文档 ID：IEC60730BCM4CM7L42UG
第 0 版，2021 年 9 月 30 日

IEC60730_B_CM4_CM7_Library_UG_v4_2

IEC60730B 库用户指南

目录

第 1 章
内核自检库

IEC60730B 内核自检库提供了执行 MCU 内核自检的功能，它由执行符合国际标准（IEC 60730、IEC 60335、UL 60730、UL 1998）的测试的独立函数组成，支持 IAR、Keil 和 MCUXpresso IDE。此恩智浦内核自检库执行以下测试：

内核相关的部分

- CPU 寄存器测试
- CPU 程序计数器测试
- 可变存储区测试
- 不可变存储区测试
- 栈测试

外设相关的部分

- 时钟测试
- 数字输入/输出测试
- 模拟输入/输出测试
- 看门狗测试
- 触摸感应接口测试（仅适用于 TSIv5 外设）

每个测试的独立章节均全面描述了相应测试的测试架构、实现、测试和验证。

该库支持基于 Arm-CM4 或 Arm-CM7 内核的 MKV3x、MKV4x、MKV5x、MKE1xF、MK2xF、LPC54S0x、LPC540x、MIMXRT10xx、MIMXRT117x、MIMXRT116x、MIMX8MNx 和 MIMX8MMx 等系列。

该内核自检库以目标代码的版本发布。对于源代码，请联系恩智浦的代表。

1.1 内核自检库——目标代码

该库的目标代码分为两部分：与内核相关的部分和与外设相关的部分，以及相应的头文件。

以下是对于给定 IDE 的目标文件：

表 1. 库的目标代码

IDE	部分	目标文件
IAR	内核	*IEC60730B_M4_M7_IAR_v4_1.a*
	外设	*IEC60730B_M4_M7_COM_IAR_v4_2.a*
Keil	内核	*IEC60730B_M4_M7_Keil_v4_1.lib*
	外设	*IEC60730B_M4_M7_COM_Keil_v4_2.lib*
MCUX	内核	*libIEC60730B_M4_M7_MCUX_v4_1.a*
	外设	*libIEC60730B_M4_M7_COM_MCUX_v4_2.a*

1.2 内核自检库——源代码

该库名为 IEC60730B_CM4_CM7。主要头文件为 *iec60730b.h* 和 *iec60730b_core.h*。*iec60730b_types.h* 中定义了此库所需的所有数据类型。

每个源文件（ **.c* 或**.S* ）都有一个相应的头文件（ **.h* ）。

表 2. 库项目列表

文件名	测试类型	函数名称	函数大小[字节]	函数 大约 运行时间 [µs]
iec60730b.h	库的头文件	-		
iec60730b_core.h	与内核相关的 库的头文件	-		
iec60730b_types.h	库的数据 类型	-		
asm_mac_common.h	通用 汇编 指令	-		
iec60730b_aio.c	模拟 I/O 测试	FS_AIO_InputInit_A2346()	18[1]	0,53[1]
	模拟 I/O 测试	FS_AIO_InputInit_A7()	22[2]	0,41[2]
	模拟 I/O 测试	FS_AIO_InputInit_A1()	38[6]	13,2[6]
	模拟 I/O 测试	FS_AIO_InputInit_A5()	22[7]	10,97[7]
	模拟 I/O 测试	FS_AIO_InputTrigger()	12[1]	0.23[1]
	模拟 I/O 测试	FS_AIO_InputSet_A1()	98[6]	14,9[6]
	模拟 I/O 测试	FS_AIO_InputSet_A23()	48[1]	0,59[1]
	模拟 I/O 测试	FS_AIO_InputSet_A4()	48[3]	0.78[3]
	模拟 I/O 测试	FS_AIO_InputSet_A7()	114[2]	1.88[2]
	模拟 I/O 测试	FS_AIO_InputSet_A5()	116[7]	22,03[7]
	模拟 I/O 测试	FS_AIO_InputSet_A6()	44[4]	0,14[4]
	模拟 I/O 测试	FS_AIO_InputCheck_A23()	112[1]	1,15[1]
	模拟 I/O 测试	FS_AIO_InputCheck_A4()	112[3]	0,58[3]
	模拟 I/O 测试	FS_AIO_InputCheck_A7()	136[2]	0,55[2]
	模拟 I/O 测试	FS_AIO_InputCheck_A5()	136[7]	33,63[7]
	模拟 I/O 测试	FS_AIO_InputCheck_A1()	120[6]	17,7[6]
	模拟 I/O 测试	FS_AIO_InputCheck_A6()	112[4]	1,71[4]
iec60730b_clock.c	时钟测试	FS_CLK_Check()	44[1]	0.51[1]
	时钟测试	FS_CLK_Init()	8[1]	0.23[1]
	时钟测试	FS_CLK_LPTMR()	12[1]	1.68[1]

表格在下一页继续...

表 2. 库项目列表（续）

文件名	测试类型	函数名称	函数大小[字节]	函数 大约 运行时间 [µs]
	时钟测试	FS_CLK_RTC()	-	-
	时钟测试	FS_CLK_GPT()	12[4]	2.16[4]
	时钟测试	FS_CLK_WKT_LPC()	-	
	时钟测试	FS_CLK_CTIMER_LPC()	-	-
iec60730b_dio.c	数字 I/O 测试	FS_DIO_Input()	-	-
	数字 I/O 测试	FS_DIO_Output()	126[1]	17,4 (delay=100)[1]
	数字 I/O 测试	FS_DIO_Output_IMXRT()	124[4]	94,33 (delay=3500)[4]
	数字 I/O 测试	FS_DIO_Output_IMX8M()	130[5]	71,1 (delay=2000)[5]
	数字 I/O 测试	FS_DIO_Output_LPC()	156[7]	34,65 (delay=75)[7]
iec60730b_dio_ext.c	扩展的数字 I/O 测试	FS_DIO_InputExt()	228[1]	1,78[1]
	扩展的数字 I/O 测试	FS_DIO_ShortToSupplySet()	152[1]	1,24[1]
	扩展的数字 I/O 测试	FS_DIO_ShortToAdjSet()	288[1]	2,23[1]
	扩展的数字 I/O 测试	FS_DIO_InputExt_IMXRT()	278[4]	0,86[4]
	扩展的数字 I/O 测试	FS_DIO_ShortToSupplySet_IMXRT()	130[4]	2,00[4]
	扩展的数字 I/O 测试	FS_DIO_ShortToAdjSet_IMXRT()	232[4]	1,76[4]
	扩展的数字 I/O 测试	FS_DIO_InputExt_IMX8M()	294[5]	13,77[5]
	扩展的数字 I/O 测试	FS_DIO_ShortToSupplySet_IMX8M()	156[5]	13,21[5]
	扩展的数字 I/O 测试	FS_DIO_ShortToAdjSet_IMX8M()	280[5]	23,25[5]
	扩展的数字 I/O 测试	FS_DIO_InputExt_LPC()	180[7]	21,04[7]
	扩展的数字 I/O 测试	FS_DIO_ShortToSupplySet_LPC()	130[7]	21,79[7]

表格在下一页继续...

表 2. 库项目列表（续）

文件名	测试类型	函数名称	函数大小[字节]	函数大约运行时间[µs]
	扩展的数字 I/O 测试	FS_DIO_ShortToAdjSet_LPC()	254[7]	35,3[7]
iec60730b_tsi.c	触摸感应接口测试	FS_TSI_InputInit()	-	-
	触摸感应接口测试	FS_TSI_InputStimulate()	-	-
	触摸感应接口测试	FS_TSI_InputRelease()	-	-
	触摸感应接口测试	FS_TSI_InputCheckNONStimulated()	-	-
	触摸感应接口测试	FS_TSI_InputCheckStimulated()	-	-
iec60730b_invariable_memory.c	不可变存储区测试（Flash）	FS_FLASH_C_HW16_K()	参见专门阐述函数的章节	
	不可变存储区测试（Flash）	FS_FLASH_C_HW16_L()	参见专门阐述函数的章节	
iec60730b_cm4_cm7_flash.S	不可变存储区测试（Flash）	FS_CM4_CM7_FLASH_HW16()	参见专门阐述函数的章节	
	不可变存储区测试（Flash）	FS_CM4_FLASH_HW16_LPC()	参见专门阐述函数的章节	
	不可变存储区测试（Flash）	FS_CM4_CM7_FLASH_SW16()	参见专门阐述函数的章节	
	不可变存储区测试（Flash）	FS_CM4_CM7_FLASH_SW32()	参见专门阐述函数的章节	
iec60730b_cm4_cm7_flash_dcp.c	不可变存储区测试（Flash）	FS_CM4_CM7_FLASH_HW32_DCP()	参见专门阐述函数的章节	
iec60730b_cm4_cm7_pc.S	程序计数器测试	FS_CM4_CM7_PC_Test()	参见专门阐述函数的章节	
iec60730b_cm4_cm7_pc_object.S	程序计数器测试	FS_PC_Object()	参见专门阐述函数的章节	
iec60730b_cm4_cm7_ram.S	可变存储区测试（RAM）	FS_CM4_CM7_RAM_AfterReset()	参见专门阐述函数的章节	
	可变存储区测试（RAM）	FS_CM4_CM7_RAM_Runtime()	参见专门阐述函数的章节	
	可变存储区测试（RAM）	FS_CM4_CM7_RAM_CopyToBackup()	参见专门阐述函数的章节	

表格在下一页继续...

表 2. 库项目列表（续）

文件名	测试类型	函数名称	函数大小[字节]	函数大约运行时间[µs]
	可变存储区测试（RAM）	FS_CM4_CM7_RAM_CopyFromBackup()	参见专门阐述函数的章节	
	可变存储区测试（RAM）	FS_CM4_CM7_RAM_SegmentMarchC()	参见专门阐述函数的章节	
	可变存储区测试（RAM）	FS_CM4_CM7_RAM_SegmentMarchX()	参见专门阐述函数的章节	
iec60730b_cm4_cm7_reg.S	寄存器测试	FS_CM4_CM7_CPU_Register()	参见专门阐述函数的章节	
	寄存器测试	FS_CM4_CM7_CPU_NonStackedRegister()	参见专门阐述函数的章节	
	寄存器测试	FS_CM4_CM7_CPU_Primask()	参见专门阐述函数的章节	
	寄存器测试	FS_CM4_CM7_CPU_SPmain()	参见专门阐述函数的章节	
	寄存器测试	FS_CM4_CM7_CPU_SPprocess()	参见专门阐述函数的章节	
	寄存器测试	FS_CM4_CM7_CPU_Control()	参见专门阐述函数的章节	
	寄存器测试	FS_CM4_CM7_CPU_Special()	参见专门阐述函数的章节	
	寄存器测试	FS_CM4_CM7_CPU_Special8PriorityLevels()	参见专门阐述函数的章节	
iec60730b_cm4_cm7_reg_fpu.S	寄存器测试	FS_CM4_CM7_CPU_ControlFpu()	参见专门阐述函数的章节	
	寄存器测试	FS_CM4_CM7_CPU_Float1()	参见专门阐述函数的章节	
	寄存器测试	FS_CM4_CM7_CPU_Float2()	参见专门阐述函数的章节	
iec60730b_cm4_cm7_stack.S	栈测试	FS_CM4_CM7_STACK_Init()	参见专门阐述函数的章节	
	栈测试	FS_CM4_CM7_STACK_Test()	参见专门阐述函数的章节	
iec60730b_wdog.c	看门狗测试	FS_WDOG_Setup_LPTMR()	90[1]	运行时间取决于 WDOG 超时时间
	看门狗测试	FS_WDOG_Setup_KE0XZ()	-	运行时间取决于 WDOG 超时时间
	看门狗测试	FS_WDOG_Setup_IMX_GPT()	64[5]	运行时间取决于 WDOG 超时时间
	看门狗测试	FS_WDOG_Setup_WWDT_LPC()	-	运行时间取决于 WDOG 超时时间

表格在下一页继续...

表 2. 库项目列表（续）

文件名	测试类型	函数名称	函数大小[字节]	函数 大约 运行时间 [µs]
	看门狗测试	FS_WDOG_Setup_WWDT_LPC_mrt()	-	运行时间 取决于 WDOG 超时时间
	看门狗测试	FS_WDOG_Check()	188[1]	1.2[1]
	看门狗测试	FS_WDOG_Check_WWDT_LPC()	-	-
	看门狗测试	FS_WDOG_Check_WWDT_LPC55SXX()	-	-

1.2.1 MIMX8MMx 的专用函数

表 3 显示了专用于 MIMX8M Mini 系列器件的函数列表。

表 3. MIMX8MMx 的专用函数

文件	适用的函数
iec60730b_clock.c	FS_CLK_Check()
	FS_CLK_Init()
	FS_CLK_GPT()
iec60730b_dio.c	FS_DIO_Output_IMX8M()
iec60730b_dio_ext.c	FS_DIO_InputExt_IMX8M()
	FS_DIO_ShortToSupplySet_IMX8M()
	FS_DIO_ShortToAdjSet_IMX8M()
iec60730b_wdog.c	FS_WDOG_Setup_IMX_GPT() refresh_index = "FS_IMX8M"
	FS_WDOG_Check() RegWide8b = "FS_WDOG_SRS_WIDE_32b"
iec60730b_cm4_cm7_flash.S	FS_CM4_CM7_FLASH_SW16()
	FS_CM4_CM7_FLASH_SW32()
iec60730b_cm4_cm7_pc.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_ram.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_reg.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_stack.S	所有 CM4/CM7 器件通用

1.2.2 MIMX8MNx 的专用函数

表 4 显示了专用于 MIMX8M Nano 系列器件的函数列表。

表 4. MIMX8MNx 的专用函数

文件	适用的函数
iec60730b_clock.c	FS_CLK_Check()
	FS_CLK_Init()
	FS_CLK_GPT()
iec60730b_dio.c	FS_DIO_Output_IMX8M()
iec60730b_dio_ext.c	FS_DIO_InputExt_IMX8M()
	FS_DIO_ShortToSupplySet_IMX8M()
	FS_DIO_ShortToAdjSet_IMX8M()
iec60730b_wdog.c	FS_WDOG_Setup_IMX_GPT() refresh_index = "FS_IMX8M"
	FS_WDOG_Check() RegWide8b = "FS_WDOG_SRS_WIDE_32b"
iec60730b_cm4_cm7_flash.S	FS_CM4_CM7_FLASH_SW16()
	FS_CM4_CM7_FLASH_SW32()
iec60730b_cm4_cm7_pc.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_ram.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_reg.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_stack.S	所有 CM4/CM7 器件通用

1.2.3 MIMXRT10xx 的专用函数

表 5 显示了专用于 MIMXRT10xx 系列器件的函数列表。

表 5. MIMXRT10xx 的专用函数

文件	适用的函数
iec60730b_aio.c	FS_AIO_InputInit_A2346()
	FS_AIO_InputTrigger()
	FS_AIO_InputSet_A6()
	FS_AIO_InputCheck_A6()
iec60730b_clock.c	FS_CLK_Check()
	FS_CLK_Init()
	FS_CLK_GPT()
iec60730b_dio.c	FS_DIO_Output_IMXRT()
iec60730b_dio_ext.c	FS_DIO_InputExt_IMXRT()
	FS_DIO_ShortToSupplySet_IMXRT()
	FS_DIO_ShortToAdjSet_IMXRT()
iec60730b_wdog.c	FS_WDOG_Setup_IMX_GPT() refresh_index = "FS_IMXRT"

表格在下一页继续...

表 5. MIMXRT10xx 的专用函数（续）

文件	适用的函数
	FS_WDOG_Check() RegWide8b = "FS_WDOG_SRS_WIDE_32b"
iec60730b_cm4_cm7_flash.S	FS_CM4_CM7_FLASH_HW32_DCP()
	FS_CM4_CM7_FLASH_SW16()
	FS_CM4_CM7_FLASH_SW32()
iec60730b_cm4_cm7_pc.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_ram.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_reg.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_stack.S	所有 CM4/CM7 器件通用

1.2.4 MIMXRT117x/116x 的专用函数

表 6 显示了专用于 MIMXRT117x 和 MIMXRT116x 系列器件的函数列表。

表 6. MIMXRT117x/116x 的专用函数

文件	适用的函数
iec60730b_aio.c	FS_AIO_InputInit_A1()
	FS_AIO_InputTrigger()
	FS_AIO_InputSet_A1()
	FS_AIO_InputCheck_A1()
iec60730b_clock.c	FS_CLK_Check()
	FS_CLK_Init()
	FS_CLK_GPT()
iec60730b_dio.c	FS_DIO_Output_IMXRT()
iec60730b_dio_ext.c	FS_DIO_InputExt_IMXRT()
	FS_DIO_ShortToSupplySet_IMXRT()
	FS_DIO_ShortToAdjSet_IMXRT()
iec60730b_wdog.c	FS_WDOG_Setup_IMX_GPT() refresh_index = "FS_IMXRTWDOG"
	FS_WDOG_Check() RegWide8b = "FS_WDOG_SRS_WIDE_32b"
iec60730b_cm4_cm7_flash.S	FS_CM4_CM7_FLASH_SW16()
	FS_CM4_CM7_FLASH_SW32()
iec60730b_cm4_cm7_pc.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_ram.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_reg.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_stack.S	所有 CM4/CM7 器件通用

1.2.5 MK2xF 的专用函数

表 7 显示了专用于 MK2xF 器件的函数列表。

表 7. MK2xF 的专用函数

文件	适用的函数
iec60730b_aio.c	FS_AIO_InputInit_A2346()
	FS_AIO_InputTrigger()
	FS_AIO_InputSet_A23()
	FS_AIO_InputCheck_A23()
iec60730b_clock.c	FS_CLK_Check()
	FS_CLK_Init()
	FS_CLK_LPTMR()
iec60730b_dio.c	FS_DIO_Output()
iec60730b_dio_ext.c	FS_DIO_InputExt()
	FS_DIO_ShortToSupplySet()
	FS_DIO_ShortToAdjSet()
iec60730b_wdog.c	FS_WDOG_Setup_LPTMR() refresh_index = "FS_KINETIS_WDOG"
	FS_WDOG_Check() RegWide8b = "FS_WDOG_SRS_WIDE_8b"
iec60730b_invariable_memory.c	FS_FLASH_C_HW16_K()
iec60730b_cm4_cm7_flash.S	FS_CM4_CM7_FLASH_HW16()
	FS_CM4_CM7_FLASH_SW16()
	FS_CM4_CM7_FLASH_SW32()
iec60730b_cm4_cm7_pc.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_ram.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_reg.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_stack.S	所有 CM4/CM7 器件通用

1.2.6 MKE1xF 的专用函数

表 8 显示了专用于 MKE1xF 器件的函数列表。

表 8. MKE1xF 的专用函数

文件	适用的函数
iec60730b_aio.c	FS_AIO_InputInit_A2346()
	FS_AIO_InputTrigger()
	FS_AIO_InputSet_A4()
	FS_AIO_InputCheck_A4()

表格在下一页继续...

表 8. MKE1xF 的专用函数 (续)

文件	适用的函数
iec60730b_clock.c	FS_CLK_Check()
	FS_CLK_Init()
	FS_CLK_LPTMR()
iec60730b_dio.c	FS_DIO_Output()
iec60730b_dio_ext.c	FS_DIO_InputExt()
	FS_DIO_ShortToSupplySet()
	FS_DIO_ShortToAdjSet()
iec60730b_wdog.c	FS_WDOG_Setup_LPTMR() refresh_index = "FS_WDOG32"
	FS_WDOG_Check() RegWide8b = "FS_WDOG_SRS_WIDE_32b"
iec60730b_invariable_memory.c	FS_FLASH_C_HW16_K()
iec60730b_cm4_cm7_flash.S	FS_CM4_CM7_FLASH_HW16()
	FS_CM4_CM7_FLASH_SW16()
	FS_CM4_CM7_FLASH_SW32()
iec60730b_cm4_cm7_pc.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_ram.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_reg.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_stack.S	所有 CM4/CM7 器件通用

1.2.7 MKV3x 的专用函数

表 9 显示了专用于 MKV3x 器件的函数列表。

表 9. MKV3x 的专用函数

文件	适用的函数
iec60730b_aio.c	FS_AIO_InputInit_A2346()
	FS_AIO_InputTrigger()
	FS_AIO_InputSet_A23()
	FS_AIO_InputCheck_A23()
iec60730b_clock.c	FS_CLK_Check()
	FS_CLK_Init()
	FS_CLK_LPTMR()
iec60730b_dio.c	FS_DIO_Output()
iec60730b_dio_ext.c	FS_DIO_InputExt()
	FS_DIO_ShortToSupplySet()

表格在下一页继续...

表 9. MKV3x 的专用函数（续）

文件	适用的函数
	FS_DIO_ShortToAdjSet()
iec60730b_wdog.c	FS_WDOG_Setup_LPTMR() refresh_index = "FS_KINETIS_WDOG"
	FS_WDOG_Check() RegWide8b = "FS_WDOG_SRS_WIDE_8b"
iec60730b_invariable_memory.c	FS_FLASH_C_HW16_K()
iec60730b_cm4_cm7_flash.S	FS_CM4_CM7_FLASH_HW16()
	FS_CM4_CM7_FLASH_SW16()
	FS_CM4_CM7_FLASH_SW32()
iec60730b_cm4_cm7_pc.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_ram.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_reg.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_stack.S	所有 CM4/CM7 器件通用

1.2.8 MKV4x 的专用函数

表 10 显示了专用于 MKV4x 器件的函数列表。

表 10. MKV4x 的专用函数

文件	适用的函数
iec60730b_aio.c	FS_AIO_InputInit_A7()
	FS_AIO_InputTrigger()
	FS_AIO_InputSet_A7()
	FS_AIO_InputCheck_A7()
iec60730b_clock.c	FS_CLK_Check()
	FS_CLK_Init()
	FS_CLK_LPTMR()
iec60730b_dio.c	FS_DIO_Output()
iec60730b_dio_ext.c	FS_DIO_InputExt()
	FS_DIO_ShortToSupplySet()
	FS_DIO_ShortToAdjSet()
iec60730b_wdog.c	FS_WDOG_Setup_LPTMR() refresh_index = "FS_KINETIS_WDOG"
	FS_WDOG_Check() RegWide8b = "FS_WDOG_SRS_WIDE_8b"
iec60730b_invariable_memory.c	FS_FLASH_C_HW16_K()
iec60730b_cm4_cm7_flash.S	FS_CM4_CM7_FLASH_HW16()
	FS_CM4_CM7_FLASH_SW16()

表格在下一页继续...

表 10. MKV4x 的专用函数（续）

文件	适用的函数
	FS_CM4_CM7_FLASH_SW32()
iec60730b_cm4_cm7_pc.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_ram.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_reg.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_stack.S	所有 CM4/CM7 器件通用

1.2.9 MKV5x 的专用函数

表 11 显示了专用于 MKV5x 器件的函数列表。

表 11. MKV5x 的专用函数

文件	适用的函数
iec60730b_aio.c	FS_AIO_InputInit_A2346()
	FS_AIO_InputTrigger()
	FS_AIO_InputSet_A23()
	FS_AIO_InputCheck_A23()
iec60730b_clock.c	FS_CLK_Check()
	FS_CLK_Init()
	FS_CLK_LPTMR()
iec60730b_dio.c	FS_DIO_Output()
iec60730b_dio_ext.c	FS_DIO_InputExt()
	FS_DIO_ShortToSupplySet()
	FS_DIO_ShortToAdjSet()
iec60730b_wdog.c	FS_WDOG_Setup_LPTMR() refresh_index = "FS_KINETIS_WDOG"
	FS_WDOG_Check() RegWide8b = "FS_WDOG_SRS_WIDE_8b"
iec60730b_invariable_memory.c	FS_FLASH_C_HW16_K()
iec60730b_cm4_cm7_flash.S	FS_CM4_CM7_FLASH_HW16()
	FS_CM4_CM7_FLASH_SW16()
	FS_CM4_CM7_FLASH_SW32()
iec60730b_cm4_cm7_pc.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_ram.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_reg.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_stack.S	所有 CM4/CM7 器件通用

1.2.10 LPC54S0x/LPC540x 的专用函数

表 12 显示了专用于 LPC54S0x/LPC540x 器件的函数列表。

表 12. LPC54S0x/LPC540x 的专用函数

文件	适用的函数
iec60730b_aio.c	FS_AIO_InputInit_A5()
	FS_AIO_InputTrigger()
	FS_AIO_InputSet_A5()
	FS_AIO_InputCheck_A5()
iec60730b_clock.c	FS_CLK_Check()
	FS_CLK_Init()
	FS_CLK_CTIMER_LPC()
iec60730b_dio.c	FS_DIO_Output_LPC()
iec60730b_dio_ext.c	FS_DIO_InputExt_LPC()
	FS_DIO_ShortToSupplySet_LPC()
	FS_DIO_ShortToAdjSet_LPC()
iec60730b_wdog.c	FS_WDOG_Setup_WWDT_LPC()
	FS_WDOG_Check_WWDT_LPC()
iec60730b_cm4_cm7_flash.S	FS_CM4_CM7_FLASH_SW16()
	FS_CM4_CM7_FLASH_SW32()
iec60730b_invariable_memory.c	FS_FLASH_C_HW16_L()
iec60730b_cm4_cm7_pc.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_ram.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_reg.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_stack.S	所有 CM4/CM7 器件通用

1.2.11 MK32L3 CM4 的专用函数

表 13 显示了专用于 MK32L3 CM4 内核的函数列表。

表 13. 针对 CM4 内核的 MK32L3 专用函数

文件	适用的函数
iec60730b_aio.c	FS_AIO_InputInit_A1()
	FS_AIO_InputTrigger()
	FS_AIO_InputSet_A1()
	FS_AIO_InputCheck_A1()
iec60730b_clock.c	FS_CLK_Check()

表格在下一页继续...

表 13. 针对 CM4 内核的 MK32L3 专用函数（续）

文件	适用的函数
	FS_CLK_Init()
	FS_CLK_LPTMR()
iec60730b_dio.c	FS_DIO_Output()
iec60730b_dio_ext.c	FS_DIO_InputExt()
	FS_DIO_ShortToSupplySet()
	FS_DIO_ShortToAdjSet()
iec60730b_wdog.c	FS_WDOG_Setup_LPTMR() refresh_index = "FS_KINETIS_WDOG"
	FS_WDOG_Check() RegWide8b = "FS_WDOG_SRS_WIDE_32b"
iec60730b_cm4_cm7_flash.S	函数在专门章节中描述
iec60730b_cm4_cm7_pc.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_ram.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_reg.S	所有 CM4/CM7 器件通用
iec60730b_cm4_cm7_Stack.S	所有 CM4/CM7 器件通用

1.3 函数性能的测量

本节包含关于函数信息量的大小和大约执行时间的备注。下面列表中的数字被用作相应章节的备注链接：

1. 函数参数在 MKV31 上测量，时钟频率为 80MHz。
2. 函数参数在 MKV46 上测量，时钟频率为 80MHz。
3. 函数参数在 MKE18F 上测量，时钟频率为 100MHz。
4. 函数参数在 MIMXRT1050 上测量，时钟频率为 600MHz。
5. 函数参数在 MIMX8MN 上测量，时钟频率为 600MHz。
6. 函数参数在 MIMXRT1170 上测量，时钟频率为 996 MHz。
7. 函数参数在 LPC54S018M 上测量，时钟频率为 96MHz。

第 2 章
模拟输入/输出（IO）测试

模拟 IO 测试程序执行处理器模拟 IO 接口的置信性检查。模拟 IO 测试可以在 MCU 复位后执行一次，也可以在运行时执行。

如果出现模拟 IO 错误，则通过特定的 FAIL 返回确保安全错误的识别。将测试函数的返回值与预期值进行比较。如果等于 FAIL 返回，则跳转到安全错误处理函数。安全错误处理函数可能是特定于应用的，不是库的一部分，主要目的是将应用置于安全状态。

模拟 IO 测试的原理基于顺序执行，将特定模拟电平连接到定义的模拟输入。测试函数检查转换后的值是否在允许范围内。该测试检查模拟输入接口和三个参考值：高参考电压、低参考电压和带隙电压。请参阅器件规范文档以设置正确的值。此模拟 IO 测试的框图如下图所示：

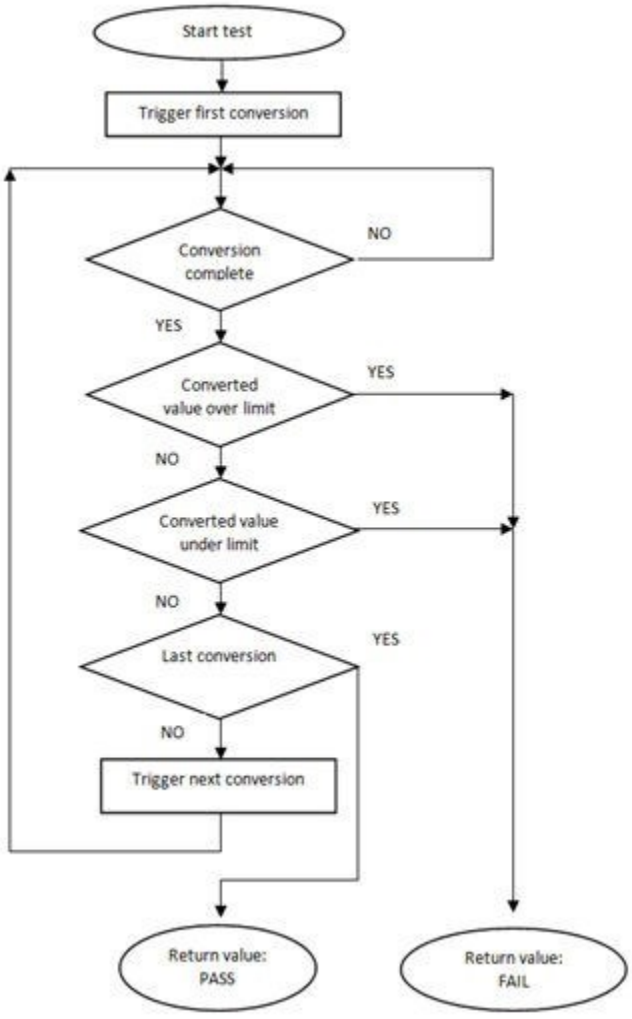

图 1. 模拟输入测试框图

2.1 符合 IEC/UL 标准的模拟输入/输出测试

所执行的过载测试满足 IEC 60730-1、IEC 60335、UL 60730 和 UL 1998 标准的安全要求，参见下表：

表 14. 符合 IEC 和 UL 标准的模拟输入/输出测试

测试	元器件	故障/错误	软件/硬件类别	可接受的测量
输入/输出外设	7. 输入/输出外设（7.2 – A/D 转换）	异常操作	B/R.1	置信性检查

2.2 模拟输入/输出测试的实现

该模拟 IO 测试的测试函数在 *iec60730b_aio.c* 文件中，写作“C”函数。带函数原型的头文件是 *iec60730b_aio.h*。*Iec60730b.h* 和 *iec60730b_types.h* 是本安全库的常用头文件。

调用以下函数测试模拟输入：

- *FS_AIO_InputInit_A1*、*FS_AIO_InputInit_A2346*、*FS_AIO_InputInit_A5*、*FS_AIO_InputInit_A7*
- *FS_AIO_InputTrigger()*
- *FS_AIO_InputSet_A1*、*FS_AIO_InputSet_A23*、*FS_AIO_InputSet_A4*、*FS_AIO_InputSet_A5*、*FS_AIO_InputSet_A6*、*FS_AIO_InputSet_A7*
- *FS_AIO_InputCheck_A1*、*FS_AIO_InputCheck_A23*、*FS_AIO_InputCheck_A4*、*FS_AIO_InputCheck_A5*、*FS_AIO_InputCheck_A6*、*FS_AIO_InputCheck_A7*

此模拟输入测试基于三个已知电压值的模拟输入的转换，并检查转换后的值是否符合规定的限制。通常，该限制应约为期望参考值的 10%。测试由 *FS_AIO_InputTrigger()* 函数触发，分为三个部分：初始化、测试执行和测试结束。

在所有支持的器件中，ADC 模块对测试中涉及的寄存器的排列略有不同。因此，为 ADC 模块创建了一个独立的检查函数。请参阅内核自检库–源代码版本（针对相应器件的专用函数）。

以下是函数调用的示例：

参数的配置

```
#define TESTED_ADC ADC0
#define ADC_RESOLUTION 12
#define ADC_MAX ((1<<(ADC_RESOLUTION))-1)
#define ADC_REFERENCE 3.06
#define ADC_BANDGAP_LEVEL 1.7
#define ADC_BANDGAP_LEVEL_RAW (((ADC_BANDGAP_LEVEL)*(ADC_MAX))/(ADC_REFERENCE))
#define ADC_DEVIATION_PERCENT 10
#define ADC_MIN_LIMIT(val) (((val) * (100 - ADC_DEVIATION_PERCENT)) / 100)
#define ADC_MAX_LIMIT(val) (((val) * (100 + ADC_DEVIATION_PERCENT)) / 100)
#define FS_CFG_AIO_CHANNELS_CNT 3
#define FS_CFG_AIO_CHANNELS_LIMITS_INIT\
{\
 {ADC_MIN_LIMIT(0), ADC_MAX_LIMIT(10)}, \
 {ADC_MIN_LIMIT(ADC_MAX), ADC_MAX_LIMIT(ADC_MAX)},\
 {ADC_MIN_LIMIT(ADC_BANDGAP_LEVEL_RAW), ADC_MAX_LIMIT(ADC_BANDGAP_LEVEL_RAW)}\
```

```
}
#define FS_CFG_AIO_CHANNELS_INIT {30, 29, 27}
```

变量的定义

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

测试的初始化

```
FS_AIO_InputInit_A2346(&aio_Str, (fs_aio_limits_t*)FS_ADC_Limits, (unsigned
char*)FS_ADC_inputs, FS_CFG_AIO_CHANNELS_CNT);
FS_AIO_InputTrigger(&aio_Str);
```

测试

```
for(uint8_t i=0;i<4;i++)
{
 psSafetyCommon->AIO_test_result = FS_AIO_InputCheck_A23(&aio_Str, (unsigned
long*)TESTED_ADC);
 switch(psSafetyCommon->AIO_test_result)
 {
 case FS_AIO_START:
 FS_AIO_InputSet_A23(&aio_Str, (unsigned long*)TESTED_ADC);
 break;
 case FS_FAIL_AIO:
 psSafetyCommon->ui32SafetyErrors |= AIO_TEST_ERROR;
 SafetyErrorHandling(psSafetyCommon);
 break;
 case FS_AIO_INIT:
 FS_AIO_InputTrigger(&aio_Str);
 break;
 case FS_PASS:
 FS_AIO_InputTrigger(&aio_Str);
 break;
 default:
 _ asm("NOP");
 break;
 }
}
```

2.2.1 FS_AIO_InputTrigger()

此函数设置模拟输入测试，以开始执行测试（为 pObj 设置状态 FS_AIO_START）。

函数原型：

*void FS_AIO_InputTrigger(fs_aio_test_t *pObj);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

函数输出：

空

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

2.2.2 FS_AIO_InputInit_A1()

此函数初始化模拟输入测试的实例。

函数原型：

*void FS_AIO_InputInit_A1(fs_aio_test_t *pObj, fs_aio_limits_t *pLimits, uint8_t *pInputs, uint8_t cntMax, uint8_t cmdBuffer, uint8_t *pSideSelect, uint8_t TriggerEvent)*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pLimits* - 此输入参数是指向测试中使用的限制数组的指针。

**pInputs* - 此输入参数是指向测试中使用的输入数字数组的指针。

cntMax - 此输入参数是输入和限制数组的大小。

cmdBuffer - 指定转换的命令缓冲区。

pSideSelect - 通道侧组 0=A 侧，1=B 侧

TriggerEvent - 软件触发事件的编号（顺序）

函数输出：

空

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

2.2.3 FS_AIO_InputSet_A1()

此函数执行 AIO 测试顺序的第一部分。该部分设置 ADC 输入通道。当 ADC 转换器配置为软件触发器时，此函数也会触发转换。状态更改为 FS_AIO_PROGRESS。当 ADC 模块空闲并准备好进行下一次转换时，可以调用此函数。

函数原型：

*FS_RESULT FS_AIO_InputSet_A1(fs_aio_test_t *pObj, fs_aio_a1_t *pAdc);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pAdc* - 此输入参数是指向模拟转换器的指针。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_AIO_PROGRESS* - 要求的返回值。这意味着输入已设定。

如果返回任何其他值，则该函数无效。

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

2.2.4 FS_AIO_InputCheck_A1()

此函数执行 AIO 测试顺序的第二部分。该部分读取转换后的模拟值，并检查该值是否符合预定义的限制。此函数仅在 pObj->state==FS_AIO_PROGRESS 时读取转换后的值。当该函数上报 FS_AIO_PASS 或 FS_FAIL_AIO 时，测试结束。

函数原型：

*FS_RESULT FS_AIO_InputCheck_A1(fs_aio_test_t *pObj, fs_aio_a1_t *pAdc);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pAdc* - 此输入参数是指向模拟转换器的指针。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS* - 测试执行成功（所有通道均被测试）。
- *FS_FAIL_AIO* - 转换后的值不符合限制。
- *FS_AIO_START* - 一次成功的转换和下一次转换的设置输入。
- *FS_AIO_PROGRESS* - 输入尚未转换。
- *FS_AIO_INIT* - 该函数无效。

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

2.2.5 FS_AIO_InputInit_A2346()

此函数初始化模拟输入测试的一个实例。

函数原型：

*void FS_AIO_InputInit_A2346(fs_aio_test_t *pObj, fs_aio_limits_t *pLimits, uint8_t *pInputs, uint8_t cntMax);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pLimits* - 此输入参数是指向测试中使用的限制数组的指针。

**pInputs* - 此输入参数是指向测试中使用的输入数字数组的指针。

**cntMax* - 此输入参数是输入和限制数组的大小。

函数输出：

空

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

2.2.6 FS_AIO_InputSet_A23()

此函数执行 AIO 测试顺序的第一部分。该部分设置 ADC 输入通道。当 ADC 转换器配置为软件触发器时，此函数也会触发转换。状态更改为 FS_AIO_PROGRESS。当 ADC 模块空闲并准备好进行下一次转换时，可以调用此函数。

函数原型：

*FS_RESULT FS_AIO_InputSet_A23(fs_aio_test_t *pObj, fs_aio_a23_t *pAdc);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pAdc* - 此输入参数是指向模拟转换器的指针。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_AIO_PROGRESS* - 要求的返回值。这意味着输入已设定。

如果返回任何其他值，则该函数无效。

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

2.2.7 FS_AIO_InputCheck_A23()

此函数执行 AIO 测试顺序的第二部分。该部分读取转换后的模拟值，并检查该值是否符合预定义的限制。此函数仅在 pObj->state==FS_AIO_PROGRESS 时读取转换后的值。当该函数上报 FS_AIO_PASS 或 FS_FAIL_AIO 时，测试结束。

函数原型：

*FS_RESULT FS_AIO_InputCheck_A23(fs_aio_test_t *pObj, fs_aio_a23_t *pAdc);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pAdc* - 此输入参数是指向模拟转换器的指针。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS* - 测试执行成功（所有通道均被测试）。
- *FS_FAIL_AIO* - 转换后的值不符合限制。
- *FS_AIO_START* - 一次成功的转换和下一次转换的设置输入。
- *FS_AIO_PROGRESS* - 输入尚未转换。
- *FS_AIO_INIT* - 该函数无效。

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

2.2.8 FS_AIO_InputSet_A4()

此函数执行 AIO 测试顺序的第一部分。该部分设置 ADC 输入通道。当 ADC 转换器配置为软件触发器时，此函数也会触发转换。状态更改为 FS_AIO_PROGRESS。当 ADC 模块空闲并准备好进行下一次转换时，可以调用此函数。

函数原型：

*FS_RESULT FS_AIO_InputSet_A4(fs_aio_test_t *pObj, fs_aio_a4_t *pAdc);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pAdc* - 此输入参数是指向模拟转换器的指针。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_AIO_PROGRESS* - 要求的返回值。这意味着输入已设定。

如果返回任何其他值，则该函数无效。

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

2.2.9 FS_AIO_InputCheck_A4()

此函数执行 AIO 测试顺序的第二部分。该部分读取转换后的模拟值，并检查该值是否符合预定义的限制。此函数仅在 pObj->state==FS_AIO_PROGRESS 时读取转换后的值。当该函数上报 FS_AIO_PASS 或 FS_FAIL_AIO 时，测试结束。

函数原型：

*FS_RESULT FS_AIO_InputCheck_A4(fs_aio_test_t *pObj, fs_aio_a4_t *pAdc);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pAdc* - 此输入参数是指向模拟转换器的指针。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS* - 测试执行成功（所有通道均被测试）。
- *FS_FAIL_AIO* - 转换后的值不符合限制。
- *FS_AIO_START* - 一次成功的转换和下一次转换的设置输入。
- *FS_AIO_PROGRESS* - 输入尚未转换。
- *FS_AIO_INIT* - 该函数无效。

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

2.2.10 FS_AIO_InputSet_A6()

此函数执行 AIO 测试顺序的第一部分。该部分设置 ADC 输入通道。当 ADC 转换器配置为软件触发器时，此函数也会触发转换。状态更改为 FS_AIO_PROGRESS。当 ADC 模块空闲并准备好进行下一次转换时，可以调用此函数。在 i.MXRT 器件上，只有通道 0（ADCx->HC[0]）可用于软件触发。有关更多信息，请参阅参考手册。

函数原型：

*FS_RESULT FS_AIO_InputSet_A6(fs_aio_test_t *pObj, fs_aio_a6_t *pAdc);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pAdc* - 此输入参数是指向模拟转换器的指针。

函数输出：

typedef uint32_t FS_RESULT;

FS_AIO_PROGRESS - 要求的返回值。这意味着输入已设定。

如果返回任何其他值，则该函数无效。

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

2.2.11 FS_AIO_InputCheck_A6()

此函数执行 AIO 测试顺序的第二部分。该部分读取转换后的模拟值，并检查该值是否符合预定义的限制。此函数仅在 pObj->state==FS_AIO_PROGRESS 时读取转换后的值。当该函数上报 FS_AIO_PASS 或 FS_FAIL_AIO 时，测试结束。此函数只能用于软件触发的模拟 I/O 测试。

函数原型：

FS_RESULT FS_AIO_InputCheck_A6(fs_aio_test_t pObj, fs_aio_a6_t *pAdc);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pAdc* - 此输入参数是指向模拟转换器的指针。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS* - 测试执行成功（所有通道均被测试）。
- *FS_FAIL_AIO* - 转换后的值不符合限制。
- *FS_AIO_START* - 一次成功的转换和下一次转换的设置输入。
- *FS_AIO_PROGRESS* - 输入尚未转换。
- *FS_AIO_INIT* - 该函数无效。

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

2.2.12 FS_AIO_InputInit_A5()

此函数初始化模拟输入测试的一个实例。

函数原型：

*void FS_AIO_InputInit_A5(fs_aio_test_t *pObj, fs_aio_limits_t *pLimits, uint8_t *pInputs, uint8_t cntMax, uint8_t sequence);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pLimits* - 此输入参数是指向测试中使用的限制数组的指针。

**pInputs* - 此输入参数是指向测试中使用的输入数字数组的指针。

cntMax - 此输入参数是输入和限制数组的大小。

sequence - 此输入参数是所用序列的索引。

函数输出：

空

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

2.2.13 FS_AIO_InputSet_A5()

此函数执行 AIO 测试顺序的第一部分。该部分设置 ADC 输入通道。当 ADC 转换器配置为软件触发器时，此函数也会触发转换。状态更改为 FS_AIO_PROGRESS。当 ADC 模块空闲并准备好进行下一次转换时，可以调用此函数。

函数原型：

*FS_RESULT FS_AIO_InputSet_A5(fs_aio_test_t *pObj, fs_aio_a5_t *pAdc);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pAdc* - 此输入参数是指向模拟转换器的指针。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_AIO_PROGRESS* - 要求的返回值。这意味着输入已设定。

如果返回任何其他值，则该函数无效。

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

2.2.14 FS_AIO_InputCheck_A5()

此函数执行 AIO 测试顺序的第二部分。该部分读取转换后的模拟值，并检查该值是否符合预定义的限制。此函数仅在 pObj->state==FS_AIO_PROGRESS 时读取转换后的值。当该函数上报 FS_AIO_PASS 或 FS_FAIL_AIO 时，测试结束。

函数原型：

FS_RESULT FS_AIO_InputCheck_A5(fs_aio_test_t pObj, fs_aio_a5_t *pAdc);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pAdc* - 此输入参数是指向模拟转换器的指针。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS* - 测试执行成功（所有通道均被测试）。
- *FS_FAIL_AIO* - 转换后的值不符合限制。
- *FS_AIO_START* - 一次成功的转换和下一次转换的设置输入。
- *FS_AIO_PROGRESS* - 输入尚未转换。
- *FS_AIO_INIT* - 该函数无效。

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

2.2.15 FS_AIO_InputInit_A7()

此函数初始化模拟输入测试的一个实例。

函数原型：

*void FS_AIO_InputInit_A7(fs_aio_test_t *pObj, fs_aio_limits_t *pLimits, uint8_t *pInputs, uint8_t *pSamples, uint8_t cntMax);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pLimits* - 此输入参数是指向测试中使用的限制数组的指针。

**pInputs* - 此输入参数是指向测试中使用的输入数字数组的指针。

**pSamples* - 此输入参数是指向测试中使用的样本数数组的指针。

cntMax - 此输入参数是输入和限制数组的大小。

函数输出：

空

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

2.2.16 FS_AIO_InputSet_A7()

此函数执行 AIO 测试顺序的第一部分。该部分设置 ADC 输入通道。当 ADC 转换器配置为软件触发器时，此函数也会触发转换。状态更改为 FS_AIO_PROGRESS。当 ADC 模块空闲并准备好进行下一次转换时，可以调用此函数。

函数原型：

*FS_RESULT FS_AIO_InputSet_A7(fs_aio_test_t *pObj, fs_aio_a7_t *pAdc);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pAdc* - 此输入参数是指向模拟转换器的指针。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_AIO_PROGRESS* - 要求的返回值。这意味着输入已设定。

如果返回任何其他值，则该函数无效。

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

2.2.17 FS_AIO_InputCheck_A7()

此函数执行 AIO 测试顺序的第二部分。该部分读取转换后的模拟值，并检查该值是否符合预定义的限制。此函数仅在 pObj->state==FS_AIO_PROGRESS 时读取转换后的值。当该函数上报 FS_AIO_PASS 或 FS_FAIL_AIO 时，测试结束。

函数原型：

*FS_RESULT FS_AIO_InputCheck_A7(fs_aio_test_t *pObj, fs_aio_a7_t *pAdc);*

函数输入：

**pObj* - 此输入参数是指向模拟测试实例的指针。

**pAdc* - 此输入参数是指向模拟转换器的指针。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS* - 测试执行成功（所有通道均被测试）。

- *FS_FAIL_AIO* - 转换后的值不符合限制。
- *FS_AIO_START* - 一次成功的转换和下一次转换的设置输入。
- *FS_AIO_PROGRESS* - 输入尚未转换。
- *FS_AIO_INIT* -该函数无效。

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

第 3 章
时钟测试

时钟测试程序测试处理器振荡器的错误频率。时钟测试可以在 MCU 复位后执行一次，也可以在运行时执行。

如果出现时钟故障，通过特定的 FAIL 返回来确保安全错误的识别。评估测试函数的返回值。如果它等于 FAIL 返回，则应跳转到安全错误处理函数。安全错误处理函数是特定于应用的，不是库的一部分。该函数的主要目的是将应用置于安全状态。

时钟测试的原理基于对两个独立时钟源的比较。如果测试例程检测到时钟源之间的频率比发生了变化，则返回故障错误代码。测试例程在应用中使用一个定时器和一个周期性事件。此周期性事件也可以是来自另一个定时器的中断，而不是那个已经用到的定时器。

该库支持的器件具有多个定时器/计数器模块。请参阅内核自检库–源代码版本，了解适用于您的器件的函数。

时钟测试的框图如图 2 所示。

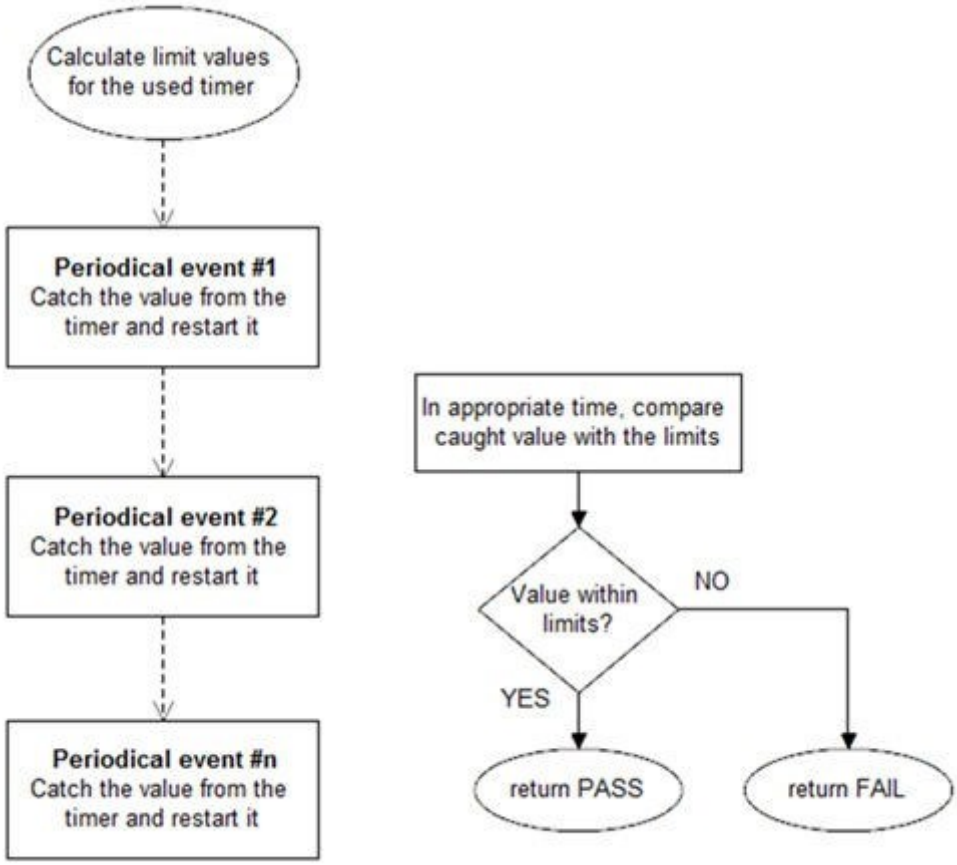

图 2. 时钟测试的框图

3.1 符合 IEC/UL 标准的时钟测试

所进行的过载测试符合 EC 60730-1、IEC 60335、UL 60730 和 UL 1998 标准的安全要求，如下表所述：

表 15. 符合 IEC 和 UL 标准的时钟测试

测试	元器件	故障/错误	软件/硬件 类别	可接受的测量
时钟测试	3.时钟	频率错误	B / R.1	频率监测

3.2 时钟测试的实现

时钟测试的测试函数在 *iec60730b_clock.c* 文件中，写作“C”函数。有函数原型的头文件是 *iec60730b_clock.h*。*Iec60730b.h* 和 *iec60730b_types.h* 是本安全库的常用头文件。

调用以下函数来测试时钟频率：

- *FS_CLK_Init()*
- *FS_CLK_LPTMR() / FS_CLK_RTC() / FS_CLK_GPT() / FS_CLK_WKT_LPC() / FS_CLK_CTIMER_LPC()*
- *FS_CLK_Check()*

配置参考定时器，选择相应的周期性事件，并计算限值。声明 32 位全局变量，用于存储计时计数器寄存器的内容。所选定时器的时钟源必须与周期性事件的时钟源不同。*FS_CLK_Init()*函数被调用一次，通常在 while()循环之前。然后，在周期性事件中调用 *FS_CLK_LPTMR()* 函数（要为你的器件选择专用函数，请参阅内核自检库–源代码版本）。用于评估的 *FS_CLK_Check()*函数可以在任何给定时间调用。如果测试位于初始化阶段，check 函数会返回“进行中”值。如果从参考计数器捕获的值在预设范围内，则 check 函数返回通过值。如果不在预设范围内，则返回所定义的失败值。

测试实现的示例如下：

```
#include “iec60730b.h”
FS_RESULT st;
unsigned long clockTestContext;
#defineISR_FREQUENCY (100)
#define CLOCK_TEST_TOLERANCE (10)
#define REF_TIMER_CLOCK_FREQUENCY (32e03l)
RTC_SC = RTC_SC_RTCLKS(2)|RTC_SC_RTCPS(1);
SysTick->VAL = 0x0;
SysTick->LOAD = 100e6*0.01;
SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_ENABLE_Msk|
SysTick_CTRL_TICKINT_Msk;
SysTick->VAL = 0x0;

FS_CLK_Init(&clockTestContext);
 while(1) { st = FS_CLK_Check(clockTestContext, FS_CLK_FREQ_LIMIT_LO,
FS_CLK_FREQ_LIMIT_HI);
if (FS_FAIL_CLK == st) SafetyError();
}

void timer_isr(void)
{
 FS_CLK_RTC((uint32_t*)RTC_BASE_PTR, &clockTestContext);
}
```

3.2.1 FS_CLK_Init()

此函数初始化时钟同步测试的一个实例，将 TestContext 的值设置为“进行中”状态。

函数原型：

*void FS_CLK_Init(uint32_t *pTestContext);*

函数输入：

**pTestContext* - 指向保存捕获的定时器值的变量的指针。

函数输出：

空

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

3.2.2 FS_CLK_Check()

此函数处理时钟测试，使用预定义的限制评估存储在 testContext 变量中的捕获值。在第一次执行相应的 Isr 函数之前，该检查函数返回 FS_CLK_PROGRESS。

函数原型：

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

函数输入：

testContext - 定时器的捕获值。

limitLow - 下限。

limitHigh - 上限。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS* - testContext 符合限制。
- *FS_FAIL_CLK* - testContext 的值不符合限制。
- *FS_CLK_PROGRESS* - 尚未读取参考计数器的值。

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

3.2.3 FS_CLK_LPTMR()

此函数仅用于 LPTMR 模块。此函数从定时器中读取计数器值，并将其保存到 TestContext 变量中。之后，该函数再次启动 LPTMR。

函数原型：

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

函数输入：

**pSafetyTmr* - 定时器模块地址。

**pTestContext* - 指向保存捕获的定时器值的变量的指针。

函数输出：

空

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

3.2.4 FS_CLK_RTC()

此函数仅用于 RTC 模块。此函数从定时器中读取计数器值，并将其保存到 TestContext 变量中。之后，它再次启动 RTC。

函数原型：

*void FS_CLK_RTC(fs_rtc_t *pSafetyTmr, uint32_t *pTestContext);*

函数输入：

**pSafetyTmr* - 定时器模块地址。

**pTestContext* - 指向保存捕获的定时器值的变量的指针。

函数输出：

空

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

3.2.5 FS_CLK_GPT()

此函数仅用于 GPT 模块。此函数从定时器中读取计数器值，并将其保存到 TestContext 变量中。之后，它再次启动 GPT。

函数原型：

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

函数输入：

**pSafetyTmr* - 定时器模块地址。

**pTestContext* - 指向保存捕获的定时器值的变量的指针。

函数输出：

空

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

3.2.6 FS_CLK_CTIMER_LPC()

此函数仅用于 CTimer 模块。此函数从定时器中读取计数器值，并将其保存到 TestContext 变量中。之后，它再次启动 CTimer。

函数原型：

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

函数输入：

**pSafetyTmr* - 定时器模块地址。

**pTestContext* - 指向保存捕获的定时器值的变量的指针。

函数输出：

空

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

3.2.7 FS_CLK_WKT_LPC()

此函数仅用于 WKT 模块。此函数从定时器中读取计数器值，并将其保存到 TestContext 变量中。之后，它再次启动 WKT。

函数原型：

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

函数输入：

**pSafetyTmr* - 定时器模块地址。

**pTestContext* - 指向保存捕获的定时器值的变量的指针。

startValue - WKT 计数器递减的起始值。

函数输出：

空

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

第 4 章
数字输入/输出测试

数字输入/输出（DIO）测试程序执行处理器数字 IO 接口的置信性检查。

如果出现数字 IO 错误，通过特定的 FAIL 返回确保安全错误的识别。评估测试函数的返回值，如果返回值等于 FAIL 返回值，则应转入安全错误处理函数。安全错误处理函数可能特定于应用，不是库的一部分。此函数的主要目的是将应用置于安全状态。

DIO 测试函数旨在检查数字输入和输出功能以及被测引脚与电源电压、接地或可选相邻引脚之间的短路情况。DIO 测试的执行必须根据最终应用调整。注意硬件连接和设计。确定哪些函数可以应用到相应的引脚。在大多数情况下，必须在应用运行期间重新配置被测（有时也是辅助）引脚。测试数字输出时，在安排测试和读取结果之间预留足够的时间。

4.1 符合 IEC/UL 标准的数字输入/输出测试

所执行的过载测试符合 IEC 60730-1、IEC 60335、UL 60730 和 UL 1998 标准的安全要求，如表 16 所示。

表 16. 符合 IEC 和 UL 标准的数字输入/输出测试

测试	元器件	故障/错误	软件/硬件 类别	可接受的测量
输入/输出外设	7. 输入/输出 外设（7.1 – 数字 I/O）	异常操作	B/R.1	置信性检查

4.2 数字输入/输出测试的实现

数字 IO 测试的测试函数在 *iec60730b_dio.c* 和 *iec60730b-dio_ext.c* 文件中。带函数原型的头文件是 *iec60730b_dio.h* 和 i*ec60730b_dio_ext.h*。*Iec60730b.h* 和 *iec60730b_types.h* 是此安全库的常用头文件。

数字输入/输出测试可适当地用以下函数来执行：

- *FS_DIO_Input()*
- *FS_DIO_Output() / FS_DIO_Output_IMXRT() / FS_DIO_Output_IMX8M() / FS_DIO_Output_LPC()*
- *FS_DIO_InputExt() / FS_DIO_InputExt_IMXRT() / FS_DIO_InputExt_IMX8M() / FS_DIO_InputExt_LPC()*
- *FS_DIO_ShortToSupplySet() / FS_DIO_ShortToSupplySet_IMXRT() / FS_DIO_ShortToSupplySet_IMX8M() / FS_DIO_ShortToSupplySet_LPC()*
- *FS_DIO_ShortToAdjSet() / FS_DIO_ShortToAdjSet_IMXRT() / FS_DIO_ShortToAdjSet_IMX8M() / FS_DIO_ShortToAdjSet_LPC()*

这个指向 fs_dio_test_t 结构类型的指针是每个函数的参数。该结构在 *iec60730b_dio.h* 文件中定义。

```
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;
```

在调用测试函数之前，必须初始化这些变量。以下是初始化示例：

```
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()

此函数执行数字输入测试。该测试测试一个数字引脚。引脚按照图 3 中的框图进行测试：

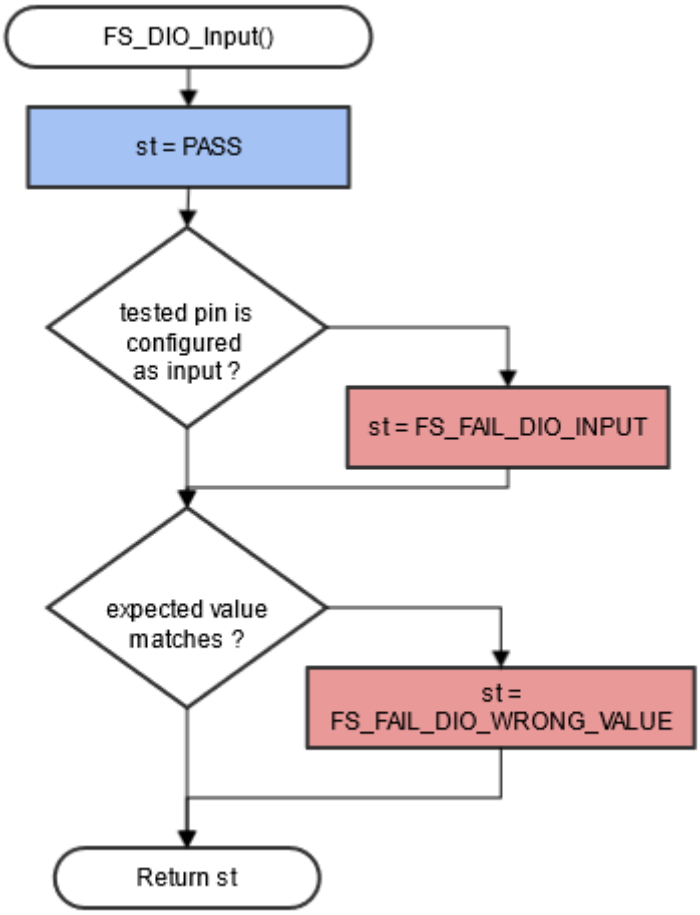

图 3. 数字输入测试的框图

函数原型：

*FS_RESULT FS_DIO_Input(fs_dio_test_t *pTestedPin, bool_t expectedValue);*

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

expectedValue - 预期的输入值。正确调整此参数。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 引脚未设置为输入。
- *FS_FAIL_DIO_WRONG_VALUE* - 引脚没有预期值。

函数总是返回第一个检测到的错误。

函数调用的示例：

fs_dio_input_test_result = FS_DIO_Input(&dio_safety_test_items[0], DIO_EXPECTED_VALUE);

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

调用限制：

被测引脚必须配置为有输入方向的 GPIO。

4.2.2 FS_DIO_Output()

数字输出测试测试被测引脚的数字输出功能。测试的原理是在被测引脚上设置和读取所有两个逻辑值。输入适当的延迟参数。必须确保时间间隔足够长，让器件达到引脚上期望的逻辑值。太低的延迟参数会导致函数返回出现错误值。

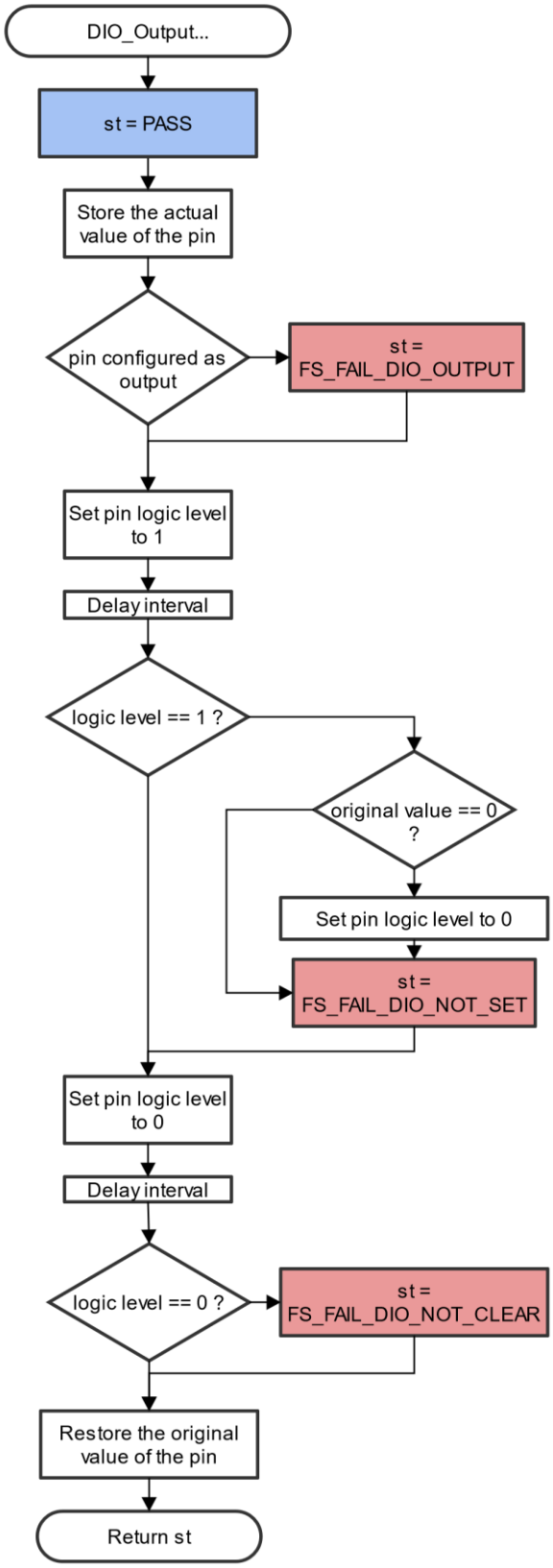

图 4. 数字输出测试的框图

函数原型：

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

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

delay - 识别被测引脚上值的变化所需的延迟。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_OUTPUT* - 引脚未设置为输出。
- *FS_FAIL_DIO_NOT_SET* - 引脚不能设置为逻辑 1。
- *FS_FAIL_DIO_NOT_CLEAR* - 引脚无法清除为逻辑 0。

函数总是返回第一个检测到的错误。

函数调用示例：

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

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

调用限制：

被测引脚必须配置为数字输出。为适当的功能定义适当的延迟。

4.2.3 FS_DIO_InputExt()

这是之前提到的数字输入测试的修改版本。它不能用于 MKE0x 器件。这个版本是一个用于“短路到”（“short-to”）测试的 get 函数。该函数应用于已经配置为 GPIO 输入的引脚，而且知道测试时期望的逻辑电平。逻辑电平可以由应用中的实际配置产生，也可以是为测试所做的初始化（如果可能）。*FS_DIO_InputExt()* 函数的框图如图 5 所示。两个输入参数与相邻的引脚相关。对于简单的输入测试函数，这些参数并不重要。输入与被测引脚相同的输入（推荐）。请参阅示例代码。

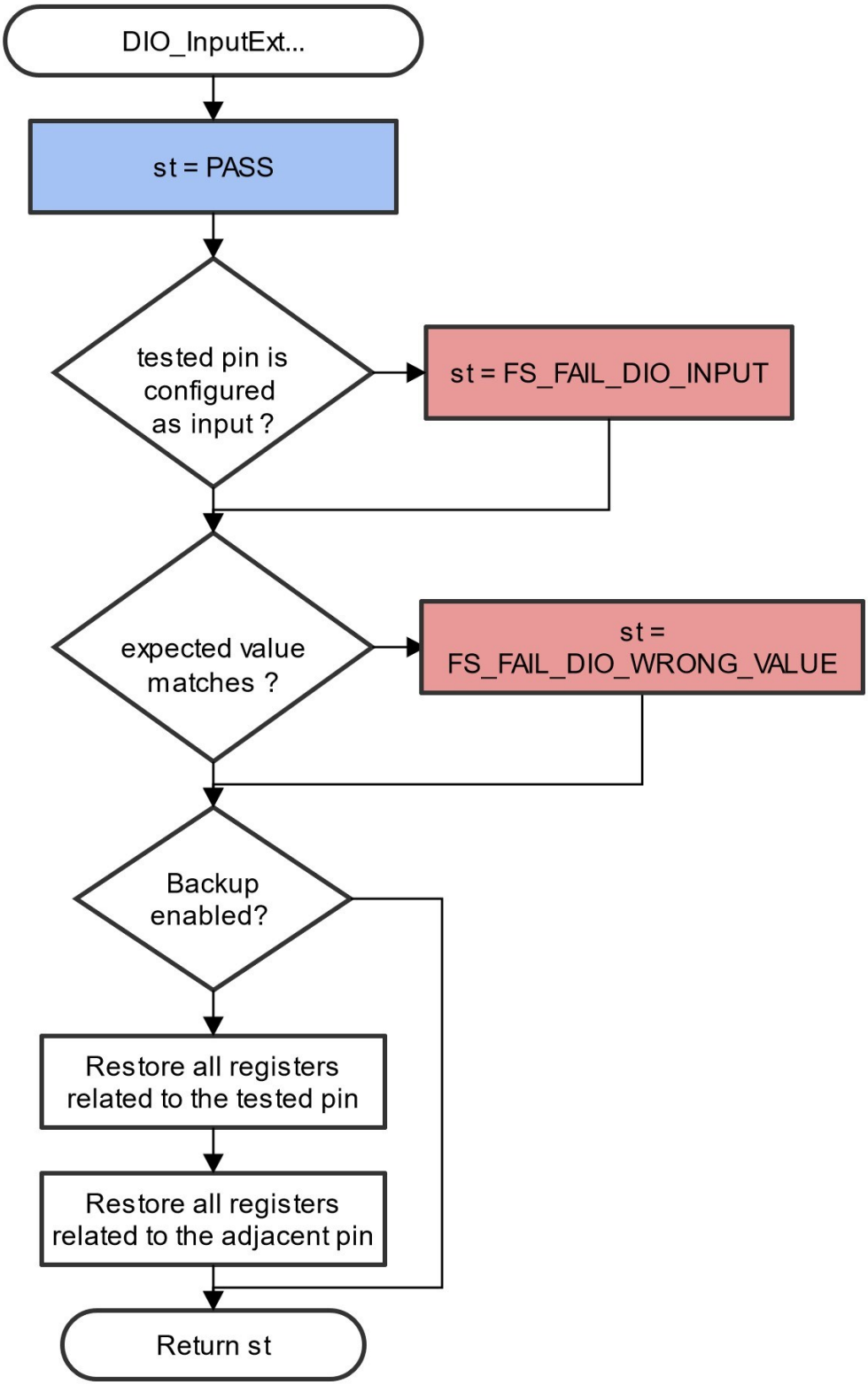

图 5. 扩展的数字输入测试

函数原型：

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

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

**pAdjPin* - 指向相邻引脚结构的指针。

testedPinValue - 被测引脚的预期值（逻辑 0 或逻辑 1）。请正确调整此参数。

backupEnable - 标志。如果该值不为零，则表示启用/激活备份功能。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 引脚未设置为输入。
- *FS_FAIL_DIO_WRONG_VALUE* - 引脚没有预期值。

函数总是返回第一个检测到的错误。

函数调用示例：

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

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

调用限制：

该函数不能用于 MKE0x 器件。在调用函数之前，必须将被测引脚配置为 GPIO 输入。即使测试中没有相邻的引脚，也要指定相邻引脚的参数。建议输入与被测引脚相同的输入。

4.2.4 FS_DIO_ShortToAdjSet()

此函数可确保相邻引脚短路测试所需的条件，目的是正确配置被测引脚和相邻引脚。相邻引脚是可选引脚，理论上可以对被测引脚造成短路。函数的框图如图 6 所示。与电源短路测试类似，该测试需要使用两个函数。第二个（get）函数评估测试结果。*FS_DIO_InputExt()* 函数在相应章节中进行了描述。指定此输入测试函数的被测引脚和相邻引脚。

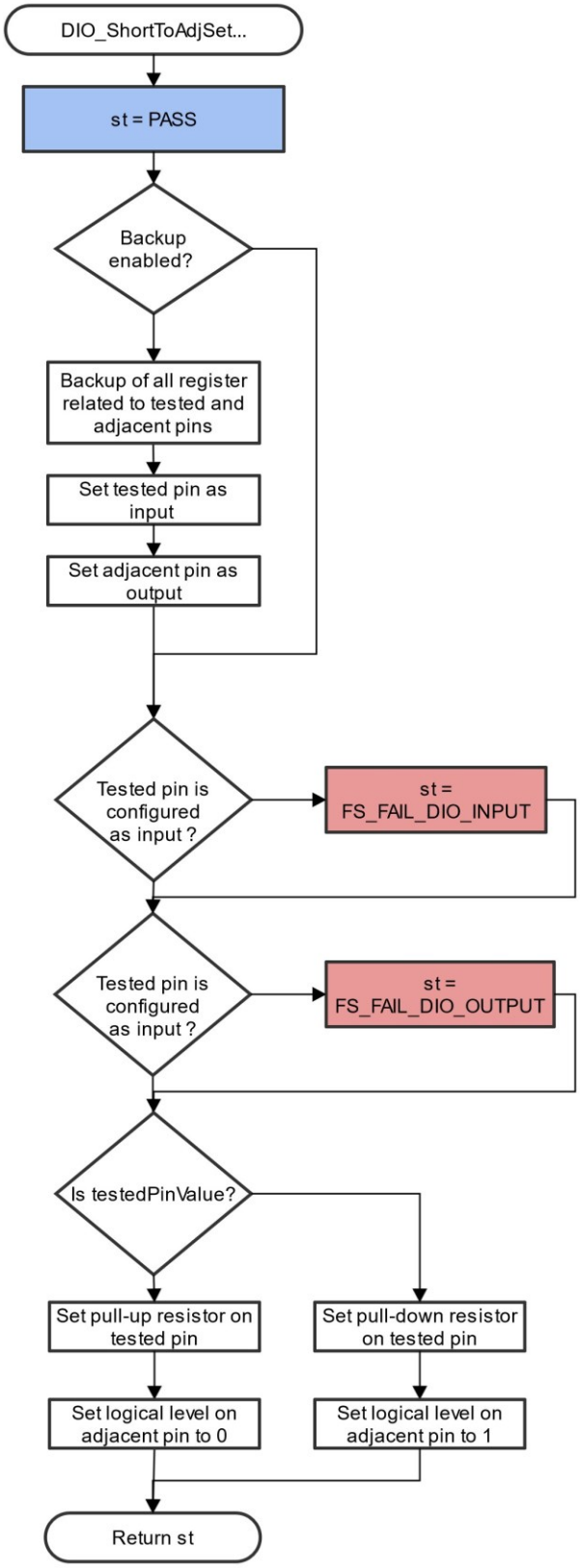

图 6. FS_DIO_ShortToAdjSet()函数的框图

函数原型：

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

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

**pAdjPin* - 指向相邻引脚结构的指针。

testedPinValue - 要在被测引脚上设置的值。

backupEnable - 标志。如果该值不为零，则表示启用/激活备份功能。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 引脚未设置为输入。
- *FS_FAIL_DIO_OUTPUT* - 相邻引脚未设置为输出。

函数总是返回第一个检测到的错误。

函数调用示例：

以下是对相邻引脚短路测试的代码示例：

```
#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);
```

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

该函数不能用于 MKE0x 器件。在调用函数之前，被测引脚必须配置为 GPIO 输入，相邻引脚必须配置为 GPIO 输出。如果启用了备份功能，该函数将为两个引脚都设置方向。如果没有，则配置方向（被测引脚作为输入，相邻引脚作为输出）。函数结束后，应用既不能操作被测引脚，也不能操作相邻引脚，直到为这些引脚调用了 *FS_DIO_InputExt()* 函数。

4.2.5 FS_DIO_ShortToSupplySet()

此函数创建电源短路测试的第一部分，可用于测试被测引脚与硬件电源电压（Vcc、Vdd）之间或被测引脚和硬件地（GND）之间的短路。其框图如图 7 所示。测试的第二部分（结果评估）通过相应章节中所述的 *FS_DIO_InputExt()* 函数完成。*FS_DIO_InputExt()* 函数的主要目的是在被测引脚上设置上拉（或下拉）电阻连接，还可以确保引脚是否正确配置并备份其设置（如果需要）。

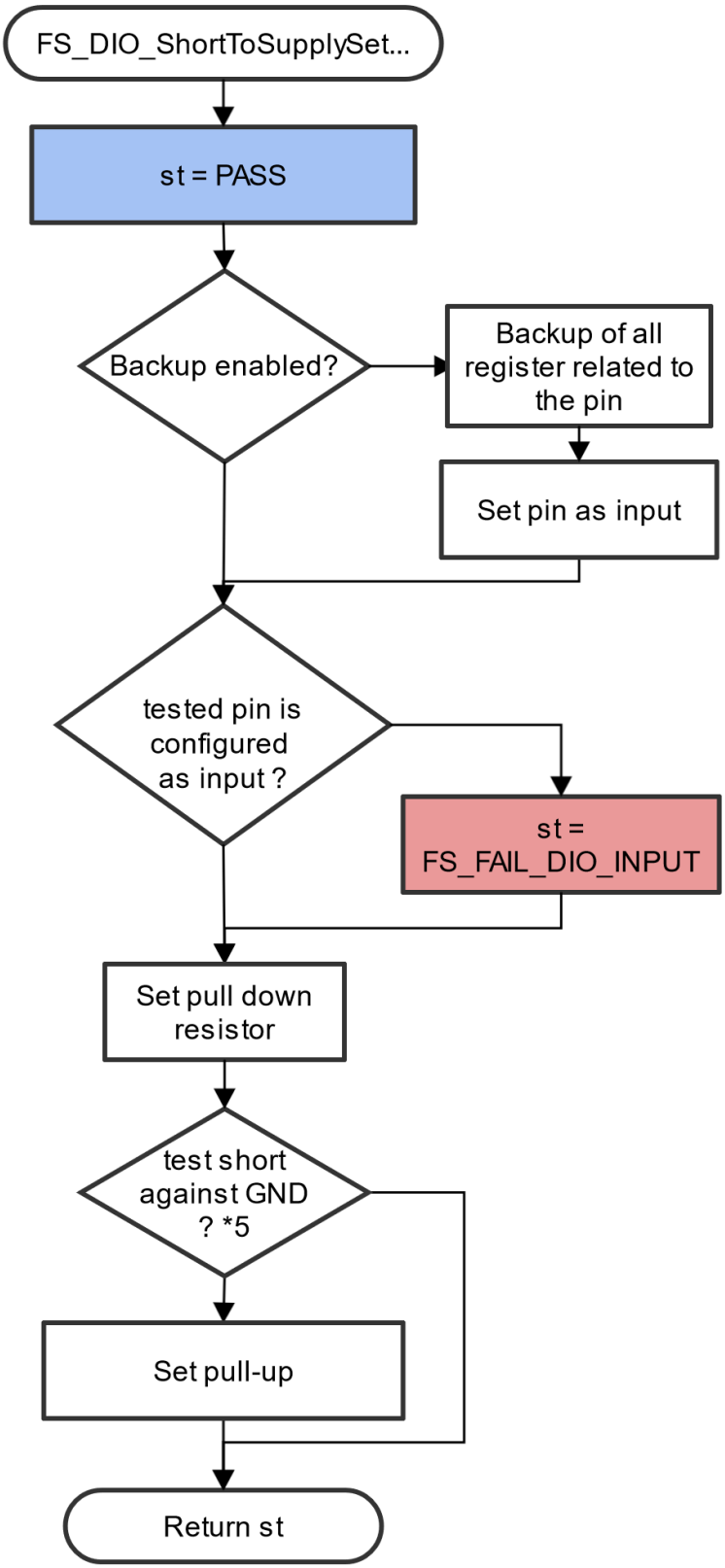

图 7. FS_DIO_ShortToSupplySet 函数的框图

函数原型：

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

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

shortToVoltage -说明是否针对 GND 或 Vdd 进行引脚短路测试。对于 GND，输入“1”。对于 VDD，输入 0 或非 0 值。

backupEnable - 标志。如果该值不为零，则备份功能处于启用/激活状态。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 引脚未设置为输入。

函数总是返回第一个检测到的错误。

函数调用示例：

以下是既对 GND 短路也对 VDD 短路情况的测试代码示例。请注意，实现的区别仅在于一个参数。如果测试对 GND 的短路，则参数必须具有非零值，反之同理。

```
#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);
```

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

该函数不能用于 MKE0x 器件。在调用函数之前，必须将被测引脚配置为 GPIO 输入。如果启用了备份功能，该函数将为被测引脚设置输入方向。如果没有启用，请配置输入方向。函数结束后，应用不能操作被测引脚，直到为被测引脚调用 *FS_DIO_InputExt()* 函数。

4.2.6 FS_DIO_InputExt_IMX8M()

这是之前提到的数字输入测试的修改版本。将此版本用作“短路到”（“short to”）测试的 get 函数。将该函数应用于已配置为 GPIO 输入的引脚，且知道测试时期望的逻辑电平。该逻辑电平可以通过应用中的实际配置产生，也可以是为此测试所做的初始化（如果可能）。*FS_DIO_InputExt_IMX8M()* 函数的框图如图 8 所示。两个输入参数与相邻的引脚相关。对于简单的输入测试功能，这些参数并不重要。输入与被测引脚相同的输入（推荐）。请参见示例代码。

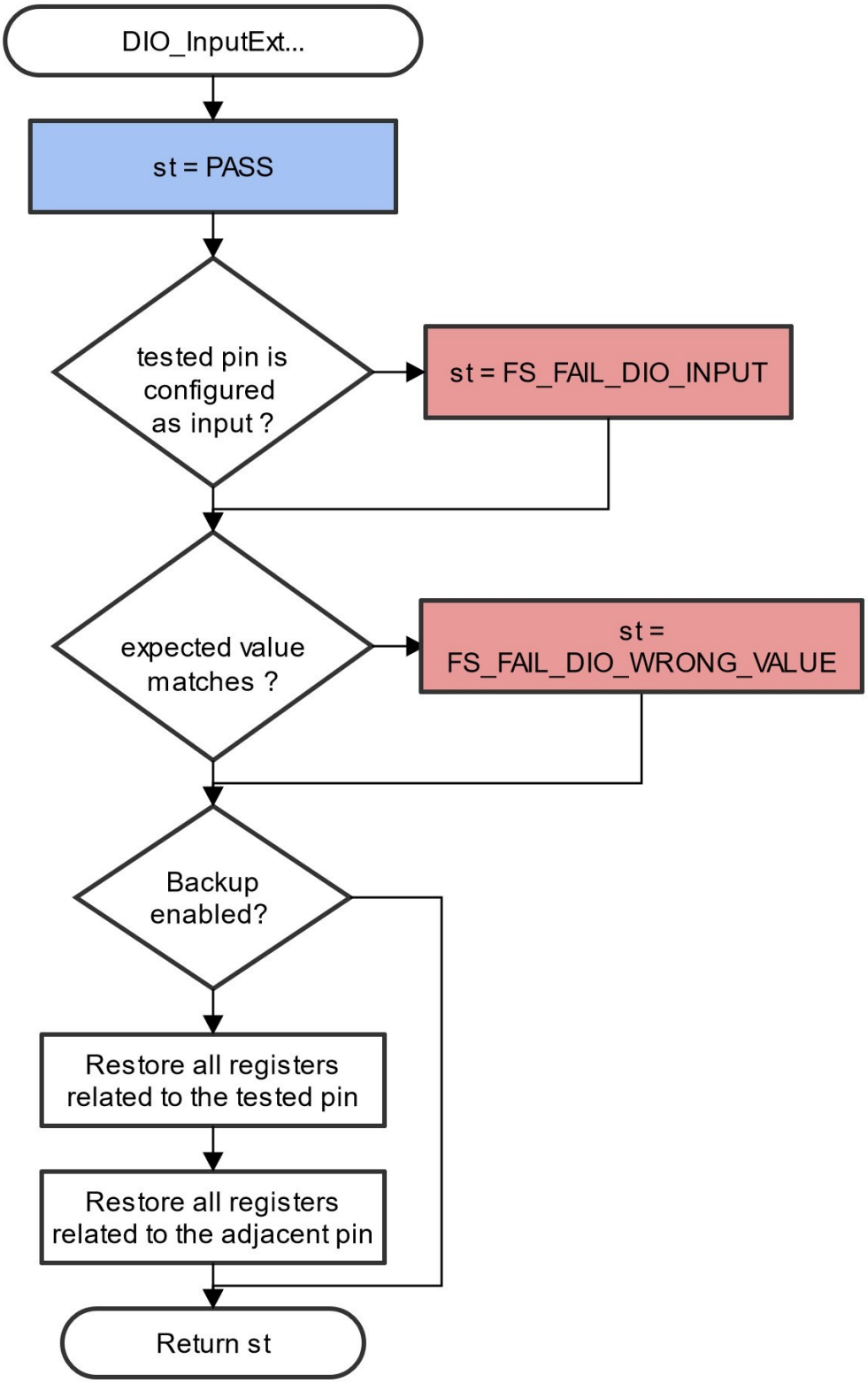

图 8. IMX8M 扩展数字输入测试

函数原型：

*FS_RESULT FS_DIO_InputExt_IMX8M(fs_dio_test_imx_t *pTestedPin, fs_dio_test_imx_t *pAdjPin, bool_t testedPinValue, bool_t backupEnable);*

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

**pAdjPin* - 指向相邻引脚结构的指针。

testedPinValue - 被测引脚的预期值（逻辑 0 或逻辑 1）。请正确调整此参数。

backupEnable - 标志。如果该值不为零，则表示启用/激活备份功能。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 引脚未设置为输入。
- *FS_FAIL_DIO_WRONG_VALUE* - 引脚没有预期值。

函数总是返回第一个检测到的错误。

函数调用示例：

fs_dio_input_test_result = FS_DIO_InputExt_IMX8M(&dio_safety_test_item_0, &dio_safety_test_item_0, DIO_EXPECTED_VALUE, BACKUP_ENABLE);

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

该函数只能用于 i.MX8M 器件。在调用函数之前，必须将被测引脚配置为 GPIO 输入。即使测试中没有涉及相邻的引脚，也要指定“相邻引脚”参数。建议输入与“被测引脚”相同的输入。

4.2.7 FS_DIO_Output_IMX8M()

此测试测试引脚的数字输出功能。此测试的原理是在被测试的引脚上设置和读取所有两个逻辑值。请输入合适的延迟参数。它必须确保时间间隔足够长，让器件在引脚上达到期望的逻辑值。太低的延迟参数会导致函数的“失败”（“fail”）返回值。

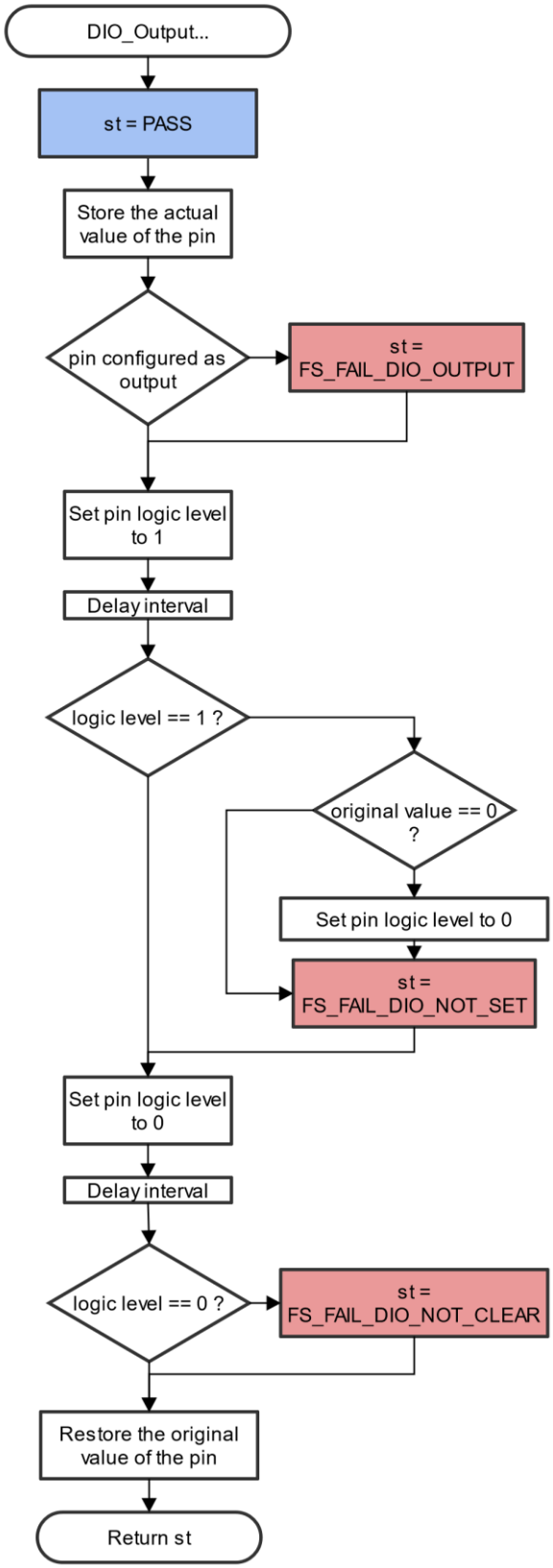
DIO_Output...
st = PASS
Store the actual value of the pin
pin configured as output
st = FS_FAIL_DIO_OUTPUT
Set pin logic level to 1
Delay interval
logic level == 1 ?
original value == 0 ?
Set pin logic level to 0
st = FS_FAIL_DIO_NOT_SET
Set pin logic level to 0
Delay interval
logic level == 0 ?
st = FS_FAIL_DIO_NOT_CLEAR
Restore the original value of the pin
Return st

图 9. 数字输出测试框图

函数原型：

*FS_RESULT FS_DIO_Output_IMX8M(fs_dio_test_imx_t *pTestedPin, uint32_t delay);*

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

delay - 识别被测引脚上值变化所需的延迟。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_OUTPUT* - 引脚未设置为输出。
- *FS_FAIL_DIO_NOT_SET* - 引脚不能设置为逻辑 1。
- *FS_FAIL_DIO_NOT_CLEAR* - 引脚无法清除为逻辑 0。

函数总是返回第一个检测到的错误。

函数调用示例：

fs_dio_output_test_result = FS_DIO_Output_IMX8M(&dio_safety_test_items[1], DIO_WAIT_CYCLE);

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

被测引脚必须配置为数字输出。为适当的功能定义适当的延迟。

4.2.8 FS_DIO_ShortToAdjSet_IMX8M()

此函数可确保相邻引脚短路测试所需的条件。此函数的目的是正确配置被测引脚和相邻引脚。相邻引脚是可选引脚，理论上可能对被测引脚造成短路。函数框图如图 10 所示。与电源短路测试类似，该测试需要使用两个函数。第二个（get）函数评估测试结果。*FS_DIO_InputExt()* 函数在相应章节中进行了描述。指定输入测试函数的被测引脚和相邻引脚。

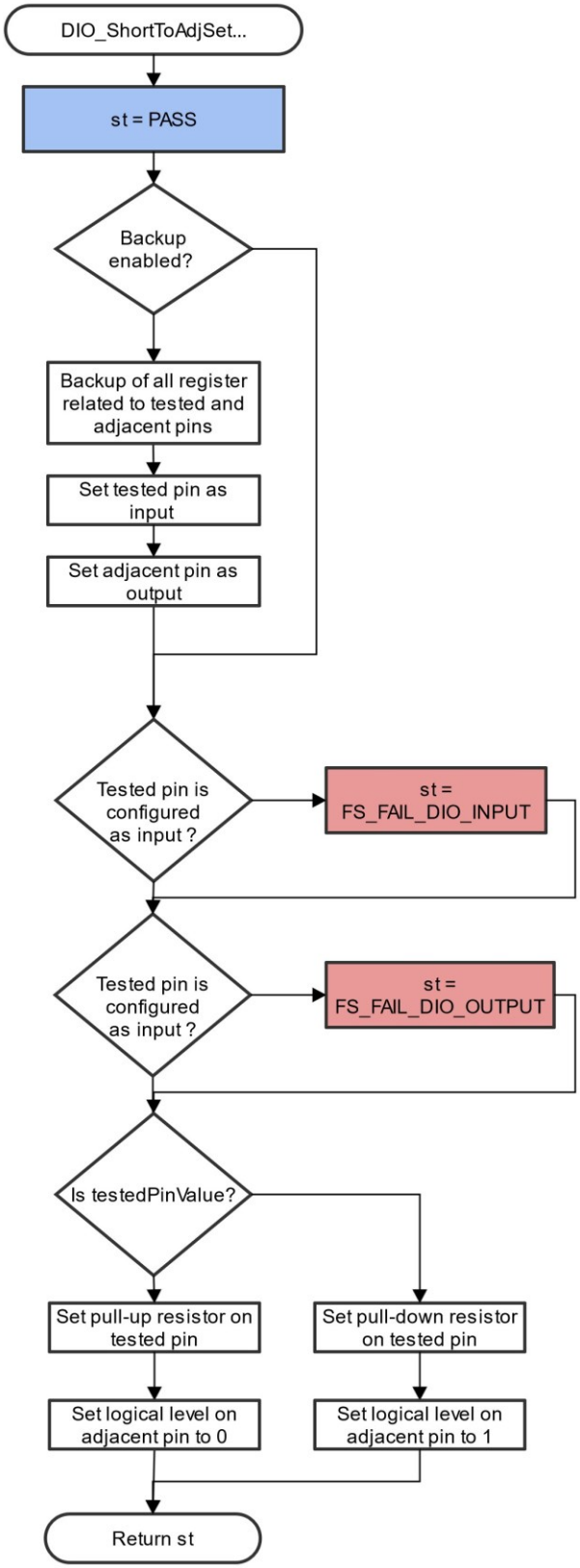

图 10. FS_DIO_ShortToAdjSet_IMX8M() 函数的框图

函数原型：

*FS_RESULT FS_DIO_ShortToAdjSet_IMX8M(fs_dio_test_imx_t *pTestedPin, fs_dio_test_imx_t *pAdjPin, bool_t testedPinValue, bool_t backupEnable);*

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

**pAdjPin* - 指向相邻引脚结构的指针。

testedPinValue - 在被测引脚上设置的值。

backupEnable - 标志。如果该值不为零，则表示启用/激活备份功能。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 引脚未设置为输入。
- *FS_FAIL_DIO_OUTPUT* - 相邻引脚未设置为输出。

函数总是返回第一个检测到的错误。

函数调用示例：

以下是相邻引脚短路测试的代码示例：

```
#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);
```

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

调用限制：

在调用函数之前，被测引脚必须配置为 GPIO 输入，相邻引脚必须配置为 GPIO 输出。如果启用了备份功能，该函数将为两个引脚都设置方向。如果没有，则配置方向（被测引脚作为输入，相邻引脚作为输出）。函数结束后，应用既无法操作被测引脚，也无法操作相邻引脚，直到为这些引脚调用 *FS_DIO_InputExt_IMX8M()* 函数。

4.2.8.1 FS_DIO_ShortToAdjSet_LPC()

此函数可确保相邻引脚短路测试所需的条件。此函数的目的是正确配置被测引脚和相邻引脚。相邻引脚是可选引脚，理论上可能对被测引脚造成短路。函数框图如图 11 所示。与电源短路测试类似，该测试需要使用两个函数。第二个（get）函数评估测试结果。*FS_DIO_InputExt()* 函数在相应章节中进行了描述。指定输入测试函数的被测引脚和相邻引脚。

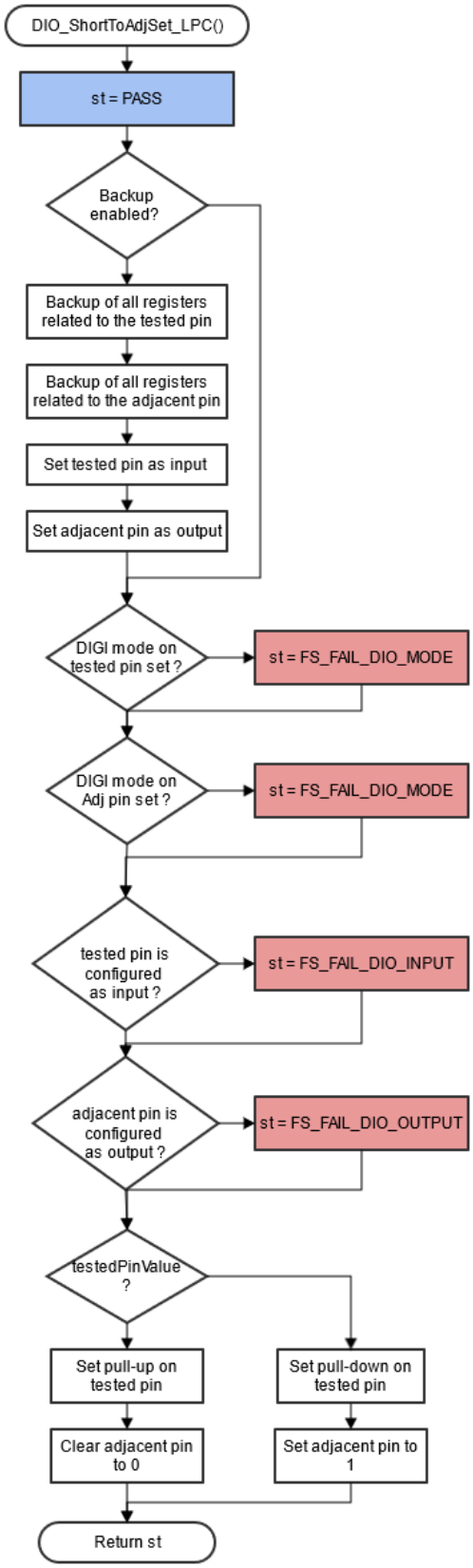

图 11. FS_DIO_ShortToAdjSet_LPC() 函数的框图

函数原型：

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

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

**pAdjPin* - 指向相邻引脚结构的指针。

testedPinValue - 在被测引脚上设置的值。

backupEnable - 标志。如果该值不为零，则表示启用/激活备份功能。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 被测引脚未设置为输入。
- *FS_FAIL_DIO_OUTPUT* - 相邻引脚未设置为输出。
- *FS_FAIL_DIO_MODE* - 被测或相邻引脚没有“数字模式”设置——仅用于特定的 LPC 器件。

函数总是返回第一个检测到的错误。

函数调用示例：

以下是相邻引脚短路测试的代码示例：

```
#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);
```

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

在调用函数之前，被测引脚必须配置为 GPIO 输入，相邻引脚必须配置为 GPIO 输出。如果启用了备份功能，该函数将为两个引脚都设置方向。如果没有，则配置方向（被测引脚作为输入，相邻引脚作为输出）。函数结束后，应用既不能操作被测引脚，也不能操作相邻引脚，直到为这些引脚调用 *FS_DIO_InputExt_LPC()* 函数。

4.2.9 FS_DIO_ShortToSupplySet_IMX8M()

此函数创建电源短路测试的第一部分，可用于测试被测引脚与硬件电源电压（Vcc、Vdd）之间或被测引脚和硬件地（GND）之间的短路。其框图如图 12 所示。测试的第二部分（结果评估）通过相应章节中所述的 *FS_DIO_InputExt_IMX8M()* 函数完成。*FS_DIO_InputExt_IMX8M()* 的主要目的是设置被测引脚上的上拉（或下拉）电阻连接，还可以确保引脚是否正确配置并备份其设置（如果需要）。

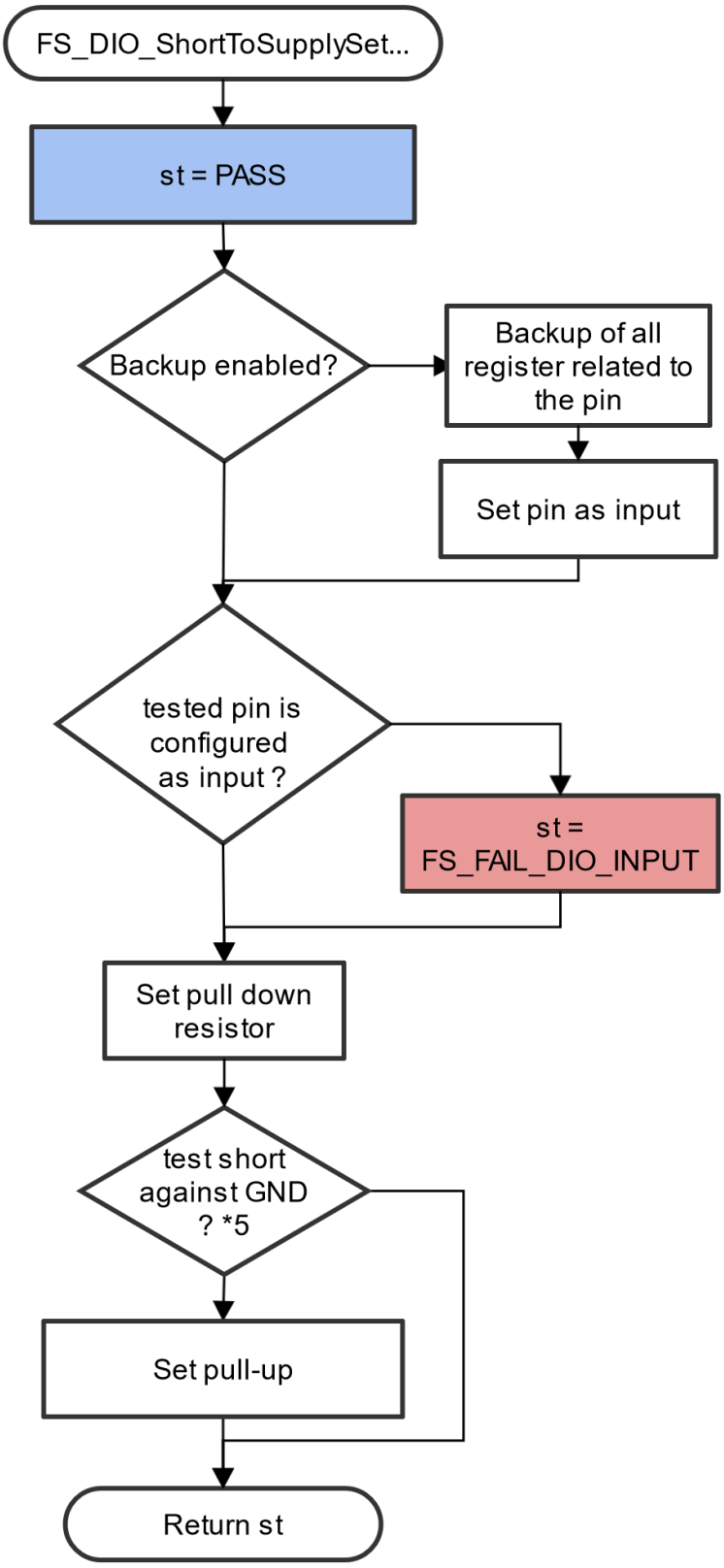

图 12. FS_DIO_ShortToSupplySet_IMX8M() 函数的框图

函数原型：

*FS_RESULT FS_DIO_ShortToSupplySet_IMX8M(fs_dio_test_imx_t *pTestedPin, bool_t shortToVoltage, bool_t backupEnable);*

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

shortToVoltage - 指定是否针对 GND 或 VDD 进行引脚短路测试。对于 GND，输入“1”。对于 VDD，输入 0 或非 0 值。

backupEnable - 标志。如果该值不为零，则表示启用/激活备份功能。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 引脚未设置为输入。

函数总是返回第一个检测到的错误。

函数调用示例：

以下既是 GND 短路也是 VDD 短路情况的测试代码示例。请注意，实现的区别仅在于一个参数。如果测试了 GND 短路，则参数必须具有非零值，反之同理。

```
#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);
```

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

在调用函数之前，必须将被测引脚配置为 GPIO 输入。如果启用了备份功能，该函数将为被测引脚设置输入方向。如果没有启用，请配置输入方向。函数结束后，应用不能操作被测引脚，直到为被测引脚调用 *FS_DIO_InputExt_IMX8M()* 函数。

4.2.10 FS_DIO_InputExt_IMXRT()

这是之前提到的数字输入测试的修改版本。将此版本用作“短路到”（“short to”）测试的 get 函数。将该函数应用于已配置为 GPIO 输入的引脚，且知道测试时期望的逻辑电平。该逻辑电平通过应用中的实际配置产生，也可以是为测试所做的初始化（如果可能）。*FS_DIO_InputExt_IMXRT()* 函数的框图如图 13 所示。两个输入参数与相邻的引脚相关。对于简单的输入测试功能，

这些参数并不重要。输入与被测引脚相同的输入（推荐）。请参见示例代码。

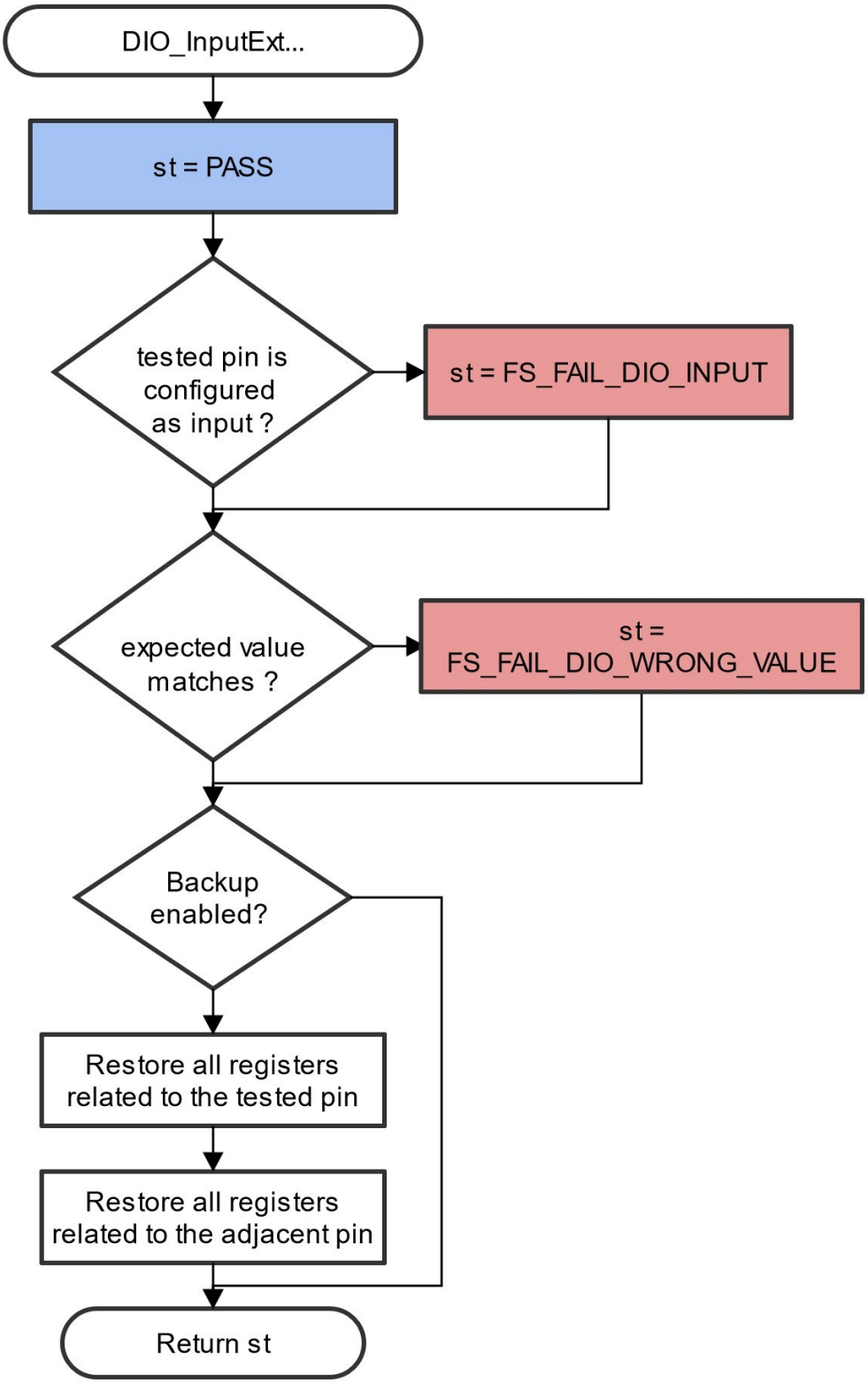

图 13. IMXRT 扩展数字输入测试

函数原型：

*FS_RESULT FS_DIO_InputExt_IMXRT(fs_dio_test_imx_t *pTestedPin, fs_dio_test_imx_t *pAdjPin, bool_t testedPinValue, bool_t backupEnable);*

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

**pAdjPin* - 指向相邻引脚结构的指针。

testedPinValue - 被测引脚的预期值（逻辑 0 或逻辑 1）。请正确调整此参数。

backupEnable - 标志。如果该值不为零，则表示启用/激活备份功能。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 引脚未设置为输入。
- *FS_FAIL_DIO_WRONG_VALUE* - 引脚没有预期值。

函数总是返回第一个检测到的错误。

函数调用示例：

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

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

该函数只能用于 i.MX RT 器件。在调用函数之前，必须将被测引脚配置为 GPIO 输入。即使测试中没有涉及相邻的引脚，也要指定“相邻引脚”参数。建议输入与“被测引脚”相同的输入。

4.2.11 FS_DIO_Output_IMXRT()

此测试测试引脚的数字输出功能。这个测试的原理是在被测引脚上设置和读取所有两个逻辑值。请输入合适的延迟参数。它必须确保时间间隔足够长，让器件达到引脚上期望的逻辑值。超低的延迟参数会导致函数的“失败”（“fail”）返回值。

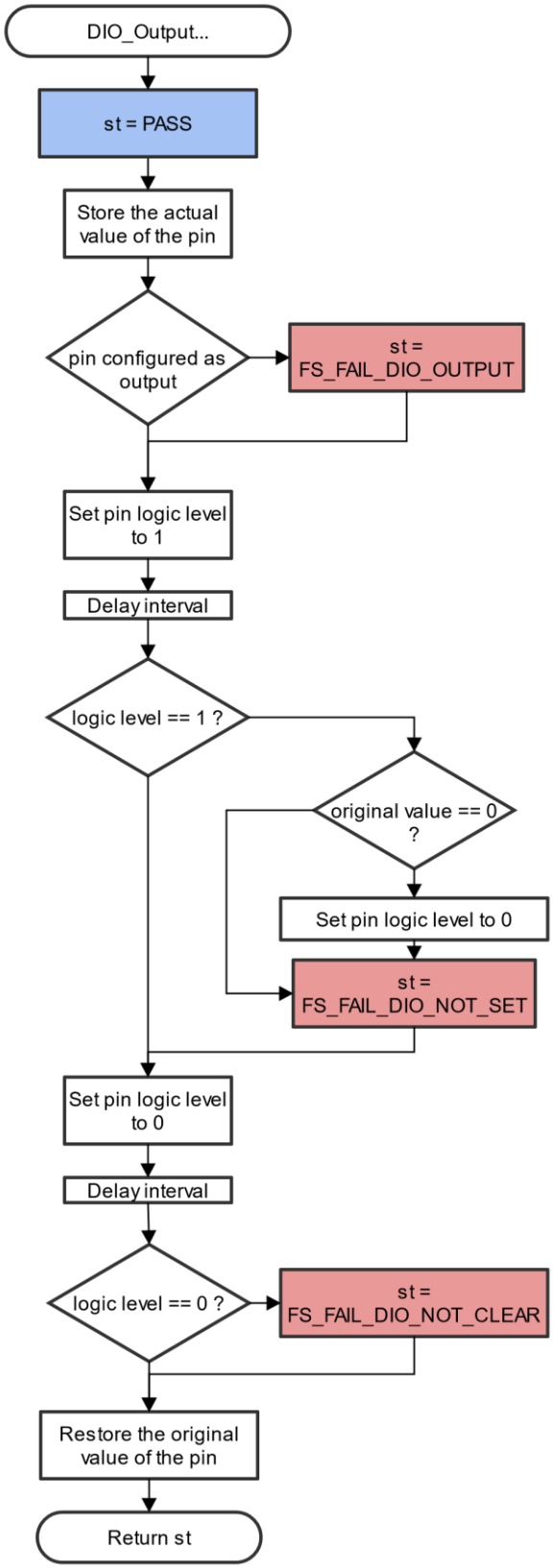

图 14. 数字输出测试的框图

函数原型：

*FS_RESULT FS_DIO_Output_IMXRT(fs_dio_test_imx_t *pTestedPin, uint32_t delay);*

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

delay - 识别被测引脚上值变化所需的延迟。

函的输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_OUTPUT* - 引脚未设置为输出。
- *FS_FAIL_DIO_NOT_SET* - 引脚不能设置为逻辑 1。
- *FS_FAIL_DIO_NOT_CLEAR* - 引脚无法清除为逻辑 0。

函数总是返回第一个检测到的错误。

函数调用示例：

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

函数的性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

被测引脚必须配置为数字输出。为适当的功能定义适当的延迟。

4.2.12 FS_DIO_ShortToAdjSet_IMXRT()

此函数可确保相邻引脚短路测试所需的条件。此函数的目的是正确配置被测引脚和相邻引脚。相邻引脚是可选引脚，理论上可能对被测引脚造成短路。函数框图如图 15 所示。与电源短路测试类似，该测试需要使用两个函数。第二个（get）函数评估测试结果。*FS_DIO_InputExt_IMXRT()* 函数在相应章节中进行描述。指定输入测试函数的被测引脚和相邻引脚。

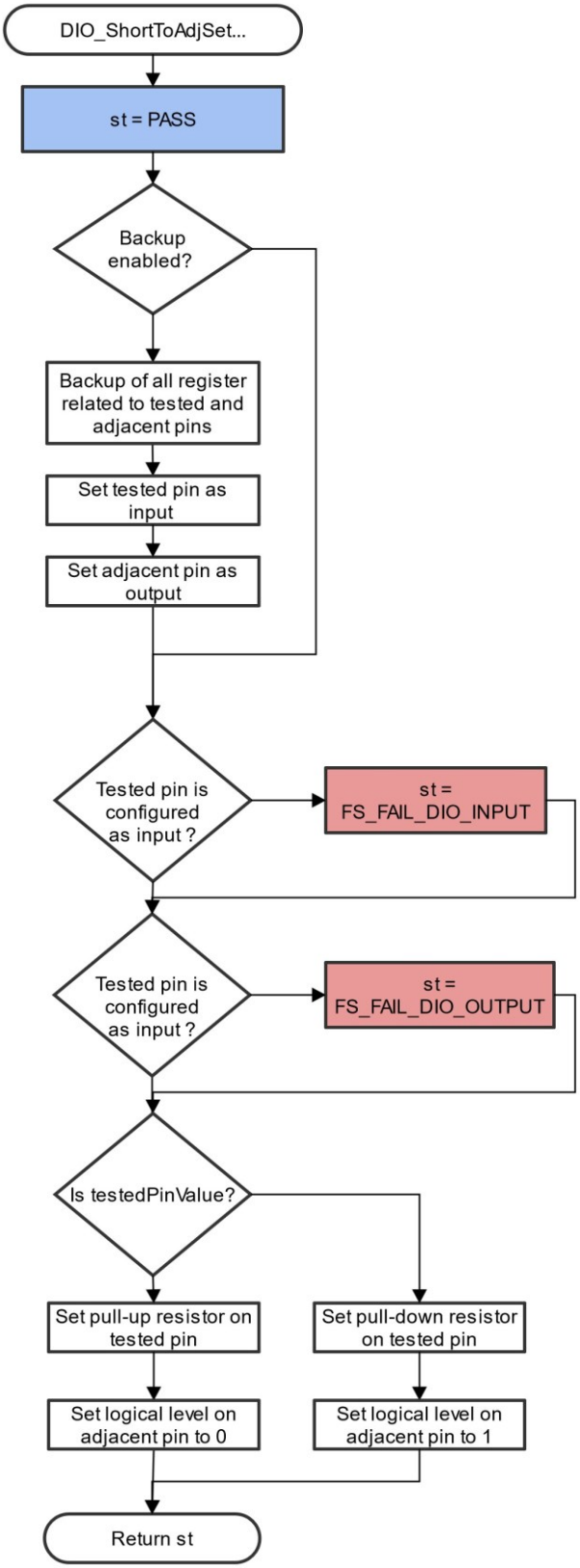

图 15. FS_DIO_ShortToAdjSet_IMXRT() 函数的框图

函数原型：

*FS_RESULT FS_DIO_ShortToAdjSet_IMXRT(fs_dio_test_imx_t *pTestedPin, fs_dio_test_imx_t *pAdjPin, bool_t testedPinValue, bool_t backupEnable);*

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

**pAdjPin* - 指向相邻引脚结构的指针。

testedPinValue - 在被测引脚上设置的值。

backupEnable - 标志。如果该值不为零，则表示启用/激活备份功能。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 被测引脚未设置为输入。
- *FS_FAIL_DIO_OUTPUT* - 相邻引脚未设置为输出。

函数总是返回第一个检测到的错误。

函数调用示例：

以下是相邻引脚短路测试的代码示例：

```
#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);
```

函数的性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

在调用函数之前，被测引脚必须配置为 GPIO 输入，相邻引脚必须配置为 GPIO 输出。如果启用了备份功能，该函数将为两个引脚都设置方向。如果没有，则配置方向（被测引脚作为输入，相邻引脚作为输出）。函数结束后，应用既无法操作被测引脚，也无法操作相邻引脚，直到为这些引脚调用 *FS_DIO_InputExt_IMXRT()* 函数。

4.2.13 FS_DIO_ShortToSupplySet_IMXRT()

此函数创建电源短路测试的第一部分，可用于测试被测引脚与硬件电源电压（Vcc、Vdd）之间或被测引脚和硬件地（GND）之间的短路。其框图如图 16 所示。测试的第二部分（结果评估）通过在相应章节中所述的 *FS_DIO_InputExt_IMXRT()* 函数完成。*FS_DIO_InputExt_IMXRT()* 函数的主要目的是在被测引脚上设置上拉（或下拉）电阻连接，还可以确保引脚是否正确配置并备份其设置（如果需要）。

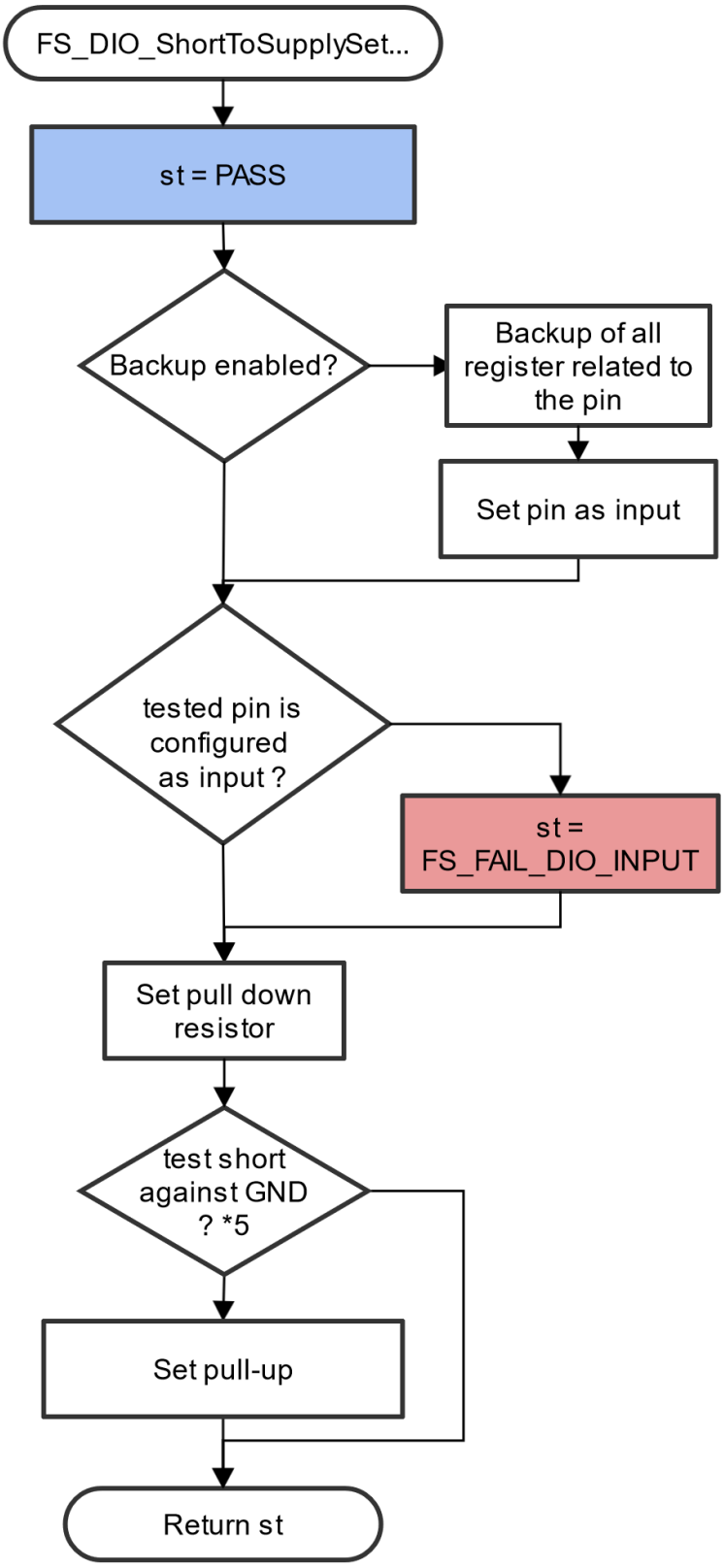

图 16. FS_DIO_ShortToSupplySet_IMXRT() 函数的框图

函数原型：

*FS_RESULT FS_DIO_ShortToSupplySet_IMXRT(fs_dio_test_imx_t *pTestedPin, bool_t shortToVoltage, bool_t backupEnable);*

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

shortToVoltage - 指定是否针对 GND 或 VDD 进行引脚短路测试。对于 GND，输入“1”。对于 VDD，输入 0 或非 0 值。

backupEnable - 标志。如果该值不为零，则表示启用/激活备份功能。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 引脚未设置为输入。

函数总是返回第一个检测到的错误。

函数调用示例：

以下既是 GND 短路也是 VDD 短路情况的测试代码示例。请注意，实现的区别仅在于一个参数。如果测试了 GND 短路，则参数必须具有非零值（反之同理）。

```
#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);
```

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

在调用函数之前，必须将被测引脚配置为 GPIO 输入。如果启用了备份功能，该函数将为被测引脚设置输入方向。如果没有，请配置输入方向。函数结束后，应用无法操作被测引脚，直到为被测引脚调用 *FS_DIO_InputExt_IMXRT()* 函数。

4.2.14 FS_DIO_InputExt_LPC()

这是之前提到的数字输入测试的修改版本。将此版本用作“短路到”（“short to”）测试的 get 函数。将该函数应用于已配置为 GPIO 输入的引脚，且知道测试时期望的逻辑电平。该逻辑电平可通过应用中的实际配置产生，也可以是为测试所做的初始化（如果可能）。*FS_DIO_InputExt_LPC()* 函数的框图如图 17 所示。两个输入参数与相邻的引脚相关。对于简单的输入测试功能，这些参数并不重要。输入与被测引脚相同的输入（推荐）。请参见示例代码。

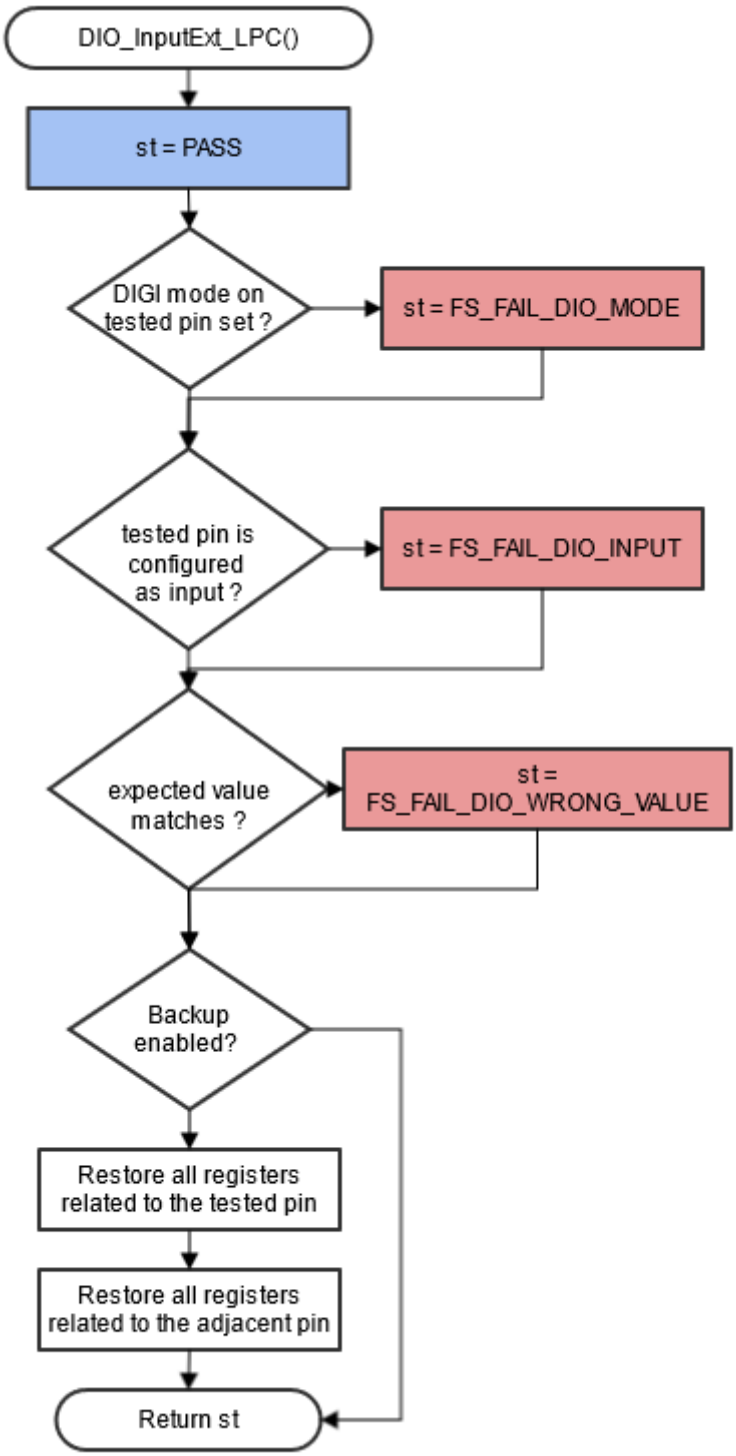

图 17. LPC 器件扩展数字输入测试

函数原型：

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

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

**pAdjPin* - 指向相邻引脚结构的指针。

testedPinValue - 被测引脚的预期值（逻辑 0 或逻辑 1）。请正确调整此参数。

backupEnable - 标志。如果该值不为零，则表示启用/激活备份功能。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 引脚未设置为输入。
- *FS_FAIL_DIO_WRONG_VALUE* - 引脚没有预期值。
- *FS_FAIL_DIO_MODE* - 引脚没有“数字模式”设置 - 仅用于特定的 LPC 器件。

函数总是返回第一个检测到的错误。

函数调用示例：

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

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

调用限制：

在函数调用之前，将被测引脚配置为 GPIO 输入。即使测试中未涉及相邻引脚，也应指定相邻引脚参数。建议输入与被测引脚相同的输入。

4.2.15 FS_DIO_Output_LPC()

此测试测试引脚的数字输出功能。该测试的原理是在被测引脚上设置和读取所有两个逻辑值。请输入合适的延迟参数。它必须确保时间间隔足够长，让器件达到引脚上期望的逻辑值。超低的延迟参数会导致函数的“失败”（“fail”）返回值。

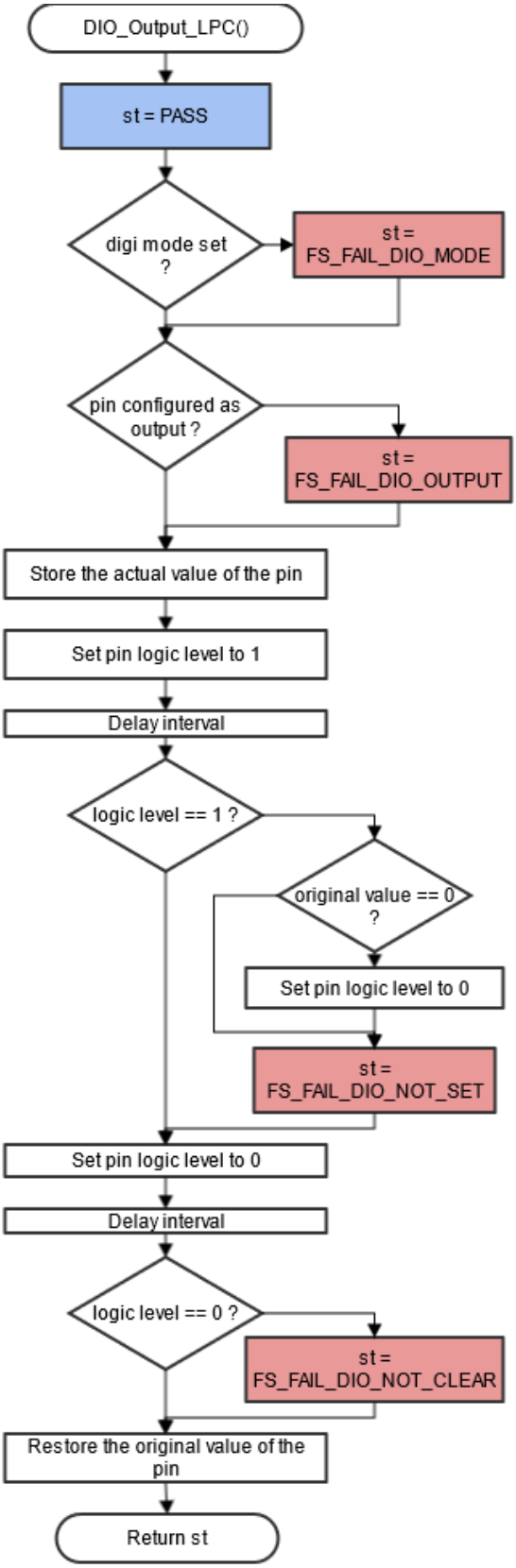

图 18. 数字输出测试的框图

函数原型：

*FS_RESULT FS_DIO_Output_LPC(fs_dio_test_lpc_t *pTestedPin, uint32_t delay);*

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

delay - 识别被测引脚上值变化所需的延迟。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_OUTPUT* - 引脚未设置为输出。
- *FS_FAIL_DIO_NOT_SET* - 引脚不能设置为逻辑 1。
- *FS_FAIL_DIO_NOT_CLEAR* - 引脚无法清除为逻辑 0。
- *FS_FAIL_DIO_MODE* - 引脚没有“数字模式”设置 - 仅用于特定的 LPC 器件。

函数总是返回第一个检测到的错误。

函数调用示例：

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

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

调用限制：

被测引脚必须配置为数字输出。为适当的功能定义适当的延迟。

4.2.16 FS_DIO_ShortToAdjSet_LPC()

此函数可确保相邻引脚短路测试所需的条件。此函数的目的是正确配置被测引脚和相邻引脚。相邻引脚是可选引脚，理论上可能对被测引脚造成短路。函数框图如图 19 所示。与电源短路测试类似，该测试需要使用两个函数。第二个（get）函数评估测试结果。*FS_DIO_InputExt_LPC()* 函数在相应章节中进行描述。指定输入测试函数的被测引脚和相邻引脚。

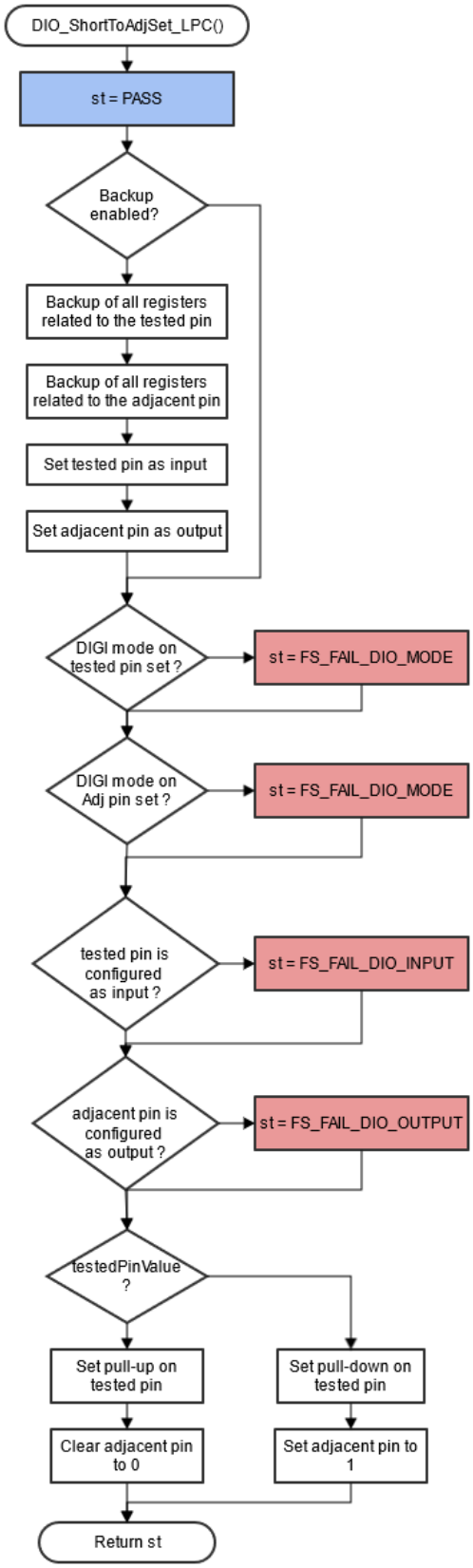

图 19. FS_DIO_ShortToAdjSet_LPC() 函数的框图

函数原型：

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

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

**pAdjPin* - 指向相邻引脚结构的指针。

testedPinValue - 在被测引脚上设置的值。

backupEnable - 标志。如果该值不为零，则表示启用/激活备份功能。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 被测引脚未设置为输入。
- *FS_FAIL_DIO_OUTPUT* - 相邻引脚未设置为输出。
- *FS_FAIL_DIO_MODE* - 被测或相邻引脚没有“数字模式”设置 - 仅用于特定的 LPC 器件。

函数总是返回第一个检测到的错误。

函数调用示例：

以下是相邻引脚短路测试的代码示例：

```
#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);
```

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

在调用函数之前，被测引脚必须配置为 GPIO 输入，相邻引脚必须配置为 GPIO 输出。如果启用了备份功能，该函数将为两个引脚都设置方向。如果没有，则配置方向（被测引脚作为输入，相邻引脚作为输出）。函数结束后，应用既不能操作被测引脚，也不能操作相邻引脚，直到为这些引脚调用 *FS_DIO_InputExt_LPC()* 函数。

4.2.17 FS_DIO_ShortToSupplySet_LPC()

此函数创建电源短路测试的第一部分，可用于测试被测引脚与硬件电源电压（Vcc、Vdd）之间或被测引脚和硬件地（GND）之间的短路。其框图如图 20 所示。测试的第二部分（结果评估）通过在相应章节中所述的 *FS_DIO_InputExt_LPC()* 函数完成。*FS_DIO_InputExt_LPC()* 函数的主要目的是在被测引脚上设置上拉（或下拉）电阻连接，还可以确保引脚是否正确配置并备份其设置（如果需要）。

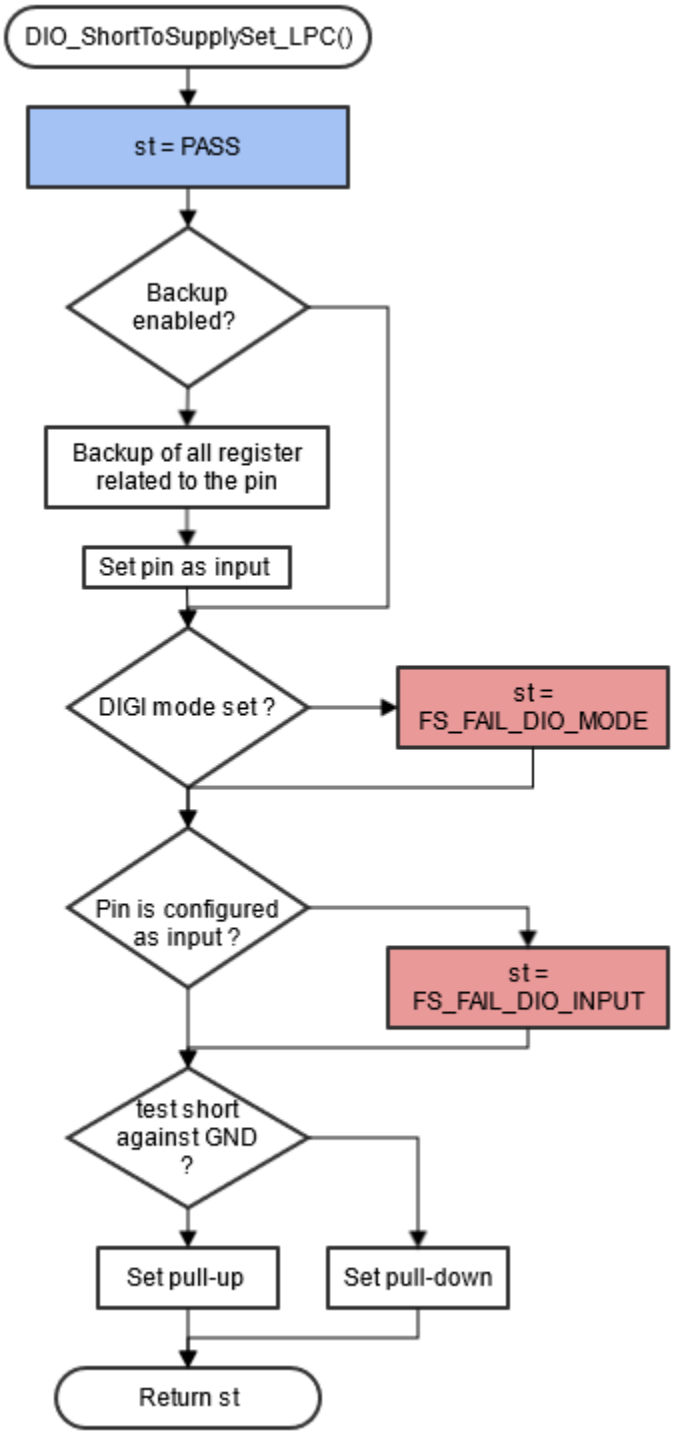

图 20. FS_DIO_ShortToSupplySet_LPC 函数的框图

函数原型：

*FS_RESULT FS_DIO_ShortToSupplySet_LPC(fs_dio_test_lpc_t *pTestedPin, bool_t shortToVoltage, bool_t backupEnable);*

函数输入：

**pTestedPin* - 指向被测引脚结构的指针。

shortToVoltage - 指定是否针对 GND 或 VDD 进行引脚短路测试。对于 GND，输入“1”。对于 VDD，输入 0 或非 0 值。

backupEnable - 标志。如果该值不为零，则表示启用/激活备份功能。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_DIO_INPUT* - 引脚未设置为输入。
- *FS_FAIL_DIO_MODE* - 引脚没有“数字模式”设置 - 仅用于特定的 LPC 器件。

函数总是返回第一个检测到的错误。

函数调用示例：

以下既是 GND 短路也是 VDD 短路情况的测试代码示例。请注意，实现的区别仅在于一个参数。如果测试了 GND 短路，则参数必须具有非零值（反之同理）。

```
#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);
```

函数性能：

有关函数性能的信息，请参阅内核自检库-源代码版本。

调用限制：

在调用函数之前，必须将被测引脚配置为 GPIO 输入。如果启用了备份功能，该函数将为被测引脚设置输入方向。如果没有启用，请配置输入方向。函数结束后，应用不能操作被测引脚，直到为被测引脚调用 *FS_DIO_InputExt_LPC()* 函数。

第 5 章
不可变存储区测试

所支持 MCU 上的不可变存储区是片上闪存。不可变存储区测试的原理是检查应用执行期间存储区内容是否有变化。为此，可以使用几种校验和的方法。校验和是一种算法，用于计算放置在被测试存储区中数据的签名。该存储模块的签名被周期性地计算，并与原始签名进行比较。

被分配的存储区的签名是在应用的链接阶段计算的。签名必须保存到不可变存储区中，但要与所计算校验和的区域不同。在运行时和复位后，必须在应用中实施相同的算法来计算校验和。对结果进行比较。如果它们不相等，则会出现安全错误状态。

在后期构建阶段计算校验和参数（签名)的算法必须与运行时使用的算法相同（用于 SW16 和 HW16 的 16 位 CRC 多项式（0x1021）或用于 HW32 和 SW32 的 0x04C11DB7），以生成 CRC 代码进行错误检测。在硬件 CRC 模块中实施同样的算法。在 IAR IDE 中，可以使用链接器计算 CRC。在其他 IDE 中，可以使用外部工具。对于 Keil uVision IDE，参见 "*在 Arm® Keil®中计算构建后 CRC*"（文档 AN12520）。

一些 MCU 有硬件 CRC 引擎，提供了一种简单的方法来计算写入其中的多个字节/字的 CRC。使用硬件进行不可变存储区测试可以提供更好的性能水平。测试的软件版本必须在没有 CRC 硬件模块的器件上使用。

5.1 符合 IEC/UL 标准的不可变存储区测试

所执行的过载测试符合 IEC 60730-1、IEC 60335、UL 60730 和 UL 1998 标准的安全要求，如表 17 所示。

表 17. 符合 IEC 和 UL 标准的不可变存储区测试

测试	元器件	故障/错误	软件/硬件 类别	可接受的测量
不可变存储区	4.1 – 不可变 存储区	所有单比特位故障	B/R.1	定期修改的校验和

5.2 不可变存储区测试的实现

测试函数对于闪存的部分放在 *iec60730b_cm4_cm7_flash.S* 中，写作汇编函数。带定义和函数原型的头文件是 *iec60730b_cm4_cm7_flash.h*。其余函数与相应的头文件一起放在 *iec60730b_invarable_memory.c* 中，用 C 语言编写。测试函数还使用以下头文件：*iec60730b.h*、*asm_mac_common.h* 和 *iec60730b_types.h*。它们是此安全库的常用头文件。

以下函数在 *iec60730b_invariable_memory.c* 中实现：

- *FS_FLASH_C_HW16_K()* / *FS_FLASH_C_HW16_K()*

以下函数在 *iec60730b_cm4_cm7_flash.S* 中实现：

- *FS_CM4_CM7_FLASH_HW16()*
- *FS_CM4_CM7_FLASH_SW16()*
- *FS_CM4_CM7_FLASH_SW32()*

以下函数在 *iec60730b_cm4_cm7_flash_dcp.c* 中实现：

- *FS_CM4_CM7_FLASH_HW32_DCP()*

硬件（*_HW）函数使用包含在所支持的 MCU 中的硬件 CRC 模块。软件函数计算 CRC 值时没有硬件支持，因此执行时间更长。

5.2.1 在应用的链接阶段计算 CRC 值

在将一个存储块写入闪存之前，必须计算其校验和。校验和计算最好使用链接器完成。然而，这并不是在所有编译器中都可以实现的。以下示例仅对 IAR IDE 有效。有关详细信息，请参阅 IAR 文档。有关在 Keil uVision IDE 中使用外部工具的信息，请参阅“*在 Arm® Keil® 中计算构建后的 CRC*”（文档 AN12520）。

CRC 计算的结果必须存储在闪存存储区中，一定不能存储在产生校验和的区域。一个好的方法是在闪存（ROM）存储区中定义一个存储校验和结果的小存储块。为此，必须修改链接器配置文件。链接器配置文件的路径为“Project（项目）> Options（选项）> linker（链接器）> Config（配置）”。文件扩展名为*.icf*。在本例中，定义了带有“.checksum”区的“CHECKSUM（校验和）”块。

```
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 };
```

CRC 计算的输入参数必须在链接器选项卡：“Project（项目）> Options （选项）> linker（链接器）”中设置。设置计算参数有两个选项。第一个选项用于计算应用中一个存储块的校验和。参数在“Checksum（校验和）”子选项卡中填写。在本例中，起始地址和结束地址分别为 0x510 和 0x3000。未使用的存储区用 0xFF 填充。校验和以 16 位存储。校验和算法是标准 0x1021 多项式的 CRC16。初始种子值为零。特定的计算数据块大小为 8 位。结果的变量为*__checksum*。

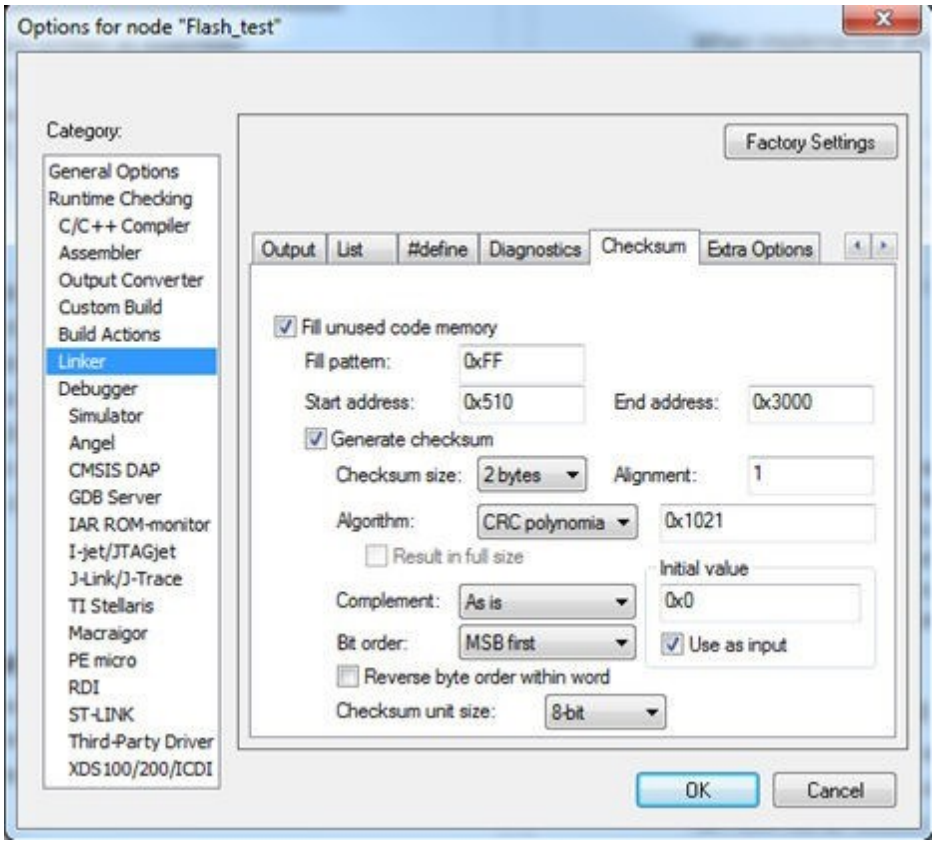

图 21. 链接器的校验和设置

常数变量名（*__checksum*）必须写入 Project（项目）>Options（选项）>Linker（链接器）>Input（输入）>Keep（保留）符号。为了在应用中使用*__checksum* 变量，必须将以下代码行放入源代码中。

```
#pragma section = ".checksum"
#pragma location = ".checksum"
extern unsigned short const __checksum;
```

如果需要对更多存储块进行 CRC 计算，请使用以下方法。链接器配置文件中定义的块中必须有足够的空间。对于本示例，计算参数与上一示例中的相同，块的地址为 0x510–0x610、0x620–0x720、0x730–0x830。变量有*__checksum_first*、*__checksum_second*、*__checksum_third*。这种情况使用了链接器命令行指令 Project（项目）>Options（选项）>linker（链接器）>Extra Options（其他选项）。使用命令行选项并输入以下命令行。取消选中“Checksum（校验和）”子选项卡的选项。

```
–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（项目）>Options（选项）>linker（链接器）>Input（输入）

将以下内容写入“保留符号”块：

```
__checksum_first
__checksum_second
__checksum_third
```

将以下行添加到源代码中，以便*__checksum_first*、*__checksum_second* 和*__checksum_third* 变量在应用中可用。

```
#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 MCU 复位后执行一次的测试

当在复位后实施或执行时间没有限制时，函数调用可如下所示：

```
#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();
```

其中：

- __checksum - 在应用链接阶段计算的 CRC 值的常数变量。
- start_address - 要测试的存储块的初始地址。
- size - 要测试的存储块的大小（首地址 – 结束地址 + 1）。
- CRC_BASE - CRC 模块的基址。
- start_seed - 启动条件种子。对于所使用的算法，它必须为“0”。

5.2.3 运行时测试

在应用运行时和执行时间有限的情况下，按顺序计算 CRC。这意味着输入参数与复位后的调用相比具有不同的含义。实现示例如下：

```
#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();
```

其中：

- *__checksum* - 在应用的后期构建阶段计算的 CRC 值的常数变量。
- *flash_crc.part_crc* - 下一次迭代的特定 CRC 结果和种子参数。
- *flash_crc.actual_address* - 要测试的存储块的实际地址。
- *CRC_BASE* - CRC 模块的基址。
- *flash_crc.block_size* - 要测试的存储块的大小。

函数的处理必须由应用开发人员执行。当在多次迭代中计算数据块的校验和时，第一次迭代（函数调用）的结果是下一次函数调用的种子值。在用测试函数处理完存储区的最后一部分后，结果就是整个被测存储块的最终校验和。

5.2.4 FS_FLASH_C_HW16_K()

此函数使用硬件 CRC 模块生成 16 位 CRC 值。

函数原型：

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

函数输入：

startAddress - 被测存储区的第一个地址。

size - 被测存储区的大小。必须能被 4 整除。

moduleAddress - CRC 模块的地址。

crcVal - 指向结果和开始条件种子变量的指针。对于第一次迭代，它通常是用户定义的值。对于后续迭代，它是上一次函数调用的结果（CRC-16-CITT-正常 0x1021）。

函数输出：

FS_RESULT

- FS_FAIL_FLASH_NULL_POINTER_C - *moduleAddress* 或 *crcVal* 输入参数为 NULL。
- FS_FAIL_FLASH_MODULO_C - 参数大小未对齐到 4 字节。
- FS_FAIL_FLASH_SIZE_C - *size* 输入参数为 0。

函数性能：

此函数参数在 LPC55S36 上以 150MHz 的时钟频率测量。

函数大小为 96 B。

函数的运行时间取决于所定义的数据块的大小。表 18 显示了几个示例：

表 18. FS_FLASH_C_HW16_K() 的运行时间取决于被测数据块的大小

数据块大小（字节）	执行时间（大约时间）
0x10	1,6 µs
0x20	1,92 µs
0x100	6,68 µs

调用限制：

该函数不能被某个改变硬件 CRC 模块内容或设置的函数中断。

5.2.5 FS_FLASH_C_HW16_L()

此函数使用硬件 CRC 模块生成 16 位 CRC 值。

函数原型：

*FS_RESULT FS_FLASH_C_HW16_L(uint32_t startAddress, uint32_t size, FS_CRC_L_Type *moduleAddress, uint16_t *crcVal);*

函数输入：

startAddress - 被测存储区的首地址。

size - 被测存储区的大小。

moduleAddress - CRC 模块的地址。

crcVal - 指向结果和开始条件种子变量的指针。对于第一次迭代，它通常是用户定义的值。对于后续迭代，它是上一次函数调用的结果（CRC-16-CCITT-正常 0x1021）。

函数输出：

FS_RESULT

- FS_FAIL_FLASH_NULL_POINTER_C - *moduleAddress* or 或 *crcVal* 输入参数为 NULL。
- FS_FAIL_FLASH_SIZE_C - *size* 输入参数为 0。

函数性能：

函数参数在 LPC54S018M 上以 96MHz 的时钟频率测量。

函数大小为 66 B。

函数的运行时间取决于定义的数据块的大小。表 19 显示了几个示例：

表 19. FS_FLASH_C_HW16_L() 的运行时间取决于被测数据块的大小

模块大小（字节）	执行时间（大约时间）
0x10	14,36 µs
0x20	18,04 µs
0x100	44,12 µs

调用限制：

该函数不能被某个改变硬件 CRC 模块内容或设置的函数中断。

5.2.6 FS_CM4_CM7_FLASH_HW16()

此函数使用硬件 CRC 模块生成 16 位 CRC 值。

函数原型：

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

函数输入：

startAddress - 被测存储区的首地址。

size - 被测存储区的大小。

moduleAddress - CRC 模块的地址。

crcVal - 开始条件种子。对于第一次迭代，它通常是用户定义的值。对于后续迭代，它是上一次函数调用的结果。

函数输出：

uint16_t - 该存储区范围的 16 位 CRC 值（CRC-16-CITT - 正常 0x1021）。

函数性能：

函数大小为 44 字节。[1]

函数的运行时间取决于定义的数据块的大小。下表显示了几个示例。

表 20. FS_CM4_CM7_FLASH_HW16()的运行时间依赖于被测数据块的大小

模块大小（字节）	时钟周期	执行时间（大约时间）
0x10	208	2.6 µs
0x20	343	4.2 µs
0x50	745	9.3 µs

调用限制：

该函数不能被某个更改硬件 CRC 模块内容或设置的函数中断。

5.2.7 FS_CM4_CM7_FLASH_HW32_DCP()

此函数使用硬件 DCP 模块生成 32 位 CRC 值。

函数原型：

*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);*

函数输入：

startAddress - 被测存储区的首地址。

size - 被测存储区的大小。

moduleAddress - CRC 模块的地址。

crcVal - 开始条件种子。对于第一次迭代，它通常是用户定义的值。对于后续迭代，它是上一次函数调用的结果。

channel - 用于计算的 DCP 通道。

psDCPState - 每个 DCP 通道的状态和结果结构。

tag - 区分同一通道上的计算。

函数输出：

uint32_t - 该存储区范围的 32 位 CRC 值（CRC-32/MPEG-2 - 0x04C11DB7）。

函数调用的示例：

xxxxxxx

```
/* Start CRC calculation for a given block of Flash memory */
    FS_CM4_CM7_FLASH_HW32_DCP(psFlashCrc->actualAddress, psFlashCrc->blockSize,
(uint32_t)FLASH_USED_DCP,
                              psFlashCrc->partCrc, g_dcpSafetyChannel,
psFlashDCPState, FLASH_DCP_TAG);
```

```
    /* Check error. */
    if (psFlashDCPState->CH3State == FS_FAIL_FLASH_DCP)
    {
        psSafetyCommon->safetyErrors |= FLASH_TEST_ERROR;
        SafetyErrorHandling(psSafetyCommon);
    }
    /* Check if calculation finished */
    else if (psFlashDCPState->CH3State == FS_FLASH_DCP_AVAILABLE)
    {
        /* Store partial result */
        psFlashCrc->partCrc = psFlashDCPState->CH3Result;
```

函数性能：

函数大小为 448 字节。[4]

函数的运行时间取决于定义的数据块的大小。下表显示了几个示例。

表 21. FS_CM4_CM7_FLASH_HW32_DCP() 的运行时间取决于被测模块的大小

模块大小（字节）	时钟周期	执行时间（大约时间）
0x10	57	2.375 µs
0x20	57	2.375 µs
0x50	67	2.791 µs
0x500	261	10.875 µs

调用限制：

该函数不能被某个更改硬件 DCP 模块内容或设置的函数中断。

支持在同一通道使用不同标签号进行多个计算，但必须将其放置在同一执行模块中，例如通道 0 计算在 Systick ISR 中，通道 1 计算在 while 循环中。

计算的数据块必须对齐到 4 个字节。

5.2.8 FS_CM4_CM7_FLASH_SW16()

此函数使用软件生成 16 位 CRC 值。

函数原型：

uint16_t FS_CM4_CM7_FLASH_SW16(uint32_t startAddress, uint32_t size, uint32_t moduleAddress, uint16_t crcVal);

函数输入：

startAddress - 被测存储区的首地址。

size - 被测存储区的大小。

moduleAddress - 没有作用。只为兼容 HW 函数。

crcVal - 开始条件种子。对于第一次迭代，它通常是用户定义的值。对于后续迭代，它是上一次函数调用的结果。

函数输出：

uint16_t - 该存储区范围的 16 位 CRC 值（CRC-16-CITT-正常 0x1021）。

函数性能：

函数大小为 54 字节。[1]

函数的运行持续时间取决于定义的数据块的大小。下表显示了几个示例。

表 22. FS_CM4_CM7_FLASH_SW16() 的运行时间取决于被测模块的大小

模块大小（字节）	时钟周期	执行时间（大约时间）
0x10	1934	24.175 µs
0x20	3936	49.2 µs
0x50	9758	121.975 µs

调用限制：

无

5.2.9 FS_CM4_CM7_FLASH_SW32()

此函数使用软件生成 32 位 CRC 值。

函数原型：

uint32_t FS_CM4_CM7_FLASH_SW32(uint32_t startAddress, uint32_t size, uint32_t moduleAddress, uint32_t crcVal);

函数输入：

startAddress - 被测存储区的首地址。

size - 被测存储区的大小。

moduleAddress - 没有作用。只为兼容 HW 函数。

crcVal - 开始条件种子。对于第一次迭代，它通常是用户定义的值。对于后续迭代，它是上一次函数调用的结果。

函数输出：

uint32_t - 该存储区范围的 32 位 CRC 值（CRC-32/MPEG-2 - 0x04C11DB7）。

函数性能：

函数大小为 78 字节。[1]

函数的运行时间取决于定义的数据块的大小。下表显示了几个示例。

表 23. FS_CM4_CM7_FLASH_SW32() 的运行时间取决于被测模块的大小

模块大小（字节）	时钟周期	执行时间（大约时间）
0x10	1795	22.438 µs
0x20	3631	45.388 µs
0x50	9030	112.875 µs

调用限制：

无

第 6 章
CPU 程序计数器测试

CPU 程序计数器寄存器测试流程测试 CPU 程序计数器寄存器是否处于卡滞状态。程序计数器寄存器测试可以在 MCU 复位后执行一次，也可在运行时执行。

如果 CPU 程序计数器寄存器无法正常工作，则通过特定的 FAIL 返回确保安全错误的识别。评估测试函数的返回值。如果返回值等于 FAIL 返回，则跳转到安全错误处理函数。安全错误处理函数可能特定于应用，不是库的一部分。此函数的主要目的是将应用置于安全状态。

与其他 CPU 寄存器相反，程序计数器不能简单地用测试模数填充。有必要强制 CPU（程序流）访问正在测试某个模数的相应地址，以验证程序计数器的功能。

程序计数器测试无需初始化函数即可工作。某个短函数（另一个对象）被写入单独的文件中。通过在链接器配置文件中声明该对象，将其放置到闪存存储区的适当地址。测试函数使用此例程地址和 RAM 存储器中的适当地址来测试程序计数器。

程序计数器寄存器测试的框图如下图所示：

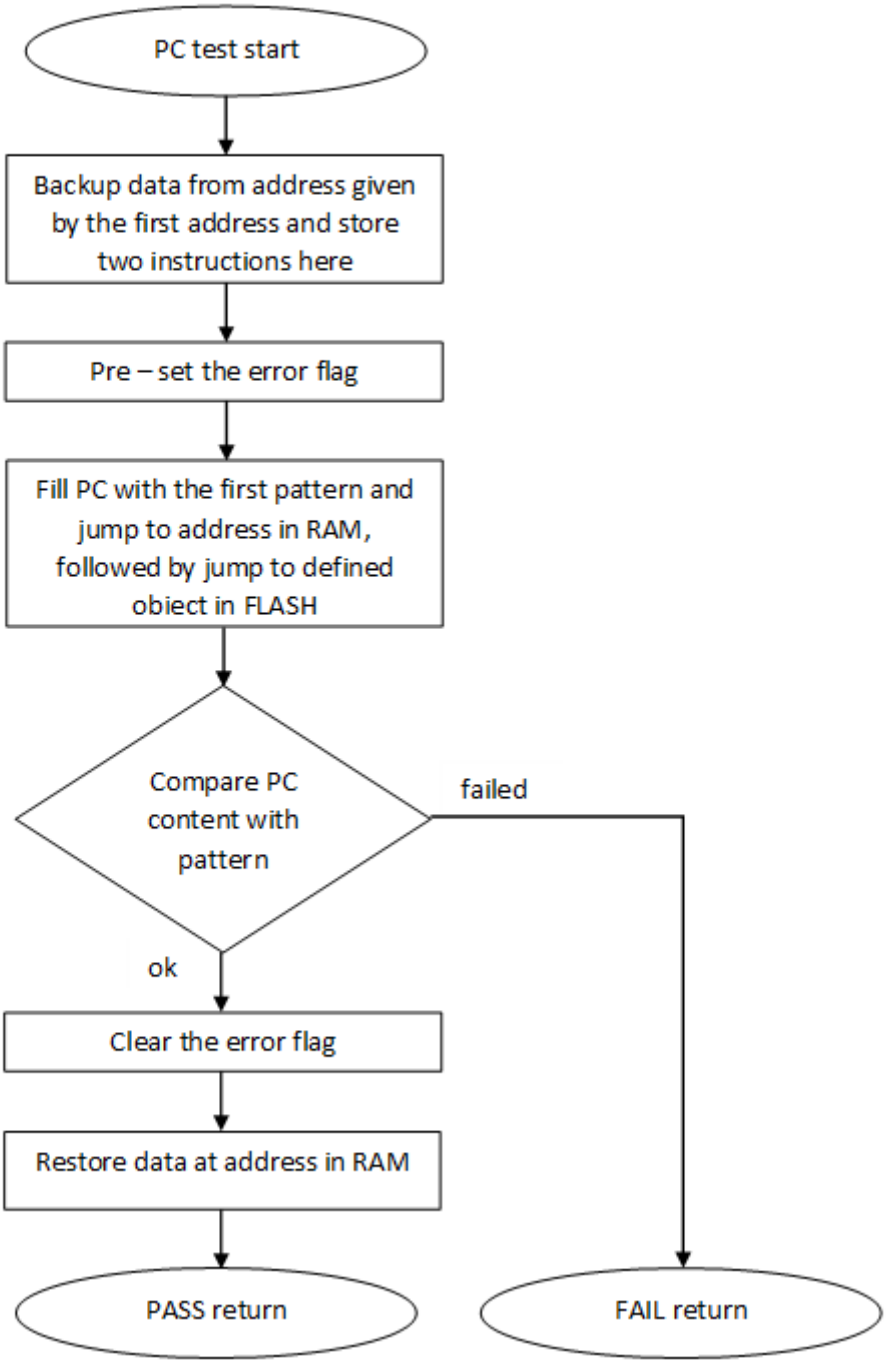

图 22. PC_Test 框图

6.1 符合 IEC/UL 标准的 CPU 程序计数器测试

所执行的过载测试符合 IEC 60730-1、IEC 60335、UL 60730 和 UL 1998 标准的安全要求，如下表所示：

表 24. 符合 IEC/UL 标准的 CPU 程序计数器测试

测试	元器件	故障/错误	软件/硬件 类别	可接受的测量
CPU	CPU（1.3 – 程序计数器）	卡滞	B/R.1	定期自检

6.2 CPU 程序计数器测试的实现

CPU 寄存器的测试函数在 *iec60730b_cm4_cm7_pc.S* 文件中，写作汇编函数。带测试模数和函数原型的头文件是 *iec60730b_cm4_cm7_pc.h*。*iec60730b.h*、*asm_mac_common.h* 和 *iec60730b-types.h* 是本安全库的常用头文件。对于第二种测试类型，*iec60730b_cm4_cm7_c_object.S* 文件必须放在闪存存储区的适当地址。

PC 测试实现的示例：

在应用中处理的唯一函数如下所示：

FS_CM4_CM7_PC_Test()

将适当的模数作为第一个输入。如果需要，可以使用不同的模数按顺序多次调用该函数。注意，测试模数必须是 RAM 中的真实地址，并且必须是偶数。将 *iec60730b_cm4_cm7_c_object.S* 文件放入闪存存储区的适当地址。

下面是函数调用的示例：

```
#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()

程序计数器寄存器根据图 22 进行测试。

函数原型：

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

函数输入：

pattern1 - 来自 RAM 存储器的地址，足以作为程序计数器的模数。

pObjectFunction - *FS_PC_Object()* 函数的地址。

**pFlag* - 用作标志的变量的地址/存储区的位置。如果标志为“0”，则测试成功（如果失败，则为“1”）。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_PC* - 如果测试执行不正确，PC_flag 的值为“1”。

函数性能：

该函数大约需要 92 个周期（1.15µs）。[1]

函数大小为 48 B。[1]

调用限制：

该函数不可被中断。

6.2.2 FS_PC_Object()

该函数由 *FS_CM4_CM7_PC_Test()* 函数内部使用，用于执行程序计数器测试。它只能由 *FS_CM4_CM7_PC_Test()* 函数调用，应该被放置在一个可靠的地址（通过编辑链接器文件）。

此示例展示了如何将函数放置到 IAR IDE 的链接器配置文件中所需的地址：

```
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};
```

函数原型：

void FS_PC_Object(void);

函数输入：

空

函数输出：

空

函数性能：

函数的运行时间包含在 *FS_CM4_CM7_PC_Test()* 函数的运行时间中。它的大小是 16 字节。

调用限制：

此函数用于执行 PC 测试，只能由 *FS_CM4_CM7_PC_Test()* 函数调用。

第 7 章
可变存储区测试

对于所支持器件的可变存储区测试检查片上 RAM 是否存在 DC 故障，还可以测试应用栈区。March C 和 March X 方案被用作控制机制。选择是使用 March C 方案还是 March X 方案。复位后测试和运行时测试的处理函数不同。这两个函数都必须在 RAM 中定义一个备份区域，由开发人员预留。该区域的大小必须至少与被测块的大小相同。RAM 测试被认为是破坏性的。这是因为存储区的数据（变量、栈区域和放置在 RAM 中的函数等）会被移动，多次重写（使用测试模数 0x55555555 和 0xAAAAAAAA），然后移回原始存储区域。测试流程非常敏感，不能被中断。RAM 测试的框图如下图所示：

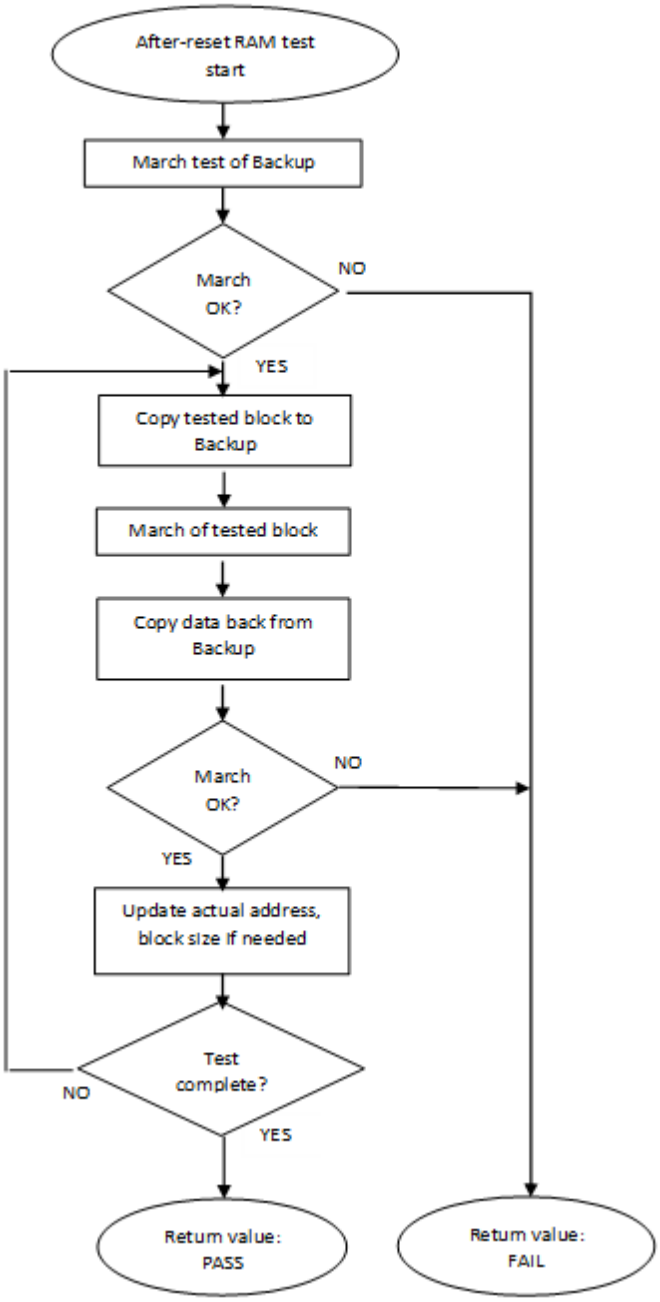

图 23. RAM 复位后测试的框图

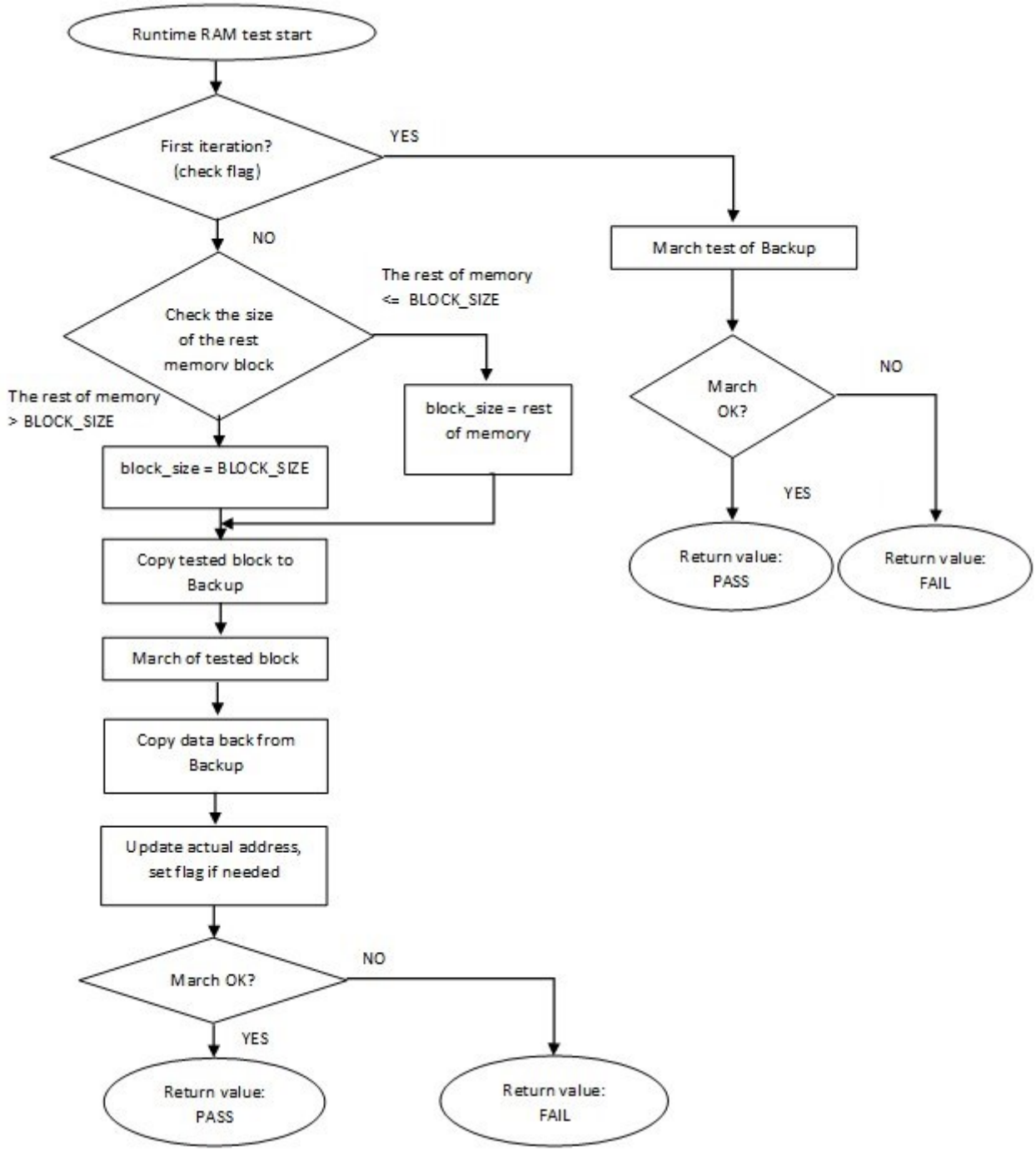

图 24. RAM 运行时测试的框图

7.1 符合 IEC/UL 标准的可变存储区测试

所执行的过载测试符合 IEC 60730-1、IEC 60335、UL 60730 和 UL 1998 标准的安全要求，如下表所述：

表 25. 符合 IEC 和 UL 标准的可变存储区测试

测试	元器件	故障/错误	软件/硬件 类别	可接受的测量
可变存储区	4.2 – 可变存储区	DC 故障	B/R.1	周期性的 March 测试

7.2 可变存储区测试的实现

可变存储区（RAM）测试的测试函数在 *iec60730b_cm4_cm7_RAM.S* 文件中，写作汇编函数。带返回值和函数原型的头文件是 *iec60730b_cm4_cm7\uram.h*。*iec60730b.h*、*asm_mac_common.h* 和 *iec60730b-types.h* 是本安全库的常用头文件。

RAM 测试包括以下公共函数：

- *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()*

前两个函数提供了复杂的 RAM 测试。您不必直接使用后面的函数。

7.2.1 FS_CM4_CM7_RAM_AfterReset()

复位后测试由 *FS_CM0_RAM_AfterReset()* 函数完成。如果执行时间不紧张，复位后调用此函数一次。为备份区域预留空闲存储区空间。块大小参数不能大于备份区域的大小。该函数首先检查备份区域。然后循环开始。将存储块复制到备份区域，并使用相应的 March 测试检查其位置。将数据复制回原始存储区域，并按照块的大小更新实际的地址。重复此操作，直到测试完最后一个存储块。如果检测到 DC 故障，该函数将返回故障模式。框图如图 23 所示。

下面是函数调用的示例：

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

函数原型：

FS_RESULT FS_CM4_CM7_RAM_AfterReset(uint32_t startAddress, uint32_t endAddress, uint32_t blockSize, uint32_t backupAddress, tFcn pMarchType);

函数输入：

startAddress - 被测 RAM 区域的首地址。

endAddress - 被测 RAM 区域后的第一个字节的地址。

blockSize - 被测块的大小。

backupAddress - 备份区域的地址。

**pMarchType* - March 函数（March X 或 March C）的地址。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_RAM*

函数性能：

函数大小为 98 B。[1]

执行时间取决于存储区大小，也因块大小和使用的 March 方法而异。[1]

表 26. FS_CM4_CM7_RAM_AfterReset 运行时间

存储区大小（字节）	块大小（字节）	周期数 - March X	周期数 – 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

调用限制：

如果执行时间不紧张，MCU 复位后使用此函数一次。

它不能被打断。

备份区域的大小必须至少与“block_size”参数定义的被测块大小相同。

7.2.2 FS_CM4_CM7_RAM_Runtime()

该运行时测试由 *FS_CM4_CM7_RAM_Runtime()* 函数完成。为备份区域预留空闲存储区空间。块大小参数不能大于备份区域的大小。在第一次调用期间，该函数检查备份区域。调用后，存储模块按顺序处理。将它们复制到备份区域，并使用相应的 March 测试检查其位置。将数据复制回原始存储区域，并更新实际地址和模块大小。重复这一过程，直到测试完最后一个存储块。如果检测到 DC 故障，该函数将返回故障模式。框图如上图所示。函数调用的示例如下所示：

```
#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();
```

函数原型：

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

函数输入：

startAddress - 被测 RAM 区域的首地址。

endAddress - 被测 RAM 区域后第一个字节的地址。

**pActualAddress* - 保存实际地址值的变量的地址。

blockSize - 被测模块的大小。

backupAddress - 备份区域的地址。

**pMarchType* - March 函数的地址（March X 或 March C）。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_RAM*

函数性能：

函数大小为 118B。[1]

执行时间取决于块的大小，对于 March C 和 March X 方法是不同的。[1]

表 27. FS_CM4_CM7_RAM_Runtime 运行时间

块大小（字节数）	周期数 - March X	周期数 - March C
0x4	202	187
0x8	250	298
0x20	532	688
0x40	908	1208

调用限制：

此函数不可被中断。

备份区域的大小必须至少与 block_size 参数定义的被测块大小相同。

执行时间取决于块的大小。

7.2.3 FS_CM4_CM7_RAM_CopyFromBackup()

该函数将存储块从备份区域复制到专用位置。

函数原型：

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

函数输入：

startAddress - 目标的首地址。

blockSize - 存储块的大小。

backupAddress - 备份区域的地址。

函数输出：

空

函数性能：

函数大小为 16 B。[1]

7.2.4 FS_CM4_CM7_RAM_CopyToBackup()

此函数将存储块复制到专用备份区域。

函数原型：

void FS_CM4_CM7_RAM_CopyToBackup(uint32_t startAddress, uint32_t blockSize, uint32_t backupAddress);

函数输入：

startAddress - 源的首地址。

blockSize - 存储块的大小。

backupAddress - 备份区域的地址。

函数输出：

空

函数性能：

函数大小为 16 B。[1]

7.2.5 FS_CM4_CM7_RAM_SegmentMarchC()

此函数对给定起始地址和块大小的存储块执行 March C 测试。执行此函数后，被测存储区的内容会保留改变。

函数原型：

FS_RESULT FS_CM4_CM7_RAM_SegmentMarchC(uint32_t startAddress, uint32_t blockSize);

函数输入：

startAddress - 被测存储块的首地址。

blockSize - 被测存储块的大小。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_RAM*

函数性能：

函数大小为 118 B。[1]

7.2.6 FS_CM4_CM7_RAM_SegmentMarchX()

此函数对给定起始地址和块大小的存储块执行 March X 测试。执行此函数后，被测存储区的内容会保留改变。

函数原型：

FS_RESULT FS_CM4_CM7_RAM_SegmentMarchX(uint32_t startAddress, uint32_t blockSize);

函数输入：

startAddress - 被测存储块的首地址。

blockSize - 被测存储块的大小。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_RAM*

函数性能：

函数大小为 98B。[1]

第 8 章
CPU 寄存器测试

CPU 寄存器测试流程测试所有 CM4/CM7 CPU 寄存器是否处于卡滞状态（程序计数器寄存器除外）。程序计数器测试作为一个独立的安全例程实现。在 MCU 复位后以及运行时执行一组测试。这组测试包括对以下寄存器的测试：

通用寄存器：

- R0-R12 寄存器

栈指针寄存器：

- SP main 寄存器
- SP process 寄存器

特殊寄存器：

- APSR 寄存器
- 控制寄存器
- PRIMASK 寄存器
- FAULTMASK 寄存器
- BASEPRI 寄存器

链接寄存器：

- LR 寄存器

FPU 寄存器：

- FPSCR 寄存器
- S0 – S31 寄存器

如果某些寄存器有卡滞故障，则通过特定的 FAIL 返回来确保安全错误的识别。评估每个函数的返回值。如果该值等于 FAIL 返回值，则应跳转到安全错误处理函数。安全错误处理函数可能是特定于应用的，不是库的一部分。此函数的主要目的是将应用置于安全状态。

在某些特殊情况下，FAIL 返回不会报告错误，因为它可能会需要内容损坏寄存器的操作。在这种情况下，函数在一个无限循环中等待复位。

CPU 寄存器卡滞错误测试的原理是在每个寄存器中写入和比较两个测试模数。将寄存器的内容与该常量或写入在另一个被测试过的寄存器中的值进行比较。大多数情况下，R0、R1 和 R2 被用作辅助寄存器。模数被定义为检查所有寄存器位中的逻辑 1 和逻辑 0 值。

对于 PRIMASK 和 CONTROL 以及 FAULTMASK 和 BASEPRI 测试，必须备份原始内容。对于 SP_main 和 SP_process 测试，必须备份 CONTROL 寄存器的内容。如果是 FPU 寄存器测试，FPSCR 的内容应备份。CPACR 系统寄存器包含一个用于使能 FPU 的位。各寄存器的框图如以下各图所示：

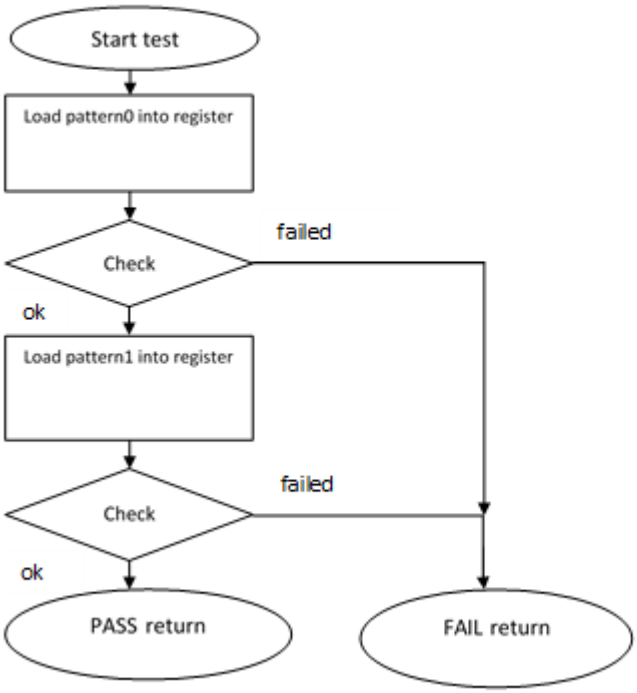

图 25. R2–R12 寄存器测试的框图

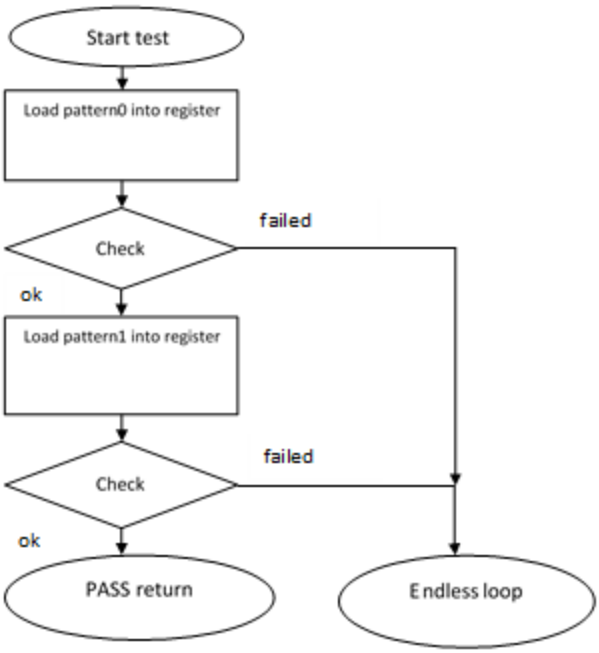

图 26. R0、R1、LR、APSR 寄存器测试的框图

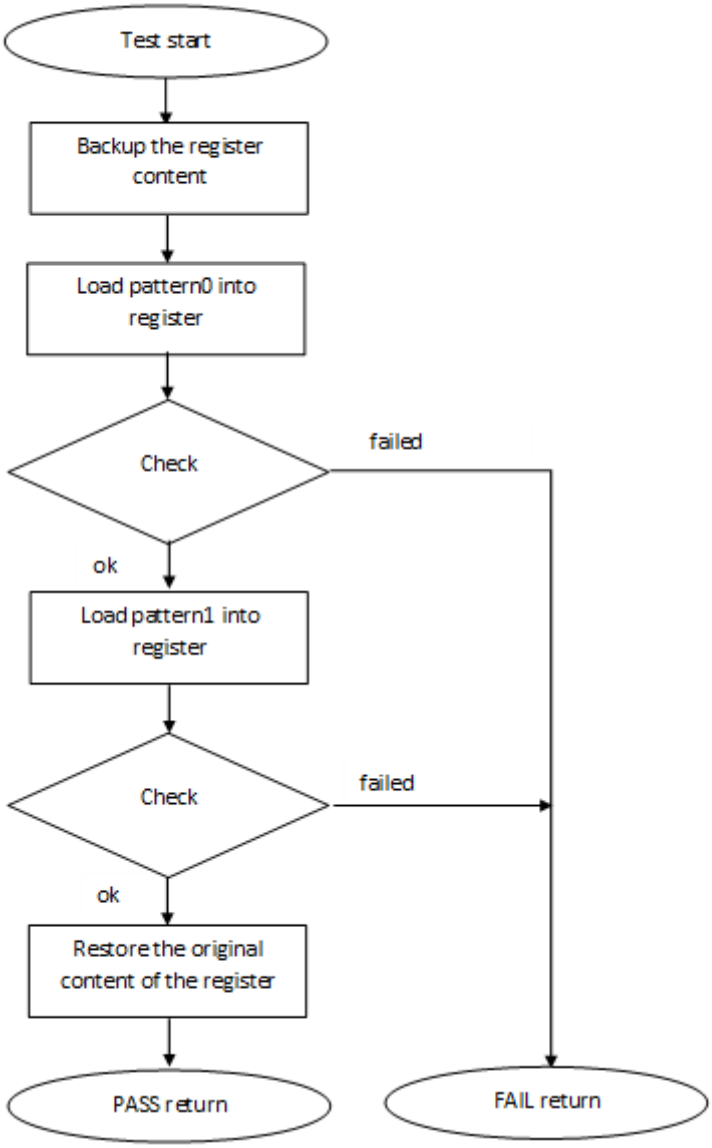

图 27. PRIMASK、FAULTMAST、BASEPRI 和 CONTROL 寄存器测试的框图

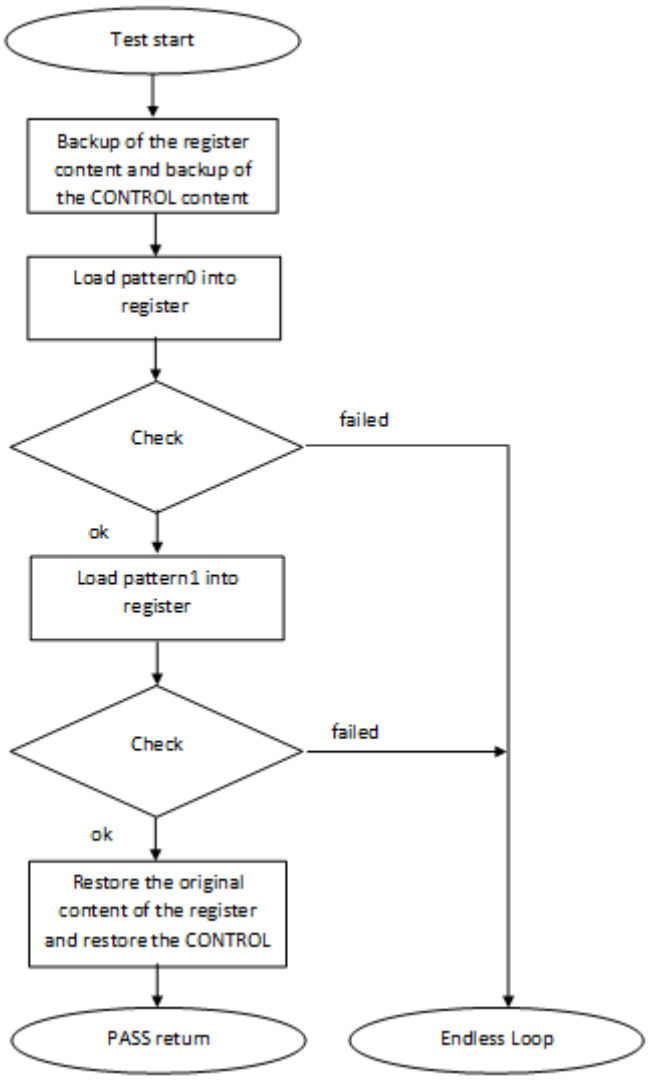

图 28. SP_main 和 SP_process 寄存器测试的框图

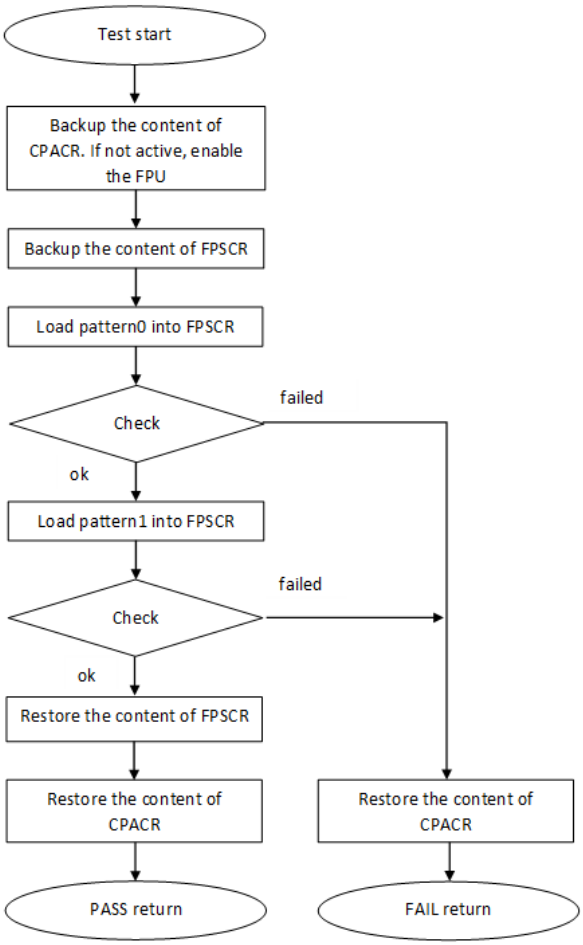

图 29. FPSCR 寄存器测试的框图

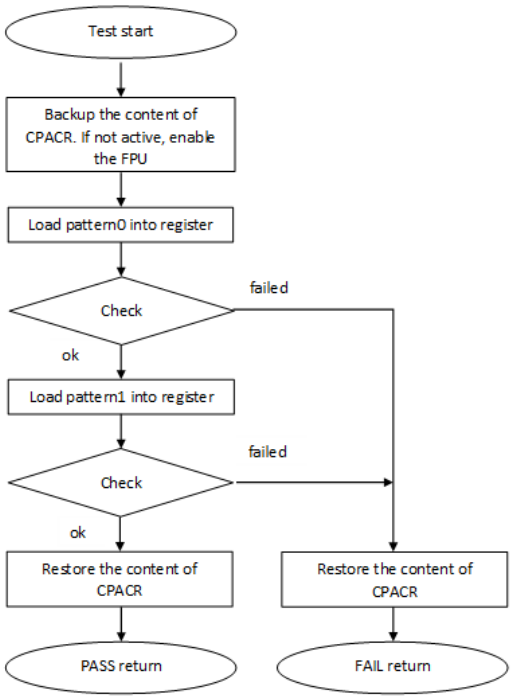

图 30. S0-S31 寄存器测试的框图

8.1 符合 IEC/UL 标准的 CPU 寄存器测试

所执行的过载测试符合 IEC 60730-1、IEC 60335、UL 60730 和 UL 1998 标准的安全要求，如下表所述：

表 28. 符合 IEC 和 UL 标准的 CPU 寄存器测试

测试	元器件	故障/错误	软件/硬件 类别	可接受测量
CPU 寄存器测试	CPU（1.1 – 寄存器）	卡滞	B/R.1	定期自检

8.2 CPU 寄存器测试的实现

CPU 寄存器的测试函数位于 *iec60730b_cm4_cm7_reg.S* 文件中，以汇编函数编写。对于包含 FPU 的器件，*iec60730b_cm4_cm7_reg_fpu.S* 是一个附加文件，其中包含 FPU 相关寄存器的测试。带返回值和函数原型的头文件是 *iec60730b_cm4_cm7_reg.h*。

iec60730b.h、*asm_mac_common.h* 和 *iec60730b_types.h* 是本安全库的常用头文件。

调用以下函数来测试相应的寄存器：

- *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()*

当器件有 FPU 时，*iec60730b_ cm4_cm7_reg_fpu.S* 中包含以下函数：

- *FS_CM4_CM7_CPU_ControlFpu()*
- *FS_CM4_CM7_CPU_Float1()*
- *FS_CM4_CM7_CPU_Float2()*

错误检测通过特定的返回值识别，如后续章节所述。有几个例外情况。如果 R0、R1、LR、APSR 和 SP 寄存器有损坏，则应用将处于无限循环中，而不是返回错误值。如果这些寄存器中的某些有损坏，应用无法进行标准操作来识别安全错误（进行比较、从函数中转移出来，或返回一个值）。

函数在复位后和运行时的使用是相同的。在运行时使用函数时要小心，如后续章节所述。

以下是函数调用的示例：

```
#include “iec60730b.h”
if (FS_FAIL_CPU_REGISTER == FS_CM4_CM7_CPU_Register())
SafetyError();
```

8.2.1 FS_CM4_CM7_CPU_Control()

此函数根据图 27 测试 CONTROL 寄存器。

函数原型：

FS_RESULT FS_CM4_CM7_CPU_Control(void);

测试模数：

CONTROL：0x00000000, 0x00000002

函数输入：

空

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_CONTROL*

函数性能：

该函数需要大约 30 个周期，包括结果比较（0.375µs）。[1]

函数大小为 48 B。[1]

调用限制：

此函数不可被中断。

8.2.2 FS_CM4_CM7_CPU_ControlFpu()

该函数根据图 27 测试 CONTROL 寄存器。

函数原型：

FS_RESULT FS_CM4_CM7_CPU_ControlFpu(void);

测试模数：

CONTROL：0x00000000, 0x00000002, 0x00000004

函数输入：

空

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_CONTROL*

函数性能：

该函数大约需要 52 个周期，包括结果比较（0.65µs）。[1]

函数大小为 62 B。[1]

调用限制：

该函数不可被中断。

该函数应该用于带 FPU 的器件，以取代 *FS_CM4_CM7_CPU_Control()* 函数。

8.2.3 FS_CM4_CM7_CPU_Float1()

该函数根据图 29 和图 30 检查 FPSCR 和 S0-S15 寄存器。在该函数中，FPU 在 CPACC 寄存器中启用。在函数结束时，恢复 CPACR 的原始内容。

函数原型：

FS_RESULT FS_CM4_CM7_CPU_Float1(void);

各寄存器的测试模数：

FPSCR：0x55400015, 0xA280008A

S0-S15：0x55555555, 0xAAAAAAAA

函数输入：

空

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_FLOAT_1*

函数性能：

该函数大约需要 286 个周期（3.575µs)。[1]

函数大小为 476 B。[1]

调用限制：

仅适用于带有浮点运算单元（FPU）的器件。

8.2.4 FS_CM4_CM7_CPU_Float2()

该函数根据图 30 检查 S16-S31 寄存器。在该函数中，FPU 在 CPACC 寄存器中启用。在函数结束时，恢复 CPACR 的原始内容。

函数原型：

FS_RESULT FS_CM4_CM7_CPU_Float2(void);

各寄存器的测试模数：

S0-S15：0x55555555, 0xAAAAAAAA

函数输入：

空

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_FLOAT_2*

函数性能：

该函数大约需要 270 个周期（3.375µs）。[1]

函数大小为 470 B。[1]

调用限制：

仅适用于带浮点运算单元（FPU）的器件。

8.2.5 FS_CM4_CM7_CPU_NonStackedRegister()

此函数测试 R8、R9、R10 和 R11 CPU 寄存器。每个寄存器都按照图 25 进行测试。

函数原型：

FS_RESULT FS_CM4_CM7_CPU_NonStackedRegister(void);

各寄存器的测试模数：

R8 – R11：0x55555555, 0xAAAAAAAA

函数输入：

空

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_NONSTACKED_REGISTER*

函数性能：

该函数需要大约 70 个周期，包括结果比较（0.875µs）。[1]

函数大小为 80 B。[1]

调用限制：

无

8.2.6 FS_CM4_CM7_CPU_Primask()

此函数根据图 27 测试 PRIMASK 寄存器。

函数原型：

FS_RESULT FS_CM4_CM7_CPU_Primask(void);

测试模数：

PRIMASK：0x00000001, 0x00000000

函数输入：

空

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_PRIMASK*

函数性能：

该函数大约需要 221 个周期，包括结果比较（2.763µs）。[1]

函数大小为 44 B。[1]

调用限制：

当全局中断被禁用时，该函数不能被中断打断。

8.2.7 FS_CM4_CM7_CPU_Register()

此函数按顺序测试 R0-R7、R12、LR 和 APSR CPU 寄存器。每个寄存器都根据图 25 和图 26 进行测试。

函数原型：

FS_RESULT FS_CM4_CM7_CPU_Register(void);

各寄存器的测试模数：

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

APSR：0x50000000, 0xA0000000

函数输入：

空

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_REGISTER*

如果 R0、R1、LR 或 APSR 寄存器损坏，则函数处于无限循环中，中断被禁用。此状态必须通过另一个安全机制（例如看门狗）来监控。

函数性能：

函数的运行时间约为 172 个周期，包括结果比较（2.15µs）。[1]

函数大小为 204 个字节。[1]

调用限制：

无

8.2.8 FS_CM4_CM7_CPU_Special()

该函数根据图 27 测试 BASEPRI 和 FAULTMASK 寄存器。

函数原型：

FS_RESULT FS_CM4_CM7_CPU_Special(void);

测试模数：

BASEPRI：0xA0, 0x50

FAULTMASK：0x1, 0x0

函数输入：

空

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_SPECIAL*

函数性能：

该函数大约需要 61 个周期（0.763µs）。[1]

函数大小为 104 B。[1]

调用限制：

无

8.2.9 FS_CM4_CM7_CPU_Special8PriorityLevels()

该函数根据图 27 测试 BASEPRI 和 FAULTMASK 寄存器。

函数原型：

FS_RESULT FS_CM4_CM7_CPU_Special8PriorityLevels(void);

测试模数：

BASEPRI：0xA0, 0x40

FAULTMASK：0x1, 0x0

函数输入：

空

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_CPU_SPECIAL*

函数性能：

该函数大约需要 53 个周期（1.104µs）。[6]

函数大小为 84 B。[x]

调用限制：

面向具有 8 个中断优先级的器件。

8.2.10 FS_CM4_CM7_CPU_SPmain()

此函数根据图 28 测试 SP_main 寄存器。

函数原型：

FS_RESULT FS_CM4_CM7_CPU_SPmain(void);

测试模数：

SP_main：0x55555554, 0xAAAAAAA8

函数输入：

空

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*

如果 SP_main 已损坏，则该函数处于无限循环中，中断被禁用。此状态必须通过另一个安全机制（例如看门狗）来监控。

函数性能：

该函数需要大约 59 个周期，包括结果比较（0.738µs）。[1]

函数大小为 58 B。[1]

调用限制：

此函数不可被中断。

8.2.11 FS_CM4_CM7_CPU_SPprocess()

此函数根据图 28 测试 SP_process 寄存器。

函数原型：

FS_RESULT FS_CM4_CM7_CPU_SPprocess(void);

测试模数：

SP_process：0x55555554, 0xAAAAAAA8

函数输入：

空

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*

如果 SP_process 已损坏，则该函数处于无限循环中，中断被禁用。此状态必须通过另一个安全机制（例如看门狗）来监控。

函数性能：

该函数需要大约 51 个周期，包括结果比较（0.638µs）。[1]

函数大小为 58 B。[1]

调用限制：

此函数不可被中断。

第 9 章
栈测试

此测试例程用于测试应用的栈的溢出和下溢条件。可变存储区测试涵盖了栈所占用存储区域卡滞故障的测试。如果栈被错误控制，或者为给定应用定义“过低”的栈区域，则可能发生栈溢出或栈下溢。

该测试的原理是用已知的模数填充栈下方和上方的区域。这些区域必须与栈一起在链接器配置文件中定义。然后，初始化函数用这些模数填充这些区域。该模数的值不能出现在应用的其他地方。测试在复位后和应用运行时以相同的方式执行。目的是检查在这些区域中是否仍然写着正确的模数。如果没有，则表示栈行为不正确。如果有，则测试函数的 FAIL 返回值必须作为安全错误进行处理。

9.1 符合 IEC/UL 标准的栈测试

此栈测试是一项附加测试，在 IEC60730 附录 H 表中未直接规定。

9.2 链接器设置

应用栈的大小和位置通常在链接器配置文件中定义。因此，还必须在栈的下方定义相应的区域。还有其他方法可以实现这一点，但这里只显示了一个示例。区域的大小必须是 0x4 的倍数。最小为 0x4。

```
define symbol __ICFEDIT_region_RAM_start_   = 0x1FFFFC10;
define symbol __ICFEDIT_region_RAM_end_   = 0x20000000;
define symbol __region_RAM2_start_   = 0x20000000;
define symbol __region_RAM2_end_   = 0x200017FF;
define symbol __ICFEDIT_size_cstack_   = 512;
define exported symbol STACK_TEST_BLOCK_SIZE = 0x10;
define exported symbol STACK_TEST_P_4 = __region_RAM2_end_  - 0x3;
define exported symbol STACK_TEST_P_3 = STACK_TEST_P_4 - STACK_TEST_BLOCK_SIZE +0x4;
define exported symbol   BOOT_STACK_ADDRESS = STACK_TEST_P_3 - 0x4;
define exported symbol STACK_TEST_P_2 = __BOOT_STACK_ADDRESS - __ICFEDIT_size_cstack__
-0x4;
define exported symbol STACK_TEST_P_1 = STACK_TEST_P_2 - STACK_TEST_BLOCK_SIZE;
define region RAM_region = mem:[from __ICFEDIT_region_RAM_start__ to
__region_RAM2_end__] - mem:[from STACK_TEST_P_1 size 0x10] - mem:[from STACK_TEST_P_3
size 0x10];

// _____________
// |_____________| --> STACK_TEST_P_1 ....ADR
// |_____________| ....ADR + 0x4
// |_____________| ....ADR + 0x8
// |_____________| --> STACK_TEST_P_2 ....ADR + 0xC
// | |
// | |
// | |
// | STACK |
// | |
// | |
// | |
// | |
// |_____________| -->   BOOT_STACK_ADDRESS
// |_____________| --> STACK_TEST_P_3
// |_____________|
```

```
// |_____________|
// |_____________| --> STACK_TEST_P_4
```

在本例中，大小设置为 0x10。STACK_TEST_P_2 和 STACK_TEST_P_3 符号定义栈下方和上方的第一个地址，它们被定义为导出符号。这意味着它们在应用中也是可见的。这些区域不包含在 RAM 区域中，因此编译器不能为任何变量或其他参数预留此位置。

9.3 栈测试的实现

栈的测试函数和初始化函数在 *iec60730b_4_cm7_stack.S* 文件中，以汇编函数编写。带返回值和函数原型的头文件是 *iec60730b_4_cm7_stack.h*。*iec60730b.h*、*asm_mac_common.h* 和 *iec60730b-types.h* 是本安全库的常用头文件。后续章节显示了链接器设置、初始化流程和实现的示例。

9.3.1 FS_CM4_CM7_STACK_Init

初始化的目的是用给定的模数填充定义的区域。第一件事是将链接器配置文件中的值放入变量中。然后，定义初始化函数所需的其他参数。

初始化示例：

```
#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;
```

函数原型：

void FS_CM4_CM7_STACK_Init(uint32_t stackTestPattern, uint32_t firstAddress, uint32_t secondAddress, uint32_t blockSize);

函数输入：

stackTestPattern - 要写入区域的模数（例如 0x7777777）。

firstAddress - 栈区域下方块的首地址。

secondAddress - 栈区域上方块的首地址。

blockSize - 栈下方和上方区域的大小。

函数输出：

空

函数性能：

对于 0x10 大小的块，该函数需要大约 86 个周期。（1.075µs）[1]

函数大小为 26 B。[1]

调用限制：

无

9.3.2 FS_CM4_CM7_STACK_Test

复位后和运行时的测试流程相同。该函数检查相应区域是否没有用与定义的模数不同的内容重写。测试函数的输入必须与初始化函数的输入相同。

函数原型：

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

函数输入：

stackTestPattern - 测试模数（例如 0x7777777）。

firstAddress - 栈前面的块的首地址。

secondAddress - 栈后面的块的首地址。

blockSize - 块的大小。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_STACK*

函数性能：

对于 0x10 大小的块，该函数需要大约 117 个周期。（1.463µs）[1]

函数大小为 42 B。[1]

调用限制：

无

第 10 章
TSI 测试

触摸感应接口（TSI）基于电容式触摸传感器提供触摸感应检测。外置电容式触摸传感器通常做在 PCB 上，传感器的电极通过器件中的 I/O 引脚连接到 TSI 输入通道。

以下是 KE15z 器件上 I/O 的简化框图：

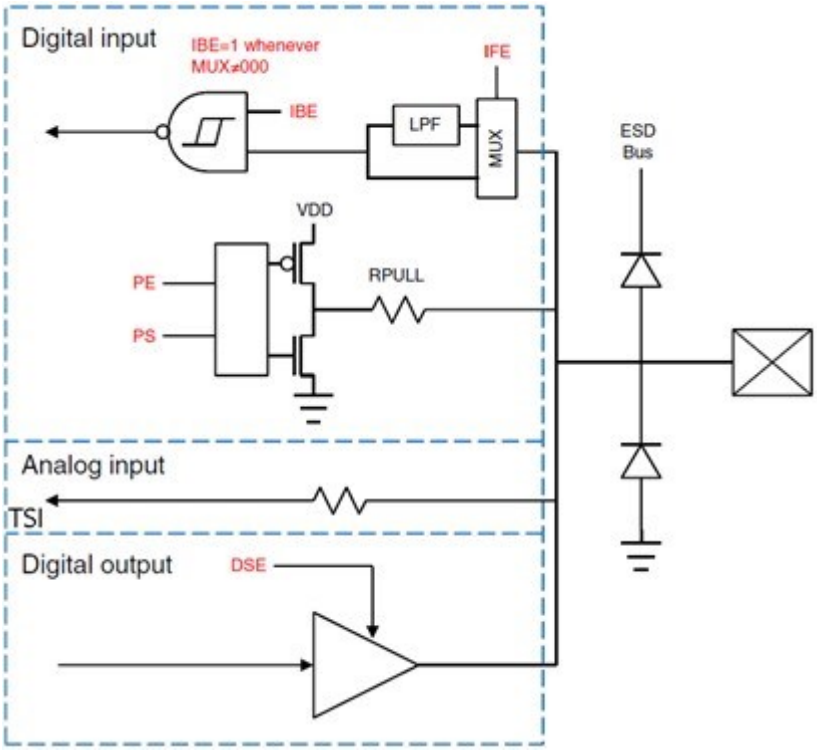

图 31. I/O 简化框图

10.1 TSI 信号短路测试

由于模拟 TSI 通道与数字 I/O 引脚复用，并且模拟或数字功能可以通过软件写入引脚控制寄存器（PCR）中相应的引脚 MUX 控制位来轻松选择或切换，因此测试程序可以在 TSI（模拟）模式和 GPIO（数字）模式之间定期切换引脚 MUX。这意味着切换到 GPIO 模式可用于测试 TSI 信号引线短路。

要测试 TSI 信号短路，可重用以下的 IEC60730 DIO 短路测试（请参见数字输入/输出测试）：

- *FS_DIO_ShortToSupplySet() / FS_DIO_InputExt()* - 测试 TSI 引线对电源 VDD 或 GND 的短路。
- *FS_DIO_ShortToAdjSet() / FS_DIO_InputExt()* - 测试 TSI 引线对相邻引脚或引线的短路。

10.2 TSI 输入测试

该测试负责检查每个 TSI 通道的典型转换结果。当触摸感应电极被释放（未被触摸）时，典型的转换结果由外部连接到 TSI 通道的固有（寄生）电容负载给出。固有电容由 PCB 板的物理属性决定，例如触摸感应电极及其类型、大小、形状和信号线长度等。当电极被触摸时，总的外部电容负载增加，这会改变转换结果。当电极被认为是释放时，将获得电极的典型 TSI 计数器值。

10.2.1 TSI 输入电极断开（开路引脚）测试

TSI 输入测试还涵盖错误（冷）焊接、腐蚀及制造过程中 PCB 组件放置不当导致的问题，例如错误的 SMD 部件值或 SMD 元器件之间的不匹配。

检测方法基于跟踪典型信号（TSI 计数器）的值。所有传感器电极都将其典型的信号基线电平存储在内部闪存中（在一个安全的闪存位置，由 CRC 管理），作为在器件生产期间校准和存储的常数。之后在应用中，实际（测量的）TSI 计数器值与每个传感器的典型值进行比较。如果实际值小于或远高于存储的典型值，则检测到故障。必须适当调整阈值，以避免由于环境漂移和老化而导致故障指示错误。

例如，可以选择两个阈值（高水印和低水印），同时期望信号在正常操作条件下保持在允许范围内，允许范围可以选择为与存储值偏差诸如+/-25%。

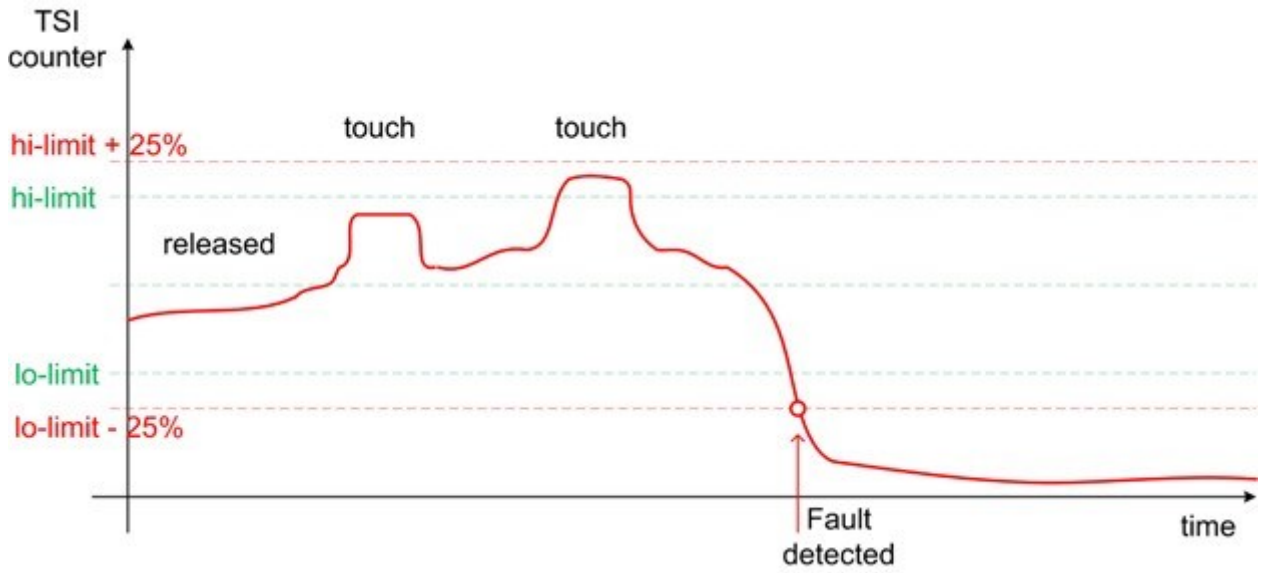

图 32. TSI 输入测试故障的检测

注意：当信号低于低水印线或高于高水印线时，会发生错误。

如果在生产或工厂校准过程中测量到异常信号电平，则意味着 PCB 制造或组装过程（如焊接、元器件放置或机械组装）中可能出现了问题（如弹簧电极短路或弯曲等）。

当电极失去连接或 MCU 引脚与电极之间的信号线终断时，信号会突然下降到正常水平以下。这主要是因为电极冷焊接或串联电阻冷焊接。由于额外的负载，信号可能突然上升到正常水平以上，这可能表明由于长期氧化而出现短路或杂散电导。

10.3 防护传感器或屏蔽电极的短路或断路

防护传感器通常是连接到专用 TSI 通道的隐藏电极，并物理地围绕 PCB 上的其他电极。它通常用于检测触控面板上是否有水，并在发生此问题时禁用其他电极。它可用于软件偏移补偿，提高系统的稳健性和安全性。防护电极信号的路径可以使用上述各种方法进行测试。

屏蔽电极是由专用 TSI 通道主动驱动（缓冲）的铜平面，用以补偿寄生电容，并提高对环境变化（漂移）的灵敏度和抗干扰度。上述类似方法也可用于测试屏蔽电极。

10.4 TSI 输入测试的架构

TSI IO 测试流程执行处理器数字 IO 接口的置信度检查。TSI IO 测试可以在 MCU 复位后和运行时执行一次。

如果发生 TSI IO 错误，通过特定的 FAIL 返回确保安全错误的识别。应用开发人员必须将测试函数的返回值与预期值进行比较。如果等于 FAIL 返回值，则必须跳转到安全错误处理函数。安全错误处理函数可能特定于应用的，不是库的一部分。此函数的主要目的是将应用置于安全状态。

10.4.1 无激励输入的 TSI 输入检查

此 TSI IO 测试基于顺序执行，此时某个级别的外部电容连接到定义的 TSI 输入。测试函数检查转换后的值是否在允许范围内。该测试包括检查 TSI 输入接口，并检查定义的 TSI 输入通道值。

无激励输入的 TSI IO 测试框图如下图所示：

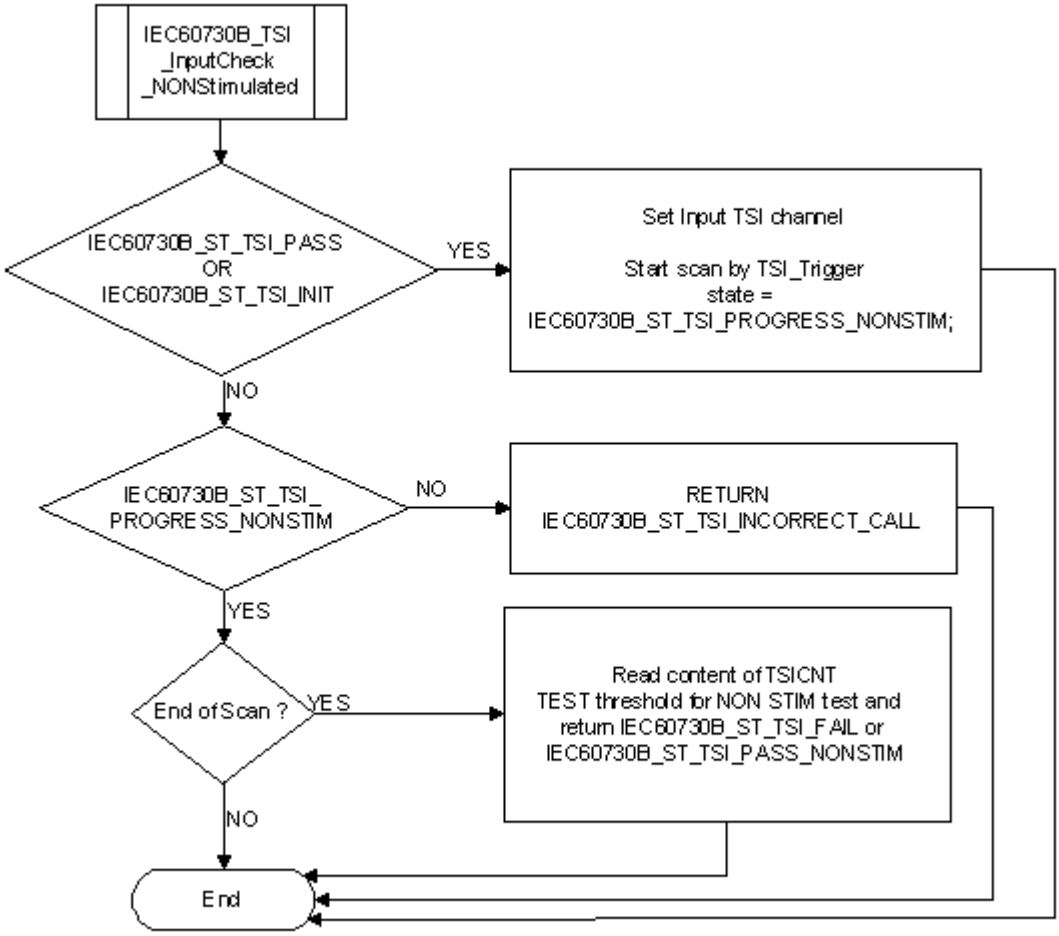

图 33. 无激发输入的 TSI 输入测试软件程序流程

10.4.2 带激励输入的 TSI 输入检查（信号增量检查）

GPIO 上拉/下拉器件可以在每个 TSI 通道引脚上启用，同时 TSI 通道被主动扫描，以通过上下拉电阻引起的额外负载影响模拟转换结果。这可用于激励引脚。该通道激励用于通过软件来模拟在期望的通道引脚上的 TSI 信号（计数器值）的变化，而无需外部触摸事件。当 TSI 测量激活时，通过启用相应 DIO 引脚上的内部下拉或上拉电阻，可以将负载添加到充电信号中，从而改变 TSI 计数器的累计数量（信号增量）。用此方法，可以检查从 TSI 输入引脚到 TSI 转换计数器的整个测量链，包括内部模拟多路复用器。您可以激励每个 TSI 通道输入，检查每个转换结果，并将它们与激励状态下的有效典型信号增量值进行比较。禁用上下拉器件时，TSI 计数器值必须返回空闲状态的典型有效电平。

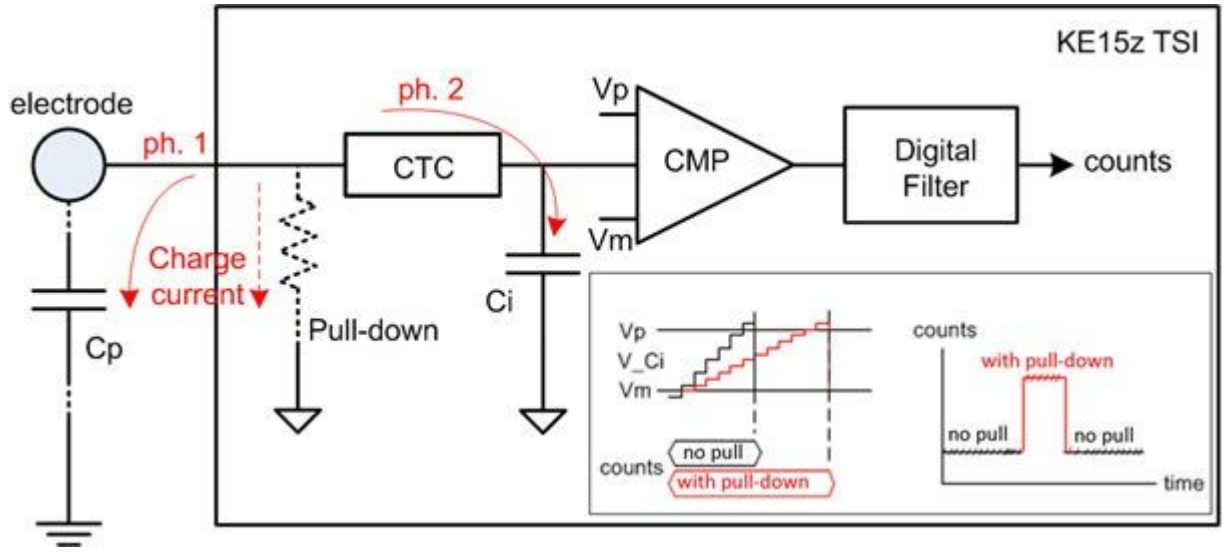

图 34. TSI 通道激发原理

10.4.2.1 TSI 输入通道的激励

在正常状态下，在每个外部充电周期（ph.1）期间，充电电流完全用于将 Cp 充电到一定水平。当下拉电阻被启用时，它为充电电流创建了额外的信号路径，其中一部分电流通过电阻泄漏到 GND。与下拉电阻被禁用的正常状态相比，Cp 会被充电到更低的电平（即由 Cp 累积的电荷更小）。

在内部充电周期（ph.2），Cp 累积的电荷被转到参考内部电容 Ci。启用内部上拉电阻时，充电步长较小。需要更大的充电步长来将 Ci 充电到适当的水平。充电步长越大，TSI 结果计数器中累积的时间越长，计数值越大。

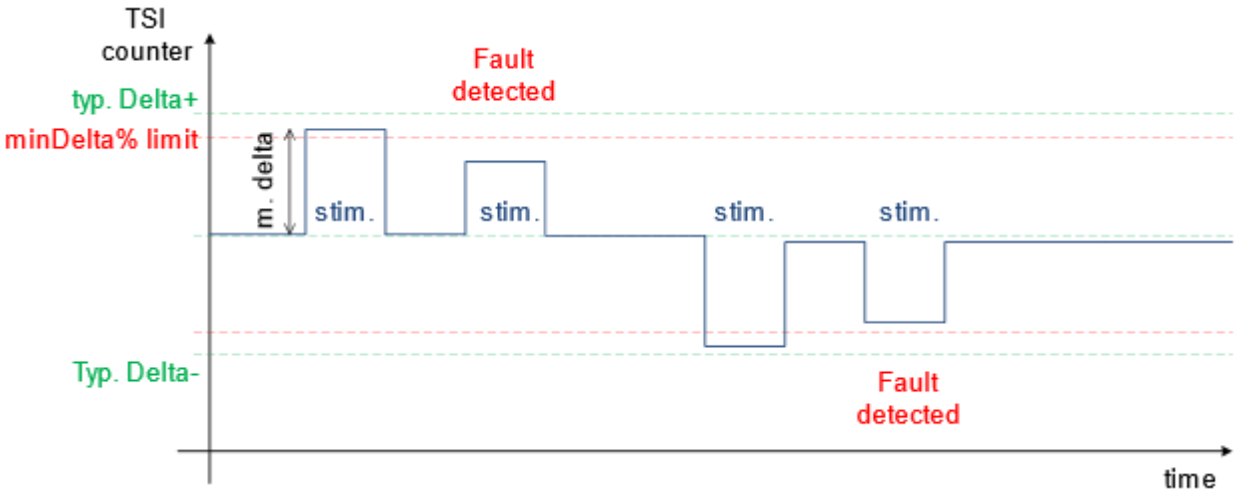

图 35. TSI 激励通道增量检查

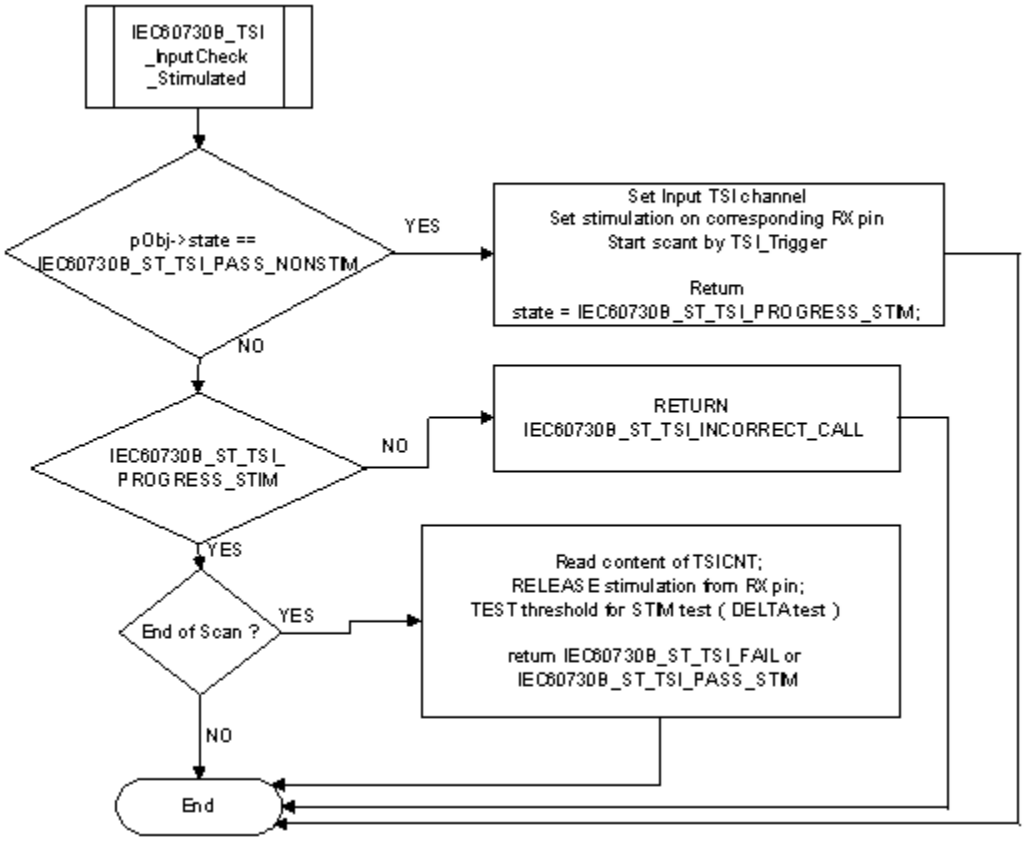

图 36. 具有激励输入的 TSI 输入测试软件程序流程

10.5 TSI 测试的实现

TSI IO 测试的测试函数在 *iec60730b_tsi .c* 文件中，以 C 函数编写。带函数原型的头文件是 *iec60730b_tsi .h*。*Iec60730b.h* 和 *iec60730b_types.h* 是本安全库的常用头文件。

调用以下函数来测试 TSI 输入：

- *FS_TSI_InputInit()*
- *FS_TSI_InputCheckNONStimulated()*
- *FS_TSI_InputCheckStimulated()*
- *FS_TSI_InputStimulate()*
- *FS_TSI_InputRelease()*

10.5.1 TSI 输入测试的原理

TSI 输入测试的原理是检查转换后的模拟值是否具有预期值。该测试使用已知转换值的 TSI 输入，并检查转换后的值是否符合定义的限制。通常应为所需参考值的+/-25%左右。

该测试由第一次调用 *FS_TSI_InputCheckNONStimulated ()* 函数触发。测试分为三个部分（初始化、测试执行和测试结束）。该测试还收集正常（非激励）状态下的 TSI 计数器数据，作为 TSI 激励输入测试的参考数据。

有关测试的详细信息，请参阅 TSI 输入测试。

10.5.2 TSI 激励输入测试的原理

该测试负责定期检查由内部上拉激励的输入信号的 TSI 计数器增量变化。测试由 *FS_TSI_InputCheckStimulated()* 函数调用触发。当通道测量完成时，电流输入所对应的上下拉电阻被禁用。以激励输入测量的 TSI 计数器值与之前没有激励时收集的值进行比较。这个差值称为 TSI 增量信号。当增量信号不为零时，TSI 输入通道工作正常。这意味着当输入受到激励时，测量到显著的计数器变化。根据 TSI 感测模式和激励的极性，增量值可能有正或负的符号。然后将该增量值与配置文件中通过实验测量和预定义的典型增量值进行比较。这意味着在校准已知的良好器件的期间，必须提前测量典型的增量值。有关测试的详细信息，请参见 TSI 输入测试。

注意

该测试要求非激励输入测试优先于激励输入测试。对于当前的 TSI 输入通道，必须依次调用 *FS_TSI_InputCheckNONStimulated()* 和 *FS_TSI_InputCheckStimulated()* 函数。如果调用顺序无效，函数返回 FS_TSI_INCORRECT_CALL 失败代码。

10.5.3 TSI 测试输入函数调用示例

```
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()

此函数初始化所定义的“fs_tsi_t”结构中的各个项，并将状态设置为“FS_TSI_INIT”。应在非激励输入测试之前调用。

函数原型：

*void FS_TSI_InputInit(fs_tsi_t *pObj);*

函数输入：

**pObj* - 此输入参数是指向 TSI 测试实例的指针。

函数输出：

空

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

10.5.5 FS_TSI_InputCheckNONStimulated()

该函数执行不带激励输入的 TSI 测试顺序的第一部分，读取 TSI 计数器值并检查该值是否符合预定义的限制。它还收集正常（非激励）状态的 TSI 计数器数据，以进行进一步的激励输入测试。

当函数报告 FS_TSI_PASS_NONSTIM 或 FS_FAIL_TSI 时，测试结束。

函数原型：

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

函数输入：

**pObj* - 此输入参数是指向 TSI 测试实例的指针。

pTsi - 此输入参数是 TSI 模块的地址。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_TSI_PASS_NONSTIM*
- *FS_TSI_INCORRECT_CALL*
- *FS_FAIL_TSI*

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

10.5.6 FS_TSI_InputCheckStimulated()

该函数执行带激励输入的 TSI 测试顺序的第二部分，检查 TSI 输入激励计数器增量是否在预期范围内。只有通过非激励测试后才能调用此测试函数，否则返回 FS_TSI_INCORRECT_CALL。

注意

通常，*FS_TSI_InputCheckNONStimulated()* 调用先于 *FS_TSI_InputCheckStimulated()* 调用。建议按封闭的顺序调用这两个测试函数。

如果函数上报 FS_TSI_PASS_STIM 或 FS_FAIL_TSI，测试结束。

函数原型：

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

函数输入：

**pObj* - 此输入参数是指向 TSI 测试实例的指针。

pTsi - 此输入参数是 TSI 模块的地址。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_TSI_PASS_STIM*
- *FS_TSI_INCORRECT_CALL*
- *FS_FAIL_TSI*

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

10.5.7 FS_TSI_InputStimulate()

需要 TSI 输入激励时，该函数通过电流 TSI 通道上的上下拉电阻激励相应的 TSI 引脚。上拉/下拉的极性由 fs_tsi_t 结构中的 stim_polarity 参数给出。

函数原型：

*FS_RESULT FS_TSI_InputStimulate(fs_tsi_t *pObj);*

函数输入：

**pObj* - 此输入参数是指向 TSI 测试实例的指针。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_TSI*

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

10.5.8 FS_TSI_InputRelease()

该函数在相应的 TSI 通道上禁用上下拉电阻激励。一旦激励输入检查完成，它也会被 *FS_TSI_InputStimulate()* 函数在内部调用。

函数原型：

*FS_RESULT FS_TSI_InputRelease(fs_tsi_t *pObj);*

函数输入：

**pObj* - 此输入参数是指向 TSI 测试实例的指针。

函数输出：

typedef uint32_t FS_RESULT;

- *FS_PASS*
- *FS_FAIL_TSI*

函数性能：

有关函数性能的信息，请参见内核自检库–源代码版本。

第 11 章
看门狗测试

看门狗测试用于测试看门狗定时器的功能。该测试检查看门狗定时器是否会导致复位，以及复位是否在预期时间发生。开始测试之前，必须配置看门狗，以便在相应的应用中使用。测试前的下一步是设置独立的器件定时器，用于看门狗超时比较。之后调用看门狗测试的第一个函数。此函数刷新看门狗定时器，激活器件定时器，并在无限循环期间捕获器件定时器计数器的值。此函数只能在上电复位（POR）后调用一次。看门狗复位后，必须调用第二个函数。除 POR 外，应在每次复位后调用此函数。此函数检查捕获的器件定时器计数器值是否与预期的看门狗超时值相对应。下一步检查看门狗复位的次数是否超过限制值。可以选择在出现错误结果后必须采取的操作。出于安全要求，为看门狗和器件定时器选择时钟源的选项有限。第一个条件是看门狗定时器时钟不能与看门狗总线接口时钟相同。查看器件参考手册，了解看门狗定时器时钟源选项。第二个条件是看门狗定时器时钟不能与器件定时器时钟相同。

11.1 符合 IEC/UL 标准的看门狗测试

IEC60730-附录 H 表中未直接规定看门狗测试，但它部分满足 IEC 60730-1、IEC 60335、UL 60730 和 UL 1998 标准的安全要求，如表 29 所述。

表 29. 符合标准的看门狗测试

测试	元器件	故障/错误	软件/硬件 类别	可接受的测量
看门狗测试	3. 时钟	频率错误	B/R.1	频率监测
看门狗测试	8. 监测器件和比较器	不符合静态和动态功能规范的任何输出	B/R.1	被测试监测

11.2 看门狗测试的实现

看门狗的测试函数在 *iec60730b_wdog.c* 文件中。头文件为 *iec60730b_wdog.h*。*iec60730b.h*、*iec60730b.h* 和 *iec60730b-types.h* 是本安全库的常用头文件。

必须有可用的空间，该空间在 RAM 存储区，在非 POR 复位后不会损坏。

这个存储区用于 *fs_wdog_test_t* 结构类型的变量，包含三个成员，在 *iec60730b _wdog.h* 文件中定义。

进行看门狗测试前，需要对看门狗模块和器件定时器进行配置。

不同器件支持的看门狗定时器模块不一样。有关相应器件的正确函数，请参见器件实现章节。

要确保此函数的处理。要确定复位源，请使用复位控制模块。常见的配置是，如果 check 函数发现了不想要的结果，则程序将在函数中保持无限循环。这将导致应用停留在看门狗复位的循环中。如果输入零作为 check 函数的第四个输入值，则不会激活无限循环。在这种情况下，请确保应用处于安全状态。

下面是看门狗测试实现（MKV1x）的示例：

```
#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()

此函数用于清零复位计数器，该计数器是 *fs_wdog_test_t* 结构的成员。它刷新看门狗，从零开始计数。它启动 LPTMR，必须在调用函数之前对其进行配置。在等待的无限循环中，LPTMR 的值被周期性地存储在 RAM 的预留区域中。

函数原型：

*void FS_WDOG_Setup_LPTMR(fs_wdog_test_t *pWatchdogBackup, uint8_t refresh_index)*

函数输入：

**pWatchdogBackup* - 指向具有 *fs_wdog_test_t* 变量的结构的指针。

refresh_index - 用于选择 WDOG 刷新顺序的索引。使用以下宏：FS_KINETIS_WDOG、FS_WDOG32 或 FS_COP_WDOG。

函数输出：

空

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

看门狗定时器和 LPTMR 必须正确配置。必须声明 *fs_wdog_test_t* 类型的变量并将其放在可靠的位置。中断应该被禁用。

如果示例应用被设置为正确的版本，则必须正确填写“refresh_index”参数。对于其他器件，请将器件的参考手册与表 30 或下表中的参考器件进行比较。

表 30. 刷新顺序

刷新索引参数	刷新顺序	参考器件
FS_KINETIS_WDOG	• WdogBase->REFRESH = 0xA602U; • WdogBase->REFRESH = 0xB480U;/* refresh sequence */	MKV11
FS_WDOG32	WdogBase->CNT = 0xB480A602U; /* refresh sequence */	MK32L2A
FS_COP_WDOG	• WdogBase->SRVCOP = FS_SIM_KL2X_SRVCOP_SRVCOP(0x55U); • WdogBase->SRVCOP = FS_SIM_KL2X_SRVCOP_SRVCOP(0xAAU);	MKL26z

11.2.2 FS_WDOG_Setup_KE0XZ()

此函数可用于 KE0xZ 器件。此函数清零复位计数器，该计数器是 *fs_wdog_test_t* 结构的成员。它刷新看门狗，从零开始计数。它启动 RTC，必须在调用函数之前进行配置。在等待的无限循环中，RTC 的值被周期性地存储在 RAM 的预留区域中。

函数原型：

*void FS_WDOG_Setup_KE0XZ(fs_wdog_test_t *pWatchdogBackup);*

函数输入：

**pWatchdogBackup* - 指向具有 *fs_wdog_test_t* 变量的结构的指针。

函数输出：

空

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

应禁用中断。看门狗定时器和 RTC 必须正确配置。必须声明 *fs_wdog_test_t* 类型的一个变量，并将其放入在应用启动期间不会被覆盖的 RAM 区域。

在调用 WDOG 测试之前，必须填写以下变量：

fs_wdog_test_t * wdogBackup

- *wdogBackup->pResetDetectRegister* - “ResetDestect” 寄存器的地址。

- *wdogBackup->ResetDetectMask* - WDOG 复位源的掩码（在复位检测寄存器中）。
- *wdogBackup->RefTimerBase* - 所使用的 RTC 定时器的基址。
- *wdogBackup->WdogBase* - 所使用的 WDOG 的基址。

11.2.3 FS_WDOG_Setup_IMX_GPT()

此函数可用于 MIMXRT10xx 和 MIMX8Mini 器件。此函数清零复位计数器，该计数器是 *fs_wdog_test_t* 结构的成员。它刷新看门狗，从零开始计数。它启动 GPT，必须在调用函数之前进行配置。在等待的无限循环中，GPT 的值被周期性地存储在 RAM 的预留区域中。

函数原型：

*void FS_WDOG_Setup_IMX_GPT(fs_wdog_test_t *pWatchdogBackup, uint8_t refresh_index)*

函数输入：

**pWatchdogBackup* - 指向具有 *fs_wdog_test_t* 变量的结构的指针。

**pGPT* - 指向 GPT 基址的指针。

refresh_index - 选择正确的 WDOG 刷新顺序 - FS_IMXRT 或 FS_IMX8M。

函数输出：

空

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

看门狗定时器和 GPT 必须正确配置。必须声明 *fs_wdog_test_t* 类型的变量并将其放在可靠的位置。中断应该被禁用。

“refresh_index”参数必须根据示例应用版本正确填写。对于其他器件，请将器件的参考手册与参考器件进行比较。见表 31。

表 31. 刷新序列

刷新索引参数	刷新顺序	参考器件
FS_IMXRT	WdogBase->CNT = 0xB480A602U;	MIMXRT10xx
FS_IMX8M	• WdogBase->WSR = 0x5555; • WdogBase->WSR = 0xAAAA;	IMX8M

11.2.4 FS_WDOG_Check()

该函数将参考计数器的捕获值与预先计算的限制值进行比较，并检查看门狗复位计数器是否溢出。如果该函数在非看门狗复位后调用，则设置“wd_test_uncomplete_flag”并返回相应的返回错误。使用“endless_loop_enable”参数，函数中的无限循环被启用或禁用（通过将其设置为 1 或 0）。如果无限循环被禁用，在以下情况下，函数将返回相应的错误：

- 在非看门狗或非 POR 复位后进入 - FS_FAIL_WDOG_WRONG_RESET。
- 看门狗测试的计数器不符合限制值 - FS_FAIL_WDOG_VALUE。
- 看门狗复位超过定义的限值 - FS_FAIL_WDOG_OVER_RESET。

函数原型：

*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)*

函数输入：

limitHigh - 参考计数器的预计算限值。

limitLow - 参考计数器的预计算限值。

limitResets - 看门狗复位的限值。

endlessLoopEnable - 在函数中启用或禁用无限循环。

**pWatchdogBackup* - 指向 *fs_wdog_test_t* 变量结构的指针。

clear_flag - 布尔值。如果为 TRUE，则删除复位检测寄存器中的 WDOG 复位标志。

RegWide8b - 当为 TRUE 时，复位检测寄存器存取为 8b（否则为 32b）。

函数输出：

如果“endlessLoopEnable”参数的设置为 1，或者返回值为 FS_FAIL_WDOG_WRONG_RESET、FS_FAIL_WDOG_VALUE、FS_FAIL_WDOG_OVER_RESET 或 FS_PASS，则函数可以保持在无限循环中。

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

必须先执行相应的设置函数。

11.2.5 FS_WDOG_Setup_WWDT_LPC_mrt()

此函数可用于具有 WWDT 和 MRT 的 LPC 器件。此函数清零复位计数器，该计数器是 *fs_wdog_test_t* 结构的成员。它刷新看门狗，从零开始计数。它启动 MRT，且必须在调用函数之前进行配置。在等待的无限循环中，MRT 的值被周期性地存储在 RAM 的预留区域中。

函数原型：

*void FS_WDOG_Setup_WWDT_LPC_mrt(fs_wdog_test_t *pWatchdogBackup, uint8_t channel);*

函数输入：

**pWatchdogBackup* - 指向 *fs_wdog_test_t* 变量结构的指针。

channel - MRT 定时器的通道索引。

函数输出：

空

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

看门狗定时器和 MRT 必须正确配置。必须声明一个 *fs_wdog_test_t* 类型的变量，并将其放入在应用启动期间不会被覆盖的 RAM 区域。中断应该被禁用。

在调用 WDOG 测试之前，必须填写以下变量：

fs_wdog_test_t * wdogBackup

- *wdogBackup->pResetDetectRegister* -“ResetDestect”寄存器的地址。
- *wdogBackup->ResetDetectMask* - WDOG 复位源的掩码（在复位检测寄存器中）。
- *wdogBackup->RefTimerBase* - 所使用的 MRT 定时器的基址。

- *wdogBackup->WdogBase* - 所使用的 WDOG 的基址。

11.2.6 FS_WDOG_Setup_WWDT_LPC()

此函数可用于带 WWDT 的 LPC 器件。此函数清零复位计数器，该计数器是 *fs_wdog_test_t* 结构的成员。它刷新看门狗，从零开始计数。它启动 CTimer，且必须在调用函数之前进行配置。在等待的无限循环中，CTimer 的值被周期性地存储在 RAM 的预留区域中。

函数原型：

*void FS_WDOG_Setup_WWDT_LPC(fs_wdog_test_t *pWatchdogBackup);*

函数输入：

**pWatchdogBackup* - 指向 *fs_wdog_test_t* 变量结构的指针。

函数输出：

空

函数性能：

此函数的允许时间取决于 WDOG 超时时间，因为该函数在 WDOG 复位中等待。函数的大小为 70 字节。

调用限制：

看门狗定时器和 Ctimer 必须正确配置。必须声明一个 *fs_wdog_test_t* 类型的变量，并将其放入在应用启动期间不会被覆盖的 RAM 区域。中断应该被禁用。

在调用 WDOG 测试之前，必须填写以下变量：

fs_wdog_test_t * wdogBackup

- *wdogBackup->pResetDetectRegister* -“ResetDestect”寄存器的地址。
- *wdogBackup->ResetDetectMask* - WDOG 复位源的掩码（在复位检测寄存器中）。
- *wdogBackup->RefTimerBase* - 所使用的 CTIMER 定时器的基址。
- *wdogBackup->WdogBase* - 所使用的 WDOG 的基址。

11.2.7 FS_WDOG_Check_WWDT_LPC()

此函数可用于带 WWDT 看门狗的器件。该函数将目标计数器的捕获值与预先计算的限制值进行比较，并检查看门狗复位计数器是否溢出。如果该函数在非看门狗复位后调用，则设置“wd_test_uncomplete_flag”。使用“endless_loop_enable”参数启用或禁用函数中的无限循环（通过将其设置为 1 或 0）。如果无限循环被禁用，在以下情况下，函数将返回相应的错误：

- 在非看门狗或非 POR 复位后进入 - FS_FAIL_WDOG_WRONG_RESET。
- 看门狗测试的计数器不符合限制值 - FS_FAIL_WDOG_VALUE。
- 看门狗复位超过定义的限值 - FS_FAIL_WDOG_OVER_RESET。

函数原型：

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

函数输入：

limitHigh - 参考计数器的预计算限值。

limitLow - 参考计数器的预计算限值。

limitResets - 看门狗复位的限值。

endlessLoopEnable - 在函数中启用或禁用无限循环。

**pWatchdogBackup* - 指向 *fs_wdog_test_t* 变量结构的指针。

函数输出：

如果“endlessLoopEnable”参数设置为 1 或返回值 FS_FAIL_WOG_WRONG_RESET、FS_FAIL_WOG_VALUE、FS_FFAIL_WOG_OVER_RESET 或 FS_PASS，则函数可以保持在无限循环中。

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

必须先执行相应的设置函数。

如果需要，在调用 WDOG 测试之前，必须填写以下变量：

fs_wdog_test_t * wdogBackup

- *wdogBackup->pResetDetectRegister* -“ResetDestect”寄存器的地址。
- *wdogBackup->ResetDetectMask* - WDOG 复位源的掩码（在复位检测寄存器中）。
- *wdogBackup->RefTimerBase* - 所使用的定时器的基址。
- *wdogBackup->WdogBase* - 所使用的 WDOG 的基址。

11.2.8 FS_WDOG_Check_WWDT_LPC55SXX()

此函数可用于 LPC55Sxx 器件。该函数将目标计数器的捕获值与预先计算的限制值进行比较，并检查看门狗复位计数器是否溢出。如果该函数在非看门狗复位后调用，则设置“wd_test_uncomplete_flag”。使用“endless_loop_enable”参数启用或禁用函数中的无限循环（通过将其设置为 1 或 0）。如果无限循环被禁用，在以下情况下，函数将返回相应的错误：

- 在非看门狗或非 POR 复位后进入 - FS_FAIL_WDOG_WRONG_RESET。
- 看门狗测试的计数器不符合限制值 - FS_FAIL_WDOG_VALUE。
- 看门狗复位超过定义的限值 - FS_FAIL_WDOG_OVER_RESET。

函数原型：

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

函数输入：

limitHigh - 参考计数器的预计算限值。

limitLow - 参考计数器的预计算限值。

limitResets - 看门狗复位的限值。

endlessLoopEnable - 在函数中启用或禁用无限循环。

**pWatchdogBackup* - 指向 *fs_wdog_test_t* 变量结构的指针。

函数输出：

如果“endlessLoopEnable”参数设置为 1 或返回值 *FS_FAIL_WOG_WRONG_RESET*、*FS_FAIL_WOG_VALUE*、*FS_FFAIL_WOG_OVER_RESET* 或 *FS_PASS*，则函数可以保持在无限循环中。

函数性能：

有关函数性能的信息，请参阅内核自检库–源代码版本。

调用限制：

必须先执行相应的设置函数。

在调用 WDOG 测试之前，必须填写以下变量：

fs_wdog_test_t * wdogBackup

- *wdogBackup->pResetDetectRegister* - “ResetDestect” 寄存器的地址。
- *wdogBackup->ResetDetectMask* - WDOG 复位源的掩码（在复位检测寄存器中）。
- *wdogBackup->RefTimerBase* - 所使用的定时器的基址。
- *wdogBackup->WdogBase* - 所使用的 WDOG 的基址。

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

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

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

Date of release: 30 September 2021

Document identifier: IEC60730BCM4CM7L42UG

arm