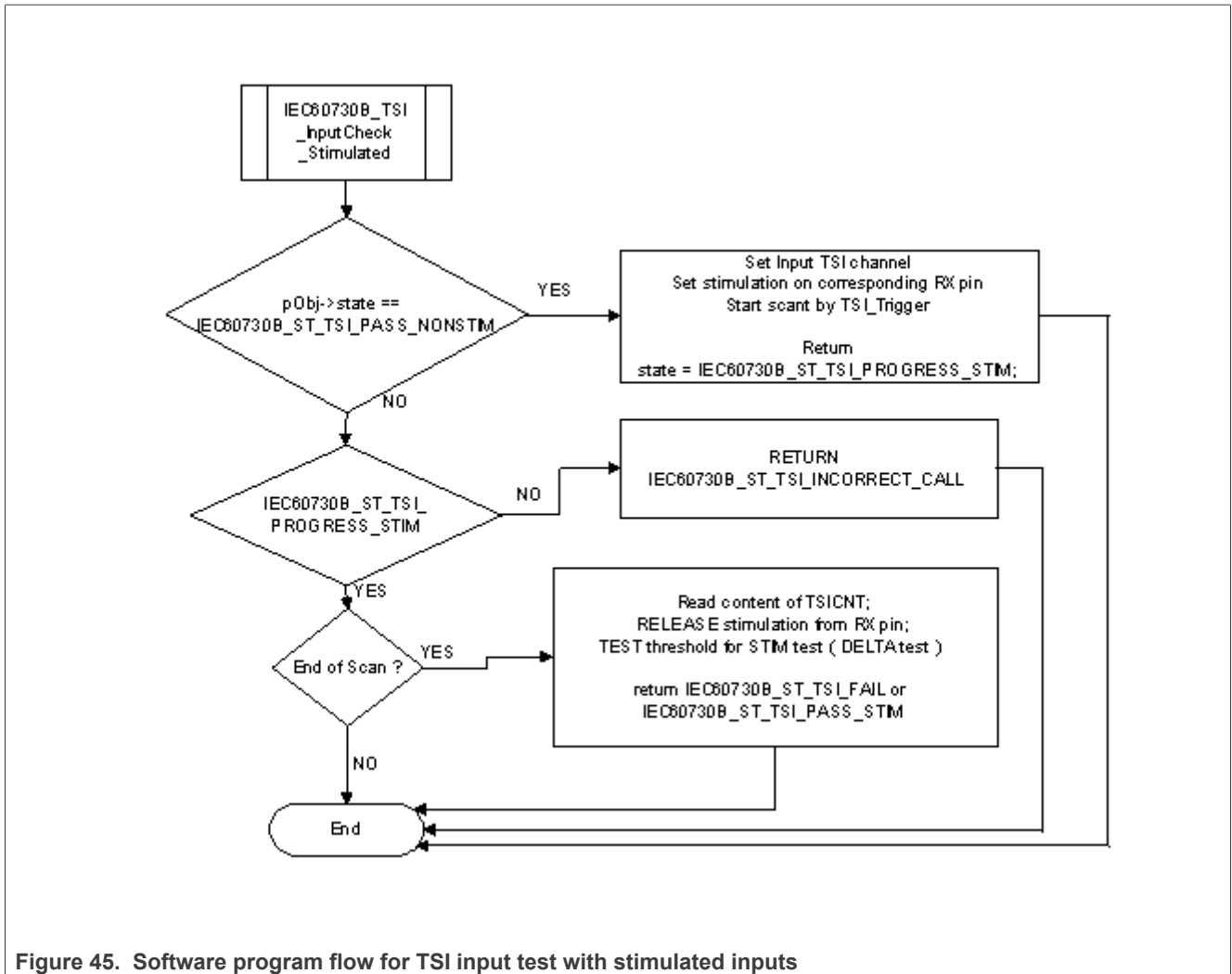


Figure 45. Software program flow for TSI input test with stimulated inputs

### 10.5 TSI test implementation

The test functions for the TSI IO test are in the *iec60730b\_tsi.c* file and they are 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()*
- *FS\_TSI\_InputCheckNONStimulated\_v6()*
- *FS\_TSI\_InputCheckStimulated\_v6()*
- *FS\_TSI\_InputStimulate\_v6()*
- *FS\_TSI\_InputRelease\_v6()*

### 10.5.1 TSI input test principles

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

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

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

### 10.5.2 TSI stimulated input test principles

This test is responsible for a periodical check of the TSI counter delta change on the input stimulated by an internal pull-up. The test is triggered by the `FS_TSI_InputCheckStimulated()` function call. When the channel measurement completes, the appropriate pull resistor is disabled on the current input. The TSI counter value measured with the stimulated input is compared with the value gathered previously without stimulation. This difference is called the TSI delta signal. The TSI input channel is working properly when the delta signal is non-zero. It means that a significant counter change is measured while the input is stimulated. Depending on the TSI sensing mode and the polarity of stimulation, the delta value may have positive or negative signs. This delta value is then compared with the typical delta value experimentally measured and predefined in the configuration file. It means that the typical delta values must be measured in advance during the calibration of a known and good device. See [Section "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()` ( or /\_v6) and `FS_TSI_InputCheckStimulated()` ( or /\_v6) 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 is dedicated for both TSI\_v5 and TSI\_v6 peripherals. 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 is dedicated for the TSI\_v5 peripheral. 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 is dedicated for the TSI\_v5 peripheral. 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 address of the TSI\_v5 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\_v5 pin by the pull-resistor on the current TSI channel when the TSI input stimulation is required. The pull-up/down polarity is given by the stim\_polarity parameter in the fs\_tsi\_t structure.

#### **Function prototype:**

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

#### **Function inputs:**

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

#### **Function output:**

```
typedef uint32_t FS_RESULT;
```

- FS\_PASS
- FS\_FAIL\_TSI

#### **Function performance:**

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

### 10.5.8 FS\_TSI\_InputRelease()

This function disables the pull-resistor stimulation on the appropriate TSI\_v5 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](#).

### 10.5.9 FS\_TSI\_InputCheckNONStimulated\_v6()

This function is dedicated for the TSI\_v6 peripheral. This function executes the first part of the TSI\_v6 test sequence with a non-stimulated input. It reads the TSI\_v6 counter value and checks whether the value fits into the predefined limits. It also gathers the TSI\_v6 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_v6(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\_v6 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](#).

### 10.5.10 FS\_TSI\_InputCheckStimulated\_v6()

This function is dedicated for the TSI\_v6 peripheral. This function executes the second part of the TSI test sequence with a stimulated input. It checks whether the TSI\_v6 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_v6()` call precedes the `FS_TSI_InputCheckStimulated_v6()` 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_v6(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\_v6 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](#).

### 10.5.11 FS\_TSI\_InputStimulate\_v6()

This function is dedicated for the TSI\_v6 peripheral. The function stimulates the appropriate TSI\_v6 pin by the pull-resistor on the current TSI\_v6 channel when the TSI input stimulation is required. The pull-up/down polarity is given by the `stim_polarity` parameter in the `fs_tsi_t` structure.

**Function prototype:**

```
FS_RESULT FS_TSI_InputStimulate_v6(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](#).

### 10.5.12 FS\_TSI\_InputRelease\_v6()

This function is dedicated for the TSI\_v6 peripheral. This function disables the pull-resistor stimulation on the appropriate TSI\_v6 channel. It is also called internally by the `FS_TSI_InputCheckStimulated_v6()` function as soon as the stimulated input check completes.

**Function prototype:**

```
FS_RESULT FS_TSI_InputRelease_v6(fs_tsi_t *pObj);
```

**Function inputs:**

*\*pObj* - The input argument is the pointer to the TSI\_v6 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](#).

## 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 32](#).

**Table 32. 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, which is not corrupted after the non-POR in the RAM memory.

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:**

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

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 correctly if your example application is set to a correct version. For other devices, compare the reference manual of your device with [Table 33](#) or with the reference device in the following table.

Table 33. Refresh sequence

| Refresh Index parameter | Refresh sequence                                                                                                                                                           | Reference device |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|
| FS_KINETIS_WDOG         | <ul style="list-style-type: none"> <li>WdogBase-&gt;REFRESH = 0xA602U;</li> <li>WdogBase-&gt;REFRESH = 0xB480U; /* refresh sequence */</li> </ul>                          | MKV11            |
| FS_WDOG32               | WdogBase->CNT = 0xB480A602U; /* refresh sequence */                                                                                                                        | MK32L2A          |
| FS_COP_WDOG             | <ul style="list-style-type: none"> <li>WdogBase-&gt;SRVCOP = FS_SIM_KL2X_SRVCOP_SRVCOP(0x55U);</li> <li>WdogBase-&gt;SRVCOP = FS_SIM_KL2X_SRVCOP_SRVCOP(0xAAU);</li> </ul> | 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:**

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

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\_IMX\_GPT()

This function can be used for devices with the GPT timer and a supported WDOG. 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 endless waiting 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.

`refresh_index` - The index of the refresh sequence. It can be `FS_IMXRT`, `FS_IMX8M`.

#### 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 the function is **TBD bytes**.

#### Calling restrictions:

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

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 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 GPT timer used.
- `wdogBackup->WdogBase` - The base address of the WDOG used.

The "refresh\_index" parameter is used to choose the type of the WDOG used. The function supports two types of WDOG for MIMX devices:

- `FS_IMXRT` - situated for example on `IMXRT1050`.

- FS\_IMX8M - situated for example on MIMX8MM.

It is necessary to compare the register memory map for your device with these three used types and choose a corresponding refresh sequence.

#### 11.2.4 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 corresponding error under the following conditions:

- 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* - Boolean value. If it is TRUE, the WDOG reset flag from the reset-detection register is deleted.

*RegWide8b* - When it is TRUE, the reset-detection register is accessed as 8b (32b otherwise).

##### **Function output:**

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

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.

#### 11.2.5 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:**

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

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.

### 11.2.6 FS\_WDOG\_Setup\_WWDT\_CTIMER()

This function can be used for the devices with the WWDT and CTIMER peripherals. 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_CTIMER(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:**

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

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.

### 11.2.7 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 conditions:

- 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.

#### Function output:

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*

#### Function performance:

For information about the function performance, see [Core self-test library – source code version](#).

#### Calling restrictions:

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

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 conditions:

- 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_LPC55SXX(uint32_t limitHigh, uint32_t limitLow, uint32_t limitResets,
bool_t endlessLoopEnable, fs_wdog_test_t *pWatchdogBackup);
```

#### Function inputs:

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

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

*limitResets* - The limit value for watchdog resets.

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

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

#### Function output:

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*

#### Function performance:

For information about the function performance, see [Core self-test library – source code version](#).

#### Calling restrictions:

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

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.

### 11.2.9 FS\_WDOG\_Check\_WWDT\_MCX()

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 conditions:

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

#### Function inputs:

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

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

*limitResets* - The limit value for watchdog resets.

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

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

#### Function output:

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

#### Function performance:

For information about the function performance, see [Core self-test library – source code version](#).

#### Calling restrictions:

Check the compatibility of this function with your device in the "Dedicated functions" section. [Core self-test library – source code version](#).

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.

## 12 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2025 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials must be provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### 13 Revision history

Table 34. Revision table

| Revision number | Release date     | Description                                      |
|-----------------|------------------|--------------------------------------------------|
| UG10315 v. 1.0  | 1 September 2025 | Release of IEC60730B safety class B library v5.0 |

## Legal information

### Definitions

**Draft** — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

### Disclaimers

**Limited warranty and liability** — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

**Right to make changes** — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

**Suitability for use** — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

**Applications** — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

**Terms and conditions of commercial sale** — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at <https://www.nxp.com/profile/terms>, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

**Export control** — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

**Suitability for use in non-automotive qualified products** — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

**HTML publications** — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

**Translations** — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

**Security** — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at [PSIRT@nxp.com](mailto:PSIRT@nxp.com)) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

**NXP B.V.** — NXP B.V. is not an operating company and it does not distribute or sell products.

### Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

**NXP** — wordmark and logo are trademarks of NXP B.V.

**AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro,  $\mu$ Vision, Versatile** — are trademarks and/or registered trademarks of Arm Limited (or its subsidiaries or affiliates) 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.

**IAR** — is a trademark of IAR Systems AB.

Tables

|          |                                                                            |    |          |                                                                                         |     |
|----------|----------------------------------------------------------------------------|----|----------|-----------------------------------------------------------------------------------------|-----|
| Tab. 1.  | List of library items .....                                                | 3  | Tab. 21. | Duration of FS_FLASH_C_HW16_K()<br>depending on tested block size .....                 | 95  |
| Tab. 2.  | MIMX8MMx dedicated functions .....                                         | 7  | Tab. 22. | Duration of FS_FLASH_C_HW16_L()<br>depending on tested block size .....                 | 96  |
| Tab. 3.  | MIMX8MNx and MIMX8MLx dedicated<br>functions .....                         | 8  | Tab. 23. | Duration of FS_CM4_CM7_FLASH_<br>HW32_DCP() in dependence of tested<br>block size ..... | 97  |
| Tab. 4.  | MIMXRT10xx dedicated functions .....                                       | 9  | Tab. 24. | Duration of FS_CM4_CM7_FLASH_<br>HW16() in dependence of tested block size ....         | 98  |
| Tab. 5.  | MIMXRT117x/116x dedicated functions .....                                  | 10 | Tab. 25. | Duration of FS_CM4_CM7_FLASH_<br>SW16() in dependence of tested block size ....         | 99  |
| Tab. 6.  | MIMXRT118x CM7 dedicated functions .....                                   | 10 | Tab. 26. | Duration of FS_CM4_CM7_FLASH_<br>SW32() in dependence of tested block size ....         | 99  |
| Tab. 7.  | MK2xF dedicated functions .....                                            | 11 | Tab. 27. | CPU program counter test in compliance<br>with IEC/UL standards .....                   | 102 |
| Tab. 8.  | MKE1xF dedicated functions .....                                           | 12 | Tab. 28. | Variable memory test in compliance with<br>IEC and UL standards .....                   | 106 |
| Tab. 9.  | MKV3x dedicated functions .....                                            | 13 | Tab. 29. | FS_CM4_CM7_RAM_AfterReset duration ....                                                 | 107 |
| Tab. 10. | MKV4x dedicated functions .....                                            | 13 | Tab. 30. | FS_CM4_CM7_RAM_Runtime duration .....                                                   | 108 |
| Tab. 11. | MKV5x dedicated functions .....                                            | 14 | Tab. 31. | CPU register test in compliance with IEC<br>and UL standards .....                      | 116 |
| Tab. 12. | LPC54S0x/LPC540x dedicated functions .....                                 | 15 | Tab. 32. | Watchdog test in compliance with the<br>standards .....                                 | 137 |
| Tab. 13. | MK32L3 dedicated functions for CM4 core .....                              | 16 | Tab. 33. | Refresh sequence .....                                                                  | 139 |
| Tab. 14. | MCXE31x dedicated functions .....                                          | 17 | Tab. 34. | Revision table .....                                                                    | 146 |
| Tab. 15. | MCXE24x dedicated functions .....                                          | 17 |          |                                                                                         |     |
| Tab. 16. | Analog input/output test in compliance with<br>IEC and UL standards .....  | 19 |          |                                                                                         |     |
| Tab. 17. | Clock test in compliance with IEC and UL<br>standards .....                | 36 |          |                                                                                         |     |
| Tab. 18. | Digital input/output test in compliance with<br>IEC and UL standards ..... | 41 |          |                                                                                         |     |
| Tab. 19. | Invariable memory test in compliance with<br>IEC and UL standards .....    | 91 |          |                                                                                         |     |
| Tab. 20. | Duration of FS_FLASH_C_HW32_K()<br>depending on tested block size .....    | 94 |          |                                                                                         |     |

Figures

|          |                                                                     |    |          |                                                                     |     |
|----------|---------------------------------------------------------------------|----|----------|---------------------------------------------------------------------|-----|
| Fig. 1.  | Block diagram for analog input test .....                           | 19 | Fig. 19. | Block diagram of FS_DIO_<br>ShortToSupplySet_IMXRT() function ..... | 69  |
| Fig. 2.  | ADC Test Code flow .....                                            | 21 | Fig. 20. | Extended digital input test for LPC devices .....                   | 71  |
| Fig. 3.  | Block diagram for clock test .....                                  | 36 | Fig. 21. | Block diagram for digital output test .....                         | 73  |
| Fig. 4.  | Block diagram for digital input test .....                          | 42 | Fig. 22. | Block diagram of FS_DIO_ShortToAdjSet_<br>LPC() function .....      | 75  |
| Fig. 5.  | Block diagram for digital output test .....                         | 44 | Fig. 23. | Block diagram of FS_DIO_<br>ShortToSupplySet_LPC function .....     | 77  |
| Fig. 6.  | Extended digital input test .....                                   | 45 | Fig. 24. | Extended digital input test for IMX RGPIO .....                     | 79  |
| Fig. 7.  | Block diagram of FS_DIO_ShortToAdjSet()<br>function .....           | 47 | Fig. 25. | Block diagram of FS_DIO_ShortToAdjSet_<br>RGPIO() function .....    | 81  |
| Fig. 8.  | Block diagram of FS_DIO_<br>ShortToSupplySet function .....         | 49 | Fig. 26. | Block diagram of FS_DIO_<br>ShortToSupplySet_RGPIO() function ..... | 83  |
| Fig. 9.  | Extended digital input test .....                                   | 51 | Fig. 27. | Extended digital input test for MCXE31x .....                       | 85  |
| Fig. 10. | Block diagram of FS_DIO_ShortToAdjSet_<br>MCX() function .....      | 53 | Fig. 28. | Block diagram of FS_DIO_ShortToAdjSet_<br>SIUL2() function .....    | 87  |
| Fig. 11. | Block diagram of FS_DIO_<br>ShortToSupplySet_MCX function .....     | 55 | Fig. 29. | Block diagram of FS_DIO_<br>ShortToSupplySet_SIUL2() function ..... | 89  |
| Fig. 12. | Extended digital input test for IMX8M .....                         | 57 | Fig. 30. | Checksum settings for linker .....                                  | 92  |
| Fig. 13. | Block diagram for digital output test .....                         | 58 | Fig. 31. | Block diagram for PC_Test .....                                     | 101 |
| Fig. 14. | Block diagram of FS_DIO_ShortToAdjSet_<br>IMX8M() function .....    | 60 | Fig. 32. | Block diagram for after-reset test of RAM .....                     | 104 |
| Fig. 15. | Block diagram of FS_DIO_<br>ShortToSupplySet_IMX8M() function ..... | 62 | Fig. 33. | Block diagram for runtime test of RAM .....                         | 105 |
| Fig. 16. | Extended digital input test for IMXRT .....                         | 64 | Fig. 34. | Block diagram for R2 – R12 registers test .....                     | 111 |
| Fig. 17. | Block diagram for digital output test .....                         | 65 | Fig. 35. | Block diagram for R0, R1, LR, APSR<br>registers test .....          | 112 |
| Fig. 18. | Block diagram of FS_DIO_ShortToAdjSet_<br>IMXRT() function .....    | 67 |          |                                                                     |     |

|          |                                                                                       |     |          |                                                                              |     |
|----------|---------------------------------------------------------------------------------------|-----|----------|------------------------------------------------------------------------------|-----|
| Fig. 36. | Block diagram for PRIMASK ,<br>FAULTMAST, BASEPRI and CONTROL<br>registers test ..... | 113 | Fig. 41. | TSI input test fault detection .....                                         | 127 |
| Fig. 37. | Block diagram for SP_main and SP_<br>process registers test .....                     | 114 | Fig. 42. | Software program flow for TSI input test<br>with non-stimulated inputs ..... | 129 |
| Fig. 38. | Block diagram for FPSCR register test .....                                           | 115 | Fig. 43. | TSI channel stimulation principles .....                                     | 130 |
| Fig. 39. | Block diagram for S0 - S31 registers test .....                                       | 116 | Fig. 44. | TSI stimulated channel Delta check .....                                     | 130 |
| Fig. 40. | I/O simplified block diagram .....                                                    | 126 | Fig. 45. | Software program flow for TSI input test<br>with stimulated inputs .....     | 131 |

Contents

|          |                                                                    |           |          |                                                                     |           |
|----------|--------------------------------------------------------------------|-----------|----------|---------------------------------------------------------------------|-----------|
| <b>1</b> | <b>Core self-test library</b> .....                                | <b>2</b>  | 2.2.7.1  | fs_ao_a8_t .....                                                    | 33        |
| 1.1      | Core-dependent part .....                                          | 2         | 2.2.7.2  | fs_ao_test_a23468_t .....                                           | 33        |
| 1.2      | Peripheral-dependent part .....                                    | 2         | 2.2.7.3  | FS_AIO_InputSet_A8() .....                                          | 34        |
| 1.3      | Core self-test library – source code .....                         | 2         | 2.2.7.4  | FS_AIO_ReadResult_A8() .....                                        | 34        |
| 1.3.1    | MIMX8MMx dedicated functions .....                                 | 7         | 2.2.8    | FS_AIO_LimitCheck() .....                                           | 35        |
| 1.3.2    | MIMX8MNx and MIMX8MLx dedicated functions .....                    | 8         | <b>3</b> | <b>Clock test</b> .....                                             | <b>35</b> |
| 1.3.3    | MIMXRT10xx dedicated functions .....                               | 9         | 3.1      | Clock test in compliance with IEC/UL standards .....                | 36        |
| 1.3.4    | MIMXRT117x/116x dedicated functions .....                          | 9         | 3.2      | Clock test implementation .....                                     | 36        |
| 1.3.5    | MIMXRT118x CM7 dedicated functions .....                           | 10        | 3.2.1    | FS_CLK_Init() .....                                                 | 37        |
| 1.3.6    | MK2xF dedicated functions .....                                    | 11        | 3.2.2    | FS_CLK_Check() .....                                                | 37        |
| 1.3.7    | MKE1xF dedicated functions .....                                   | 12        | 3.2.3    | FS_CLK_LPTMR() .....                                                | 38        |
| 1.3.8    | MKV3x dedicated functions .....                                    | 12        | 3.2.4    | FS_CLK_RTC() .....                                                  | 38        |
| 1.3.9    | MKV4x dedicated functions .....                                    | 13        | 3.2.5    | FS_CLK_GPT() .....                                                  | 39        |
| 1.3.10   | MKV5x dedicated functions .....                                    | 14        | 3.2.6    | FS_CLK_TIMER() .....                                                | 39        |
| 1.3.11   | LPC54S0x/LPC540x dedicated functions .....                         | 15        | 3.2.7    | FS_CLK_WKT_LPC() .....                                              | 39        |
| 1.3.12   | MK32L3 CM4 dedicated functions .....                               | 16        | 3.2.8    | FS_CLK_STM() .....                                                  | 40        |
| 1.3.13   | MCXE31x dedicated functions .....                                  | 16        | <b>4</b> | <b>Digital input/output test</b> .....                              | <b>40</b> |
| 1.3.14   | MCXE24x dedicated functions .....                                  | 17        | 4.1      | Digital input/output test in compliance with IEC/UL standards ..... | 40        |
| 1.4      | Functions performance measurement .....                            | 18        | 4.2      | Digital input/output test implementation .....                      | 41        |
| <b>2</b> | <b>Analog Input/Output (IO) test</b> .....                         | <b>18</b> | 4.2.1    | FS_DIO_Input() .....                                                | 42        |
| 2.1      | Analog input/output test in compliance with IEC/UL standards ..... | 19        | 4.2.2    | FS_DIO_Output() .....                                               | 43        |
| 2.2      | Analog input/output test implementation .....                      | 19        | 4.2.3    | FS_DIO_InputExt() .....                                             | 45        |
| 2.2.1    | ADC type A1 .....                                                  | 23        | 4.2.4    | FS_DIO_ShortToAdjSet() .....                                        | 46        |
| 2.2.1.1  | fs_ao_a1_t .....                                                   | 23        | 4.2.5    | FS_DIO_ShortToSupplySet() .....                                     | 48        |
| 2.2.1.2  | fs_ao_test_a1_t .....                                              | 23        | 4.2.6    | FS_DIO_InputExt_MCX() .....                                         | 50        |
| 2.2.1.3  | FS_AIO_InputSet_A1() .....                                         | 23        | 4.2.7    | FS_DIO_ShortToAdjSet_MCX() .....                                    | 52        |
| 2.2.1.4  | FS_AIO_ReadResult_A1() .....                                       | 24        | 4.2.8    | FS_DIO_ShortToSupplySet_MCX() .....                                 | 54        |
| 2.2.2    | ADC type A23 .....                                                 | 24        | 4.2.9    | FS_DIO_InputExt_IMX8M() .....                                       | 56        |
| 2.2.2.1  | fs_ao_a23_t .....                                                  | 25        | 4.2.10   | FS_DIO_Output_IMX8M() .....                                         | 58        |
| 2.2.2.2  | fs_ao_test_a23468_t .....                                          | 25        | 4.2.11   | FS_DIO_ShortToAdjSet_IMX8M() .....                                  | 59        |
| 2.2.2.3  | FS_AIO_InputSet_A23() .....                                        | 25        | 4.2.12   | FS_DIO_ShortToSupplySet_IMX8M() .....                               | 61        |
| 2.2.2.4  | FS_AIO_ReadResult_A23() .....                                      | 26        | 4.2.13   | FS_DIO_InputExt_IMXRT() .....                                       | 63        |
| 2.2.3    | ADC type A4 .....                                                  | 26        | 4.2.14   | FS_DIO_Output_IMXRT() .....                                         | 65        |
| 2.2.3.1  | fs_ao_a4_t .....                                                   | 26        | 4.2.15   | FS_DIO_ShortToAdjSet_IMXRT() .....                                  | 66        |
| 2.2.3.2  | fs_ao_test_a23468_t .....                                          | 26        | 4.2.16   | FS_DIO_ShortToSupplySet_IMXRT() .....                               | 68        |
| 2.2.3.3  | FS_AIO_InputSet_A4() .....                                         | 27        | 4.2.17   | FS_DIO_InputExt_LPC() .....                                         | 70        |
| 2.2.3.4  | FS_AIO_InputSet_A4_2() .....                                       | 27        | 4.2.18   | FS_DIO_Output_LPC() .....                                           | 72        |
| 2.2.3.5  | FS_AIO_ReadResult_A4() .....                                       | 28        | 4.2.19   | FS_DIO_ShortToAdjSet_LPC() .....                                    | 74        |
| 2.2.4    | ADC type A6 .....                                                  | 28        | 4.2.20   | FS_DIO_ShortToSupplySet_LPC() .....                                 | 76        |
| 2.2.4.1  | fs_ao_a6_t .....                                                   | 28        | 4.2.21   | FS_DIO_InputExt_RGPI() .....                                        | 78        |
| 2.2.4.2  | fs_ao_test_a23468_t .....                                          | 28        | 4.2.22   | FS_DIO_ShortToAdjSet_RGPI() .....                                   | 80        |
| 2.2.4.3  | FS_AIO_InputSet_A6() .....                                         | 29        | 4.2.23   | FS_DIO_ShortToSupplySet_RGPI() .....                                | 82        |
| 2.2.4.4  | FS_AIO_ReadResult_A6() .....                                       | 29        | 4.2.24   | FS_DIO_InputExt_SIUL2() .....                                       | 84        |
| 2.2.5    | ADC type A5 .....                                                  | 30        | 4.2.25   | FS_DIO_ShortToAdjSet_SIUL2() .....                                  | 86        |
| 2.2.5.1  | fs_ao_a5_t .....                                                   | 30        | 4.2.26   | FS_DIO_ShortToSupplySet_SIUL2() .....                               | 88        |
| 2.2.5.2  | fs_ao_test_a5_t .....                                              | 30        | <b>5</b> | <b>Invariable memory test</b> .....                                 | <b>90</b> |
| 2.2.5.3  | FS_AIO_InputSet_A5() .....                                         | 30        | 5.1      | Invariable memory test in compliance with IEC/UL standards .....    | 90        |
| 2.2.5.4  | FS_AIO_ReadResult_A5() .....                                       | 31        | 5.2      | Invariable memory test implementation .....                         | 91        |
| 2.2.6    | ADC type A7 .....                                                  | 31        | 5.2.1    | Computing of CRC value in linking phase of application .....        | 91        |
| 2.2.6.1  | fs_ao_a7_t .....                                                   | 32        | 5.2.2    | Test performed once after MCU reset .....                           | 93        |
| 2.2.6.2  | fs_ao_test_a7_t .....                                              | 32        | 5.2.3    | Runtime test .....                                                  | 93        |
| 2.2.6.3  | FS_AIO_InputSet_A7() .....                                         | 32        | 5.2.4    | FS_FLASH_C_HW32_K() .....                                           | 94        |
| 2.2.6.4  | FS_AIO_ReadResult_A7() .....                                       | 33        |          |                                                                     |           |
| 2.2.7    | ADC type A8 .....                                                  | 33        |          |                                                                     |           |

|           |                                                              |            |           |                                                             |            |
|-----------|--------------------------------------------------------------|------------|-----------|-------------------------------------------------------------|------------|
| 5.2.5     | FS_FLASH_C_HW16_K()                                          | 95         | 10.4.2    | TSI input check with stimulated inputs (signal delta check) | 129        |
| 5.2.6     | FS_FLASH_C_HW16_L()                                          | 95         | 10.4.2.1  | TSI input channel stimulation                               | 130        |
| 5.2.7     | FS_CM4_CM7_FLASH_HW32_DCP()                                  | 96         | 10.5      | TSI test implementation                                     | 131        |
| 5.2.8     | FS_CM4_CM7_FLASH_HW16()                                      | 98         | 10.5.1    | TSI input test principles                                   | 132        |
| 5.2.9     | FS_CM4_CM7_FLASH_SW16()                                      | 98         | 10.5.2    | TSI stimulated input test principles                        | 132        |
| 5.2.10    | FS_CM4_CM7_FLASH_SW32()                                      | 99         | 10.5.3    | TSI test input function call example                        | 132        |
| <b>6</b>  | <b>CPU program counter test</b>                              | <b>100</b> | 10.5.4    | FS_TSI_InputInit()                                          | 133        |
| 6.1       | CPU program counter test in compliance with IEC/UL standards | 101        | 10.5.5    | FS_TSI_InputCheckNONStimulated()                            | 133        |
| 6.2       | CPU program counter test implementation                      | 102        | 10.5.6    | FS_TSI_InputCheckStimulated()                               | 134        |
| 6.2.1     | FS_CM4_CM7_PC_Test()                                         | 102        | 10.5.7    | FS_TSI_InputStimulate()                                     | 134        |
| 6.2.2     | FS_PC_Object()                                               | 103        | 10.5.8    | FS_TSI_InputRelease()                                       | 135        |
| <b>7</b>  | <b>Variable memory test</b>                                  | <b>103</b> | 10.5.9    | FS_TSI_InputCheckNONStimulated_v6()                         | 135        |
| 7.1       | Variable memory test in compliance with IEC/UL standards     | 105        | 10.5.10   | FS_TSI_InputCheckStimulated_v6()                            | 135        |
| 7.2       | Variable memory test implementation                          | 106        | 10.5.11   | FS_TSI_InputStimulate_v6()                                  | 136        |
| 7.2.1     | FS_CM4_CM7_RAM_AfterReset()                                  | 106        | 10.5.12   | FS_TSI_InputRelease_v6()                                    | 136        |
| 7.2.2     | FS_CM4_CM7_RAM_Runtime()                                     | 107        | <b>11</b> | <b>Watchdog test</b>                                        | <b>137</b> |
| 7.2.3     | FS_CM4_CM7_RAM_CopyFromBackup()                              | 108        | 11.1      | Watchdog test in compliance with IEC/UL standards           | 137        |
| 7.2.4     | FS_CM4_CM7_RAM_CopyToBackup()                                | 109        | 11.2      | Watchdog test implementation                                | 137        |
| 7.2.5     | FS_CM4_CM7_RAM_SegmentMarchC()                               | 109        | 11.2.1    | FS_WDOG_Setup_LPTMR()                                       | 138        |
| 7.2.6     | FS_CM4_CM7_RAM_SegmentMarchX()                               | 109        | 11.2.2    | FS_WDOG_Setup_KE0XZ()                                       | 139        |
| <b>8</b>  | <b>CPU register test</b>                                     | <b>110</b> | 11.2.3    | FS_WDOG_Setup_IMX_GPT()                                     | 140        |
| 8.1       | CPU register test in compliance with IEC/UL standards        | 116        | 11.2.4    | FS_WDOG_Check()                                             | 141        |
| 8.2       | CPU register test implementation                             | 116        | 11.2.5    | FS_WDOG_Setup_WWDT_LPC_mrt()                                | 141        |
| 8.2.1     | FS_CM4_CM7_CPU_Control()                                     | 117        | 11.2.6    | FS_WDOG_Setup_WWDT_CTIMER()                                 | 142        |
| 8.2.2     | FS_CM4_CM7_CPU_ControlFpu()                                  | 117        | 11.2.7    | FS_WDOG_Check_WWDT_LPC()                                    | 143        |
| 8.2.3     | FS_CM4_CM7_CPU_Float1()                                      | 118        | 11.2.8    | FS_WDOG_Check_WWDT_LPC55SXX()                               | 144        |
| 8.2.4     | FS_CM4_CM7_CPU_Float2()                                      | 119        | 11.2.9    | FS_WDOG_Check_WWDT_MCX()                                    | 145        |
| 8.2.5     | FS_CM4_CM7_CPU_NonStackedRegister()                          | 119        | <b>12</b> | <b>Note about the source code in the document</b>           | <b>145</b> |
| 8.2.6     | FS_CM4_CM7_CPU_PrimaryMask()                                 | 120        | <b>13</b> | <b>Revision history</b>                                     | <b>146</b> |
| 8.2.7     | FS_CM4_CM7_CPU_Register()                                    | 120        |           | <b>Legal information</b>                                    | <b>147</b> |
| 8.2.8     | FS_CM4_CM7_CPU_Special()                                     | 121        |           |                                                             |            |
| 8.2.9     | FS_CM4_CM7_CPU_Special&PriorityLevels()                      | 121        |           |                                                             |            |
| 8.2.10    | FS_CM4_CM7_CPU_SPmain()                                      | 122        |           |                                                             |            |
| 8.2.11    | FS_CM4_CM7_CPU_SPprocess()                                   | 122        |           |                                                             |            |
| <b>9</b>  | <b>Stack test</b>                                            | <b>123</b> |           |                                                             |            |
| 9.1       | Stack test in compliance with IEC/UL standards               | 123        |           |                                                             |            |
| 9.2       | Linker setup                                                 | 123        |           |                                                             |            |
| 9.3       | Stack test implementation                                    | 124        |           |                                                             |            |
| 9.3.1     | FS_CM4_CM7_STACK_Init                                        | 124        |           |                                                             |            |
| 9.3.2     | FS_CM4_CM7_STACK_Test                                        | 125        |           |                                                             |            |
| <b>10</b> | <b>TSI tests</b>                                             | <b>125</b> |           |                                                             |            |
| 10.1      | TSI signal shorts tests                                      | 126        |           |                                                             |            |
| 10.2      | TSI input test                                               | 126        |           |                                                             |            |
| 10.2.1    | TSI input electrode disconnected (open pin) tests            | 126        |           |                                                             |            |
| 10.3      | Shorts or disconnection on guard sensors or shield electrode | 127        |           |                                                             |            |
| 10.4      | TSI input test architecture                                  | 128        |           |                                                             |            |
| 10.4.1    | TSI input check with non-stimulated inputs                   | 128        |           |                                                             |            |

Please be aware that important notices concerning this document and the product(s) described herein, have been included in section 'Legal information'.