NXP Semiconductors Document Number: AN0000000000001

Application Notes. Rev. n, 01/2020

S32K1xx Safety Cookbook

1. Introduction

The purpose of this document is to show how the assumptions of the S32K1xx series Safety Manual could be implemented. In the Safety Manual, assumptions for different software requirements are described and the implementation hints and rationales are given.
To show how those assumptions could be covered with software, a set of examples was created. In this document pieces of those codes are shown in tables with the corresponding assumption and implementation. This was done to make the Safety Manual easier to understand and to implement.

It is assumed that the reader has already read the S32K1xx series Safety Manual.

Contents

2. Software requirements

The codes developed in this application note are intended to exemplify how the assumptions from the Safety Manual could be implemented. The codes were classified in a way that one example covers many assumptions.

Code organization

Family	Project name	Modules	Assumptions
S32K11x	**S32K116_Safety_Clock**	CLK	80, 83, 84, 100, 140
	S32K116_Safety_Configuration	RCM, SCG	140
	S32K116_Safety_eDMA	eDMA, CRC	101, 104
	S32K116_Safety_FTM_Double_PWM	FTM	133, 137
	S32K116_Safety_PMC_LVD	ADC, CRC, PMC	70, 84, 85, 100, 204
	S32K116_Safety_RAM	ERM, LPIT	48, 98, 99, 100, 113, 140
	S32K116_Safety_Stack	DWT	100, 139
	S32K116_Safety_VLPS_WDOG_LPIT	CLK, LPIT, VLPS, WDOG	48, 67, 69, 82, 100, 107, 140, 202
	S32K116_Safety_MPU	MPU	94,95
S32K14x	**S32K142_Safety_Clock**	CLK	78, 80, 81, 84, 100, 213
	S32K142_Safety_Configuration	RCM, SCG	140
	S32K142_Safety_eDMA	eDMA, CRC	101, 104
	S32K142_Safety_FTM_Double_PWM	FTM	133, 137
	S32K142_Safety_PMC_LVD	ADC, CRC, PMC	70, 84, 85, 100, 204
	S32K142_Safety_RAM	ERM, LPIT	48, 99, 100, 113
	S32K142_Safety_Stack	DWT	100, 139
	S32K142_Safety_VLPS_WDOG_LPIT	CLK, LPIT, VLPS, WDOG	48, 67, 69, 82, 100, 107, 202
	S32K142_Safety_MPU	MPU	94, 95

Table structure

The next table shows how the tables in document are divided. Each assumption is explained as in the S32K1xx Safety Manual, a piece of code which covers the assumption is shown and an explanation of the code is given.

Assumptions		
SM_XXX (Assumption number as written in the S32K1XX Safety Manual)	**Description**	**Code**
	(Assumption description as written in the S32K1XX Safety Manual)	In red it is explained if the piece of code applies for the entire S32K1 family or if it only applies to a subset of the family. S32K1xx - Entire family S32K11x - S32K116 and S32K118 S32K14x - S32K142, S32K144, S32K146 and S32K148 (Piece of code which covers the assumption, the rest can be found in the projects attached)
	Implementation	
	(Explanation of the code which covers the assumption)	

Summary of implemented assumptions

Assumptions					
Category	**Number**	**Implemented**	**Category**	**Number**	**Implemented**
Power	SM_084	Yes	**MPU**	SM_094	Yes
	SM_204	Yes		SM_095	Yes
	SM_085	Yes	**NVIC**	SM_098	Yes
Clocks	SM_078	Yes		SM_099	Yes
	SM_213	Yes		SM_100	Yes
	SM_083	Yes	**eDMA**	SM_101	Yes
	SM_080	Yes		SM_102	Yes
	SM_081	Yes		SM_104	Yes
Flash	SM_114	Yes	**WDOG**	SM_067	Yes
	SM_116	In progress		SM_202	Yes
	SM_117	Yes		SM_069	Yes
	SM_119	In progress	**LPIT**	SM_107	Yes
	SM_118	In progress	**Low power**	SM_082	Yes
RAM	SM_113	Yes	**CRC**	SM_070	Yes
Debug mode	SM_047	NA	**Communications**	SM_051	NA
	SM_048	Yes	**I/O functions**	SM_133	Yes
Stack	SM_139	Yes		SM_137	Yes
S32K1XX config	SM_140	Yes	**ADC**	SM_130	Yes
AWIC/External NMI	SM_126	Yes			

NA: This assumptions do not need code to be covered.

Power

The S32K1xx family uses the PMC module that manages the supply voltages for all modules on the device. This unit includes the internal regulator for the logic power supply and a set of voltage monitors for low voltage detector (LVD) and low voltage reset (LVR).

2.4.1. Power Management Controller (PMC)

The Power Management Controller has monitors to detect when the voltage goes below the Low Voltage Detect and Low Voltage Reset thresholds. If the voltage goes below the LVR a reset is triggered to prevent a potential failure. If the voltage goes below the LVD a warning, an interrupt or a reset can be triggered.

The table below summarizes how to implement the assumptions related to the Power section of the Safety Manual in section 5.2.

Assumptions

SM_084

Description

The application software must check the status registers of the RCM for error flags.

Implementation

After a reset a sanity check is performed. The RCM_SRS register is checked to determine if a low voltage detect (LVD) triggered the past reset.

Code

```
S32K1xx_ADC_CRC_PMC -> main.c

LVD_status_t Sanity_check(void)
{
        /* Check if the LVD bit on the RCM_SRS register
is set */
        if(True == ((RCM->SRS & RCM_SSRS_SLVD_MASK) >>
RCM_SSRS_SLVD_SHIFT))
        {
                /* Return the LVD_NOT_OK status */
                return LVD_NOT_OK;
        }

        /* By default return the LVD_OK status*/
        return LVD_OK;
}
```

SM_204

Description

It is assumed that the ADCs are used to monitor the bandgap reference voltage of the PMC and to monitor the internal supplies connected to ADCs.

Implementation

The ADC0 internal channel 0 is connected to the internal supply monitoring and used to measure the bandgap. This bandgap voltage should be the same value as the one stated in the S32K1xx Data Sheet.

Code

```
S32K1xx_ADC_CRC_PMC -> main.c

/* Get the PMC reference voltage */
PMC_reference_voltage = ADC_Get_Internal_Supply();

S32K1xx_ADC_CRC_PMC -> ADC.c

uint16_t ADC_Get_Internal_Supply(void)
{
        uint16_t Bandgap_reference_voltage = 0;

        /* Wait for the conversion to be ready */
        while (False == ((ADC0->SC1[0] &
ADC_SC1_COCO_MASK) >> ADC_SC1_COCO_SHIFT));

        /* Read the conversion */
        Bandgap_reference_voltage = ADC0->R[0];

        return Bandgap_reference_voltage;
}
```

SM_085

Description

Software must not disable the direct transition by the RCM into a safe state due to an overvoltage or undervoltage indication.

Implementation

The sanity check jumps to a safe sate if the past reset was triggered by LVD. The safe state is application specific.

Code

```
S32K1xx_ADC_CRC_PMC -> main.c

/* If the status of the Sanity_check is LVD_NOT_OK then
execute the safe state */
LVD_status = Sanity_check();
if(LVD_NOT_OK == LVD_status)
{
        /* Execute the safe state */
        Safe_state();
}

LVD_status_t Sanity_check(void)
{
        /* Check if the LVD bit on the RCM_SRS register
is set */
        if(True == ((RCM->SRS & RCM_SSRS_SLVD_MASK) >>
RCM_SSRS_SLVD_SHIFT))
        {
                /* Return the LVD_NOT_OK status */
                return LVD_NOT_OK;
        }

        /* By default return the LVD_OK status*/
```

		```
		        return LVD_OK;
		}
		```

Clock

The S32K1xx family uses the System Clock Generator (SCG) module to generate most of the clocks used by the device. The SCG module controls which clock source (internal references, external crystals, external clocks) is used to derive system clocks. The SCG also divides the selected clock source into a variety of clock domains, including clocks for system bus masters, system bus slaves, and flash memory.

2.5.1. Clock

For safety applications it is important to have a high quality and reliable clock source. For the S32K14x the System Phase-Locked Loop (SPLL) should be used, it has a monitor to detect if there is a loss of clock. For the S32K11x the FIRC with the CMU should be used. The CMU can detect if there is a loss of FIRC and if the FIRC frequency is between a certain range.

The table below summarizes how to implement the assumptions related to the Clocks section of the Safety Manual in section 5.3.

Assumptions		
	Description	**Code**

SM_078	Before executing any safety function, a high quality clock (low noise, low likelihood for glitches) based on an external clock source shall be configured as the system clock of the S32K14x.	`S32K14x_CLK -> main.c`
		
		`/* Initialize the SPLL and its monitor */`
		`SPLL_init();`
		
		`S32K14x_CLK -> clocks.c`
		
		`/* Disable the SPLL so changes can be made */`
		`SCG->SPLLCSR &= ~SCG_SPLLCSR_SPLLEN_MASK;`
		
		`/* Divide by 1 the SPLL in the DIV 1 and 2 */`
		`SCG->SPLLDIV |= SCG_SPLLDIV_SPLLDIV1(1) |`
		`SCG_SPLLDIV_SPLLDIV2(1);`
		
		`/* Set the multiply factor to 20 and the divide factor to 2 */`
		`SCG->SPLLCFG |= SCG_SPLLCFG_MULT(4) |`
		`SCG_SPLLCFG_PREDIV(1);`
		
		`/* Enable the SPLL */`
		`SCG->SPLLCSR |= SCG_SPLLCSR_SPLLSEL_MASK |`
		`SCG_SPLLCSR_SPLLEN_MASK;`
		
		`/* Wait until the SPLL is considered valid */`
		`while(False == ((SCG->SPLLCSR &`
		`SCG_SPLLCSR_SPLLVLD_MASK) >>`
		`SCG_SPLLCSR_SPLLVLD_SHIFT));`

	Implementation	
	The PLL should be used for functional safety applications. It is configured at 80 MHz by receiving a frequency of 8MHz from the external oscillator and by multiplying it 10 times.	
	Description	**Code**

SM_213	A check should be implemented to verify that with an intended PLL configuration the PLL locks with the correct output clock.	`S32K14x_CLK -> clocks.c`
		
		`/* Map the CLKOUT to the PTB5 */`
		`/* Enable the clock to the port B */`
		`PCC->PCCn[PCC_PORTB_INDEX] = PCC_PCCn_CGC_MASK;`
		`/* Set the Port B pin 5 as output */`
		`PTB->PDDR |= 1 << PTB5;`
		`/* Select the CLKOUT option -> ALT5*/`

	Implementation	

	The PLL is routed to the PTB5 GPIO in order to verify if the PLL frequency is 80 MHz.	PORTB->PCR[PTB5] = PORT_PCR_MUX(5);
SM_083	**Description**	**Code**

	The following supervisor functions are required:
	• Loss of fast internal reference clock
	• System FIRC frequency higher than the (programmable) upper frequency reference
	• System FIRC frequency lower than the (programmable) lower frequency reference
		S32K11x_CLK -> clocks.c
		
		CMU0 configuration
		void CMU0_init(void)
		{
		CMU_FC_0->RCCR = 7;
		
		/* Enable the CMU0 interrupt */
		CMU_FC_0->IER |= CMU_FC_IER_FHHAEE_MASK | CMU_FC_IER_FLLAEE_MASK;
		
		/* Enable the frequency check */
		CMU_FC_0->GCR = CMU_FC_GCR_FCE_MASK;
		
		/* Wait until the frequency check starts running */
		while(False == (CMU_FC_0->SR & CMU_FC_SR_RS_MASK) >> CMU_FC_SR_RS_SHIFT);
		}

	Implementation	

	The two modules of the CMU are enabled. The CMU0 is used to determine if there is a loss of FIRC.
	The CMU1 is used to determine if the FIRC frequency is within a certain range.
		CMU1 configuration
		void CMU1_init(void)
		{
		/* Enable the clock gate of the CMU1 */
		PCC->PCCn[PCC_CMU1_INDEX] = PCC_PCCn_CGC_MASK;
		
		CMU_FC_1->RCCR = 7;
		
		CMU_FC_1->HTCR = 47;
		CMU_FC_1->LTCR = 37;
		
		/* Enable the CMU1 interrupt */
		CMU_FC_1->IER = CMU_FC_IER_FHHIE_MASK | CMU_FC_IER_FLLIE_MASK;
		
		/* Enable the frequency check */
		CMU_FC_1->GCR = CMU_FC_GCR_FCE_MASK;
		
		/* Wait until the frequency check starts running */
		while(False == (CMU_FC_1->SR & CMU_FC_SR_RS_MASK) >> CMU_FC_SR_RS_SHIFT);
		}

SM_080	**Description**	**Code**
	For safety-relevant applications, the use of the clock monitors is mandatory. If the modules that the SCG monitors are used by the application safety function, the user shall verify that the clock monitors are not disabled and their faults are managed by the software.	**S32K11x_CLK -> clocks.c** **FIRC monitor** /* Enable the clock gate of the CMU0 */ PCC->PCCn[PCC_CMU0_INDEX] = PCC_PCCn_CGC_MASK; CMU_FC_0->RCCR = 7; /* Enable the CMU0 interrupt */ CMU_FC_0->IER |= CMU_FC_IER_FHHAEE_MASK | CMU_FC_IER_FLLAEE_MASK; /* Enable the frequency check */ CMU_FC_0->GCR = CMU_FC_GCR_FCE_MASK;
	Implementation	

	The corresponding clock monitors are enabled before executing any functional safety application. The clock sources with clock monitors are the SPLL (S32K14x only), the FIRC(S32K11x only) and the SOSC.	`S32K14x_CLK -> clocks.c`
		
		`SOSC monitor`
		`/* Enable the SOSC clock monitor */`
		`SCG->SOSCCSR |= SCG_SOSCCSR_SOSCCM_MASK;`
		
		`/* Configure the SOSC clock monitor to trigger an interruption when an error is detected*/`
		`SCG->SOSCCSR &= ~SCG_SOSCCSR_SOSCCMRE_MASK;`
		
		`SPLL monitor`
		`/* Enable the SPLL monitor */`
		`SCG->SPLLCSR |= SCG_SPLLCSR_SPLLCM_MASK;`
		
		`/* Configure the SPLL Clock Monitor to trigger an interruption when an error is detected */`
		`SCG->SPLLCSR &= ~SCG_SPLLCSR_SPLLCMRE_MASK;`

SM_081	**Description**	**Code**

	The following supervisor functions are required: Loss of external clock, SPLL frequency higher than the (programmable) upper frequency reference and SPLL frequency lower than the (programmable) lower frequency reference.	`S32K14x_CLK -> clocks.c`
		
		`SPLL monitor`
		`/* Enable the SPLL monitor */`
		`SCG->SPLLCSR |= SCG_SPLLCSR_SPLLCM_MASK;`
		
		`/* Configure the SPLL Clock Monitor to trigger an interruption when an error is detected */`
		`SCG->SPLLCSR &= ~SCG_SPLLCSR_SPLLCMRE_MASK;`

	Implementation	
	The SPLL clock monitor is enabled after the configuration of the clock is done and before executing any functional safety application.	

Flash

The S32K1xx has a nonvolatile flash memory to store program code. The flash memory is protected using Error Correcting Codes (ECC), it is capable of correcting single bit errors and of detecting double bit errors.

2.6.1. Flash memory

The S32K1xx has no way of injecting an ECC error on flash, the Error Injection Module is only for the ECC on RAM. Therefore it is suggested to follow the recommendations that are stated in the S32K1xx Reference Manual.

Assumptions		
	Description	**Code**

SM_114	The software using the EEPROM for storage of information will use checks to detect incorrect data returned from the EEPROM emulation.	S32K1xx_Flash -> eeprom.c
		
		`/* Verify if both CRCs are equal, if not then return an error */`
		`if(Before_store_CRC != After_store_CRC)`
		`{`
		`status = EEPROM_ERROR;`
		`}`

	Implementation	
	A write function was implemented, it performs the data storing and internally verifies the data. This is done, creating a CRC before writing the data and an other one after. They are compared, if the CRCs are not equal, an error is returned.	
	Description	**Code**
SM_116	A software test should be implemented to check for potential multi-bit errors introduced by permanent failures in the flash controller. It is assumed that if the embedded controller is classified as safety relevant, the activation of embedded controller is accompanied by flash integrity checks. First, one shall check that the started flash related command was completed and returned with a pass status.	-

	Depending on the safety application, the flash integrity checks shall be done for code flash or/and data flash. Safety relevant code and data shall be saved in flash with a CRC or hash signature to detect any integrity violation.	
	Implementation	
	Bootloader crea un CRC del Código, lo almacena en una direccion conocida. Despues se calcula de nuevo el CRC y se compara.	
SM_117	**Description**	**Code**

	A software safety mechanism shall be implemented to ensure the correctness of any write operation to the flash memory.	S32K1xx_Flash -> eeprom.c
		
		`while(wordQty)`
		`    {`
		`        Read_Buffer[index] = *eeprom_ptr;`
		`/*Data form the EEPROM and initial buffer are compared*/`
		`/*If the data are equal, increment the variable*/`
		`        if(Read_Buffer[index] == *buff)`
		`                read_verify++;`
		`/*Increment the value and the address direction*/`
		`        index++;`
		`        eeprom_ptr++;`
		`        buff++;`
		`        /*Decrease variable*/`
		`        wordQty--;`
		`    }`
		
		`/*If the value of read verify is equal to the elements into the array*/`
		`if(BUFFER_SIZE == read_verify)`
		`        return EEPROM_OK;`
		`else`
		`        return EEPROM_ERROR;`

	Implementation	
	Read after writing and verifying the data is correct.	
SM_119	**Description**	**Code**
	The Flash memory ECC failure reporting path should be checked to validate if detected ECC faults are correctly reported.	-

	Implementation	
	Codigo con interrupcion y todo listo para recibir un error de ECC en flash	
SM_118	**Description**	**Code**
	Depending on the application type and its safety requirements regarding the security subsystem, a set of software checks is recommended to be implemented to guarantee data integrity involved in a security operation.	-
	Implementation	
	-	

SRAM

The on-chip RAM is split in two regions: SRAM_L and SRAM_U. The RAM is implemented such that the SRAM_L and SRAM_U ranges from a contiguous block in the memory map.

2.7.1. Error Correction Code (ECC)

For the S32K11x there is only ECC for the SRAM_U, for the S32K14x there is ECC support for the SRAM_L and SRAM_U. The Error Reporting Module (ERM) is responsible for providing notifications of ECC errors detected in the SRAM channels.

The table below summarizes how to implement the assumptions related to the SRAM section of the Safety Manual in section 5.5.

Assumptions		
	Description	**Code**
SM_113	It is assumed that if safety relevant data are stored in this memory, additional integrity checks are done. **Implementation** To protect the safety relevant data that could be stored in this memory, the single and double bit errors interrupts are enabled. If any of these errors is detected by the Error Reporting Module (ERM) an interrupt is triggered and the system should jump to a safe state.	`S32K14x_ERM_LPIT -> main.c` `/* Initialize the Error Reporting Module */` `ERM_init();` `S32K14x_ERM_LPIT -> ERM.c` `void ERM_init(void)` `{` `/* Clear the pending interruptions before they are enabled */` `ERM->SR0 |= ERM_SR0_SBC0_MASK |ERM_SR0_NCE0_MASK | ERM_SR0_SBC1_MASK |ERM_SR0_NCE1_MASK;` `/* Enable the Single Error Correction Interrupt for the SRAM_L */` `ERM->CR0 |= ERM_CR0_ESCIE0_MASK;` `/* Enable the Non-Correctable Interrupt for the SRAM_L */` `ERM->CR0 |= ERM_CR0_ENCIE0_MASK;` `/* Enable the Single Error Correction Interrupt for the SRAM_U */` `ERM->CR0 |= ERM_CR0_ESCIE1_MASK;` `/* Enable the Non-Correctable Interrupt for the SRAM_U */` `ERM->CR0 |= ERM_CR0_ENCIE1_MASK;` `}`

Processing modules

2.8.1. Debug mode

The debugging mode is a possible source of failure if it is activated during functional safety applications. In this mode the core could be halted, breakpoints could stop the execution, the core registers could be modified and the registers from functional safety modules could be modified while running. Therefore, the application should not enter debugging mode during functional safety applications.

The table below summarizes how to implement the assumptions related to the debug mode section of the Safety Manual in section 5.6.2.1.

Assumptions		
SM_047	**Description**	**Code**
	Debugging will be disabled in the field while the device is being used for safety-relevant functions.	NA
	Implementation	
	When a functional safety application is running in the field, no debugging device should be connected. Debugging should be disabled to avoid interruption the execution.	
SM_048	**Description**	**Code**

	If modules like the Watchdog Timer (WDOG), Low Power Serial Peripheral Interface (LPSPI), Low Power Periodic Interrupt Timer (LPIT), FlexCAN, or in general any modules which can be frozen in debug mode, are functional safety relevant, it is required that application software configure these modules to continue execution during debug mode, and not freeze the module operation if debug mode is entered.	`S32K1xx_ERM_LPIT -> LPIT.c`
		
		`LPIT`
		`/* Allow the timer channels to continue to run in debug mode */`
		`LPIT0->MCR |= LPIT_MCR_DBG_EN_MASK;`
		
		`S32K14x_CLK_LPIT_VLPS_WDOG -> WDOG.c`
		
		`WDOG`
		`/* Configure the WDOG */`
		`WDOG->CS |= WDOG_CS_EN_MASK /* Enable the WDOG */`
		`        | WDOG_CS_CLK(WDOG_CLK_LPO) /* Set the LPO as the WDGO clock source */`
		`        | WDOG_CS_UPDATE_MASK /* Enable the WDOG to allow updates */`
		`        | WDOG_CS_CMD32EN_MASK /* Enable the WDOG support for 32 bit refresh/unlock command write words */`
		`        | WDOG_CS_INT_MASK /* Enable the WDOG interrupt, the reset is delayed 128 bus clocks */`

	Implementation	

	The debug mode is enabled on the LPIT and WDOG to ensure they keep operating even if there exists a debug session. Functional safety applications should not be halted due to debugging.	`| WDOG_CS_STOP_MASK /* Enable the WDOG to work on stop mode */`
		 `| WDOG_CS_PRES_MASK /* Enable the WDOG prescaler to divide the CLK by 256 */`
		 `| WDOG_CS_DBG_MASK; /* Allow the WDOG counter to continue running while debugging */`

2.8.2. Stack

A common fault is the stack overflow or underflow, which is caused by systematic faults within the application software. The overflow occurs when the application is using too much memory, also known as pushing too much information into the stack. The underflow occurs when the application is reading too much information, also known as popping too much information from the stack. These faults could be detected using data watchpoints.

The table below summarizes how to implement the assumptions related to the stack section of the Safety Manual in section 5.6.3.1.

Assumptions		
	Description	**Code**

SM_139	When stack underflow and stack overflow due to systematic faults within the application software endangers the system level, functional safety mechanisms may be implemented to detect stack underflow and stack overflow faults.	`S32K1xx_DWT -> main.c`
		
		`/* Initialize the DWT module and set the watchpoint to detect stack overflow */`
		`DWT_init();`
		
		`S32K1xx_DWT -> DWT.c`
		
		`void DWT_init(void)`
		`{`
		`        /* Pointer to the StackPointer register */`
		`    register uint32_t * stackPointer asm("sp");`
		`    /* Volatile pointer to the memory address of the DEMCR register */`
		`    volatile uint32_t * DEMCR_reg = (volatile uint32_t *) 0xE000EDFC;`
		
		`        *DEMCR_reg = DEMCR_TRCENA_MASK | DEMCR_MON_EN_MASK;`
		
		`        /* Set comparator value to stack pointer value minus 16 words (Addresses will be compared with this value) */`
		`        DWT->COMPn[0].COMP = (uint32_t)stackPointer - (16 * sizeof(uint32_t));`
		`        /* All bits are compared to find a specific address */`
		`        DWT->COMPn[0].MASK = 0;`
		`        /*`
		`         * Generate watchpoint debug event, see the ARMv7-M Architecture Reference Manual,`
		`         * Table C1-14 DWT address comparison functions`
		`         */`
		`        DWT->COMPn[0].FUNCTION = DWT_FUNCTIONn_FUNCTION_(7);`
		`        /* Perform address comparison and match for address */`
		`        DWT->COMPn[0].FUNCTION &= ~(DWT_FUNCTIONn_DATAVMATCH_MASK | DWT_FUNCTIONn_CYCMATCH_MASK);`
		
		`}`

	Implementation	
	The Debug Watchpoint and Trace (DWT) was used to set watchpoints in order to trigger events when the SP is out of the stack bounds. The DWT watchpoints are used to monitor if the SP reaches the stack bounds by comparing the addresses.	

2.8.3. S32K1xx configuration

Before executing functional safety application, it must be verified that the S32K1xx initialization is correct. This must be done after the start up. The minimum checks that must be passed are:

- LPO enabled
- WDOG enabled
- WDOD fast test
- Error handling in SCG and RCM registers
- IRC_SW_CHECK
- ERM events notification
- Clock monitors enabled

The table below summarizes how to implement the assumption related to the S32K1xx configuration section of the Safety Manual in section 5.6.3.2.

Assumptions		
	Description	**Code**

SM_140	Application software must verify that the initialization of the S32K1xx is correct before activating the safety-relevant functionality.
	Implementation
	Before executing any functional safety application, the system should be initialized correctly. The section 2.7.3 Safety configuration requirements should be covered.
	The LPO needs to be enabled and configured.
	The WDOG should be enabled with the appropriate period to assure that FTTI will be covered. The WDOG must pass the fast test.
	The errors reported in the SCG and RCM registers must be checked and safety measures must be taken according to the results of those checks.
	The FIRC frequency must be checked using two timers with two different clock sources.
	The ERM must be enabled, as well as their interrupts.
	The clock monitors must be enabled. For the S32K11x series the CMU must be used to check the FIRC. For the S32K14x series the SPLL and the SOSC monitors must be enabled.
		`S32K1xx_Safety_Configuration -> main.c`
		
		`Please refer to the Safety Configuration project, because it is al focused on covering this assumption.`

2.8.4. MPU

The Memory Protection Unit is a mechanism that prevents masters from accessing restricted memory regions. The protection is done at the Crossbar Switch level. This module assigns access rights to the different memory regions to the Crossbar switch masters.

The table below summarizes how to implement the assumptions related to the MPU section of the Safety Manual in section 5.6.3.2.

Assumptions		
	Description	**Code**

SM_094	It is assumed that the system MPU is checked for correct functionality before it is used in safety applications. One can configure the possible access rights of each present master and check for expected system reaction. The check shall be done once within L-FTTI (at start up).	S32K1x_mpu -> MPU.c
		
		`/* Start Address */`
		`MPU->RGD[2].WORD0 = 0x0003FF00;`
		`/* End Address */`
		`MPU->RGD[2].WORD1 = 0x0003FF1F;`
		
		`/* Core user mode WX`
		` * Core supervisor mode WX`
		` * DMA user mode RWX`
		` * DMA supervisor mode RWX */`
		
		`MPU->RGD[2].WORD2 |=`
		`MPU_RGD_WORD2_M0UM(3)`
		`| MPU_RGD_WORD2_M0SM(3)`
		`| MPU_RGD_WORD2_M2UM(7)`
		`| MPU_RGD_WORD2_M2SM(3);`
		
		`MPU->RGDAAC[2] |=`
		`MPU_RGD_WORD2_M0UM(3)`
		`| MPU_RGDAAC_M0SM(3)`
		`| MPU_RGDAAC_M2UM(7)`
		`| MPU_RGDAAC_M2SM(3);`
		
		`//Enable the region2`
		`MPU->RGD[2].WORD3 = MPU_RGD_WORD3_VLD_MASK;`

	Implementation	
	Enable the MPU and protect different regions of memory for the crossbar masters with the different modes (Supervisor/User)	
	Description	**Code**
SM_095	System level functional safety integrity measures must cover bus operations to reduce the likelihood of shared resources being erroneously modified by the present masters (Core, eDMA).	-
	Implementation	

	-	

2.8.5. Nested Vectored Interrupt Controller (NVIC)

The NVIC is responsible for prioritizing, blocking and directing the Interrupt Requests (IRQs). Before executing functional safety applications, there must be a way to detect spurious or missing IRQs.

The table below summarizes how to implement the assumptions related to the Nested Vector Interrupt Controller (NVIC) section of the Safety Manual in section 5.6.6.

Assumptions		
	Description	**Code**

SM_098	It is assumed that application software will detect the critical failure modes of NVIC for all safety critical interrupts.	S32K14x_ERM_LPIT -> main.c
		
		`/* Verify if the interrupt request is valid */`
		`Verification_result =`
		`Interrupt_verification(Current_channel);`

	Implementation	S32K14x_ERM_LPIT -> main.c
		
		`/* If the interruption is enabled, the`
		`verification_status is Verification_pass */`
		`if(True == (ERM->CR0 & ERM_CR0_ESCIE0_MASK) >>`
		`ERM_CR0_ESCIE0_SHIFT)`
		`{`
		`        /* If the module has a pending interrupt */`
		`        if(True == (ERM->SR0 & ERM_SR0_SBC0_MASK) >>`
		`ERM_SR0_SBC0_SHIFT)`
		`        {`
		`                /* If the interruption is enabled in`
		`the NVIC module, the verification_status is`
		`Verification_pass */`
		`                if(True == (S32_NVIC->ISER[1] & ( 1 <<`
		`(ERM_single_fault_IRQn % 32))) >>`
		`(ERM_single_fault_IRQn % 32))`
		`                {`
		`                        /* If the currently executing`
		`interrupt is the one that triggered the ISR */`
		`                        if(CurrentInterrupt ==`
		`(ERM_single_fault_IRQn + 16))`
		`                        {`
		`                                /* It passed all the`
		`verifications, therefore the interrupt request is`
		`correct */`
		`                                Verification_status =`
		`Verification_pass;`
		`                        }`
		`                }`
		`        }`
		`}`

	A function was implemented to verify if the interruption should be performed or if it was triggered erroneously. It is checked if the interrupt was enabled in the module, if there is a pending interruption in the module, if the interruption was enabled in the NVIC module and if the currently executed interruption is the one that triggered the ISR.	
	Description	**Code**

SM_099	Periodic low latency IRQs will use a running timer/counter to ensure their call period is expected.	S32K1xx_ERM_LPIT -> main.c
		
		`/* If it is the first time that the IRQ is executed`
		`start the LPIT */`
		`if(False == First_IRQ_flag)`
		`{`
		`        /* The flag is set */`

	Implementation	

	The LPIT is used to measure the period of time it takes to the ERM IRQ to repeat. Based on that measurement it can be determined if the period was not the expected one.	`First_IRQ_flag = 1;`
		
		`/* Start the timer if the flag is equal to zero */`
		`LPIT_start();`
		`}`
		`/* If it is the second time that the IRQ is executed stop the LPIT */`
		`else if(True == First_IRQ_flag)`
		`{`
		`/* The flag is cleared */`
		`First_IRQ_flag = 0;`
		
		`/* Stop the timer if the flag is set and save the counter value */`
		`ERM_IRQ_period = LPIT_MAX_PERIOD - LPIT_stop();`
		`}`

SM_100	**Description**	**Code**

	Applications that are not resilient against spurious or missing interrupt requests may need to include detection or protection measures on the system level.	`S32K1xx_ADC_CRC_PMC -> main.c`
		
		`void LVD_LVW_IRQHandler(void)`
		`{`
		`if(True == ((PMC->LVDSC2 & PMC_LVDSC2_LVWF_MASK) >> PMC_LVDSC2_LVWF_SHIFT))`
		`{`
		`/* Turn the blue LED on to show that the Low Voltage Warning Interrupt was triggered */`
		`PTD->PSOR |= 1 << BLUE_LED;`
		
		`/* Erase the LVWF which triggers the interruption */`
		`PMC->LVDSC2 |= PMC_LVDSC2_LVWACK_MASK;`
		
		`for(;;)`
		`{`
		
		`}`
		`}`
		`}`

	Implementation	
	In each handler it must be verified that the interrupt was triggered by the corresponding request.	

2.8.6. Enhanced Direct Memory Access (eDMA)

The eDMA is a modules capable of performing complex data transfers with almost no intervention from the core. This modules computes the source and destination addresses.

The table below summarizes how to implement the assumptions related to the Enhanced Direct Memory Access section of the Safety Manual in section 5.6.7.

Assumptions		
	Description	**Code**

SM_101	The eDMA will be supervised by software which detects spurious, excessive, or constant activation. **Implementation** A check sum is calculated before the data transfer, recalculated after the transfer and then both values are compared. If the result is not the same, then a problem occurred during the data transfer and the data should not be used.	`S32K1xx_eDMA_CRC -> main.c`
		
		`/* Initialize the DMA Transfer Control Descriptors */`
		`DMA_TCD_init((uint32_t) Source1, Size1, (uint32_t *) &Dest);`
		
		`/* Turn the blue LED on two times to show that the signature will be send */`
		`RGB_blink(BLUE_LED);`
		
		`/* Transmit the source1 using the eDMA*/`
		`DMA_send();`

	Description	**Code**
SM_102	Applications that are not resilient to spurious, or missing functional safety-relevant, eDMA requests cannot use the LPIT module to trigger functional safety-relevant eDMA transfer requests. **Implementation** In these cases, the eDMA should be triggered by any module other than the LPIT. The TRGMUX or other modules could be used.	NA
	Description	**Code**

SM_104	If safety-relevant software is using the eDMA to transfer data to a non-replicated peripheral or within the RAM, the following holds: "always on" channels of the eDMA Channel Mux should	`S32K1xx_eDMA_CRC -> eDMA.c`
		
		`void DMA_send(void)`
		`{`
		`    /* Loop till DONE = 1 (Major loop is completed) */`
		`    while (!((DMA->TCD[0].CSR >> DMA_TCD_CSR_DONE_SHIFT) & 1))`

	not be used. Instead, the eDMA should be triggered by software. If "always on" channels are used, their failure has to be detected by software. In this case, software must ensure that the eDMA transfer was triggered as expected at the correct rate and the correct number of times. This test should detect unexpected, spurious interrupts.	`    {`
		`        /* Set channel 0 START bit to initiate next minor loop */`
		`        DMA->SSRT = 0;`
		
		`        /* Wait for a minor loop to be completed */`
		`        while (((DMA->TCD[0].CSR >> DMA_TCD_CSR_START_SHIFT) & 1) |        /* Wait for START = 0 */`
		`        ((DMA->TCD[0].CSR >> DMA_TCD_CSR_ACTIVE_SHIFT) & 1));       /* and ACTIVE = 0     */`
		`    }`
		
		`    /* Clear DONE bit of channel 0 */`
		`    DMA->CDNE = 0;`
		`}`

	Implementation	
	Instead of using always on channels, the data transfer is triggered by software. This is done by using the Transfer Control Descriptor (TCD) of the eDMA.	

2.8.7. Watchdog timer (WDOG)

The WDOG is a safety feature that ensures that the application is executing as planned. It detects if the CPU is stuck in an infinite loop or executing non planned code. This is done using a refresh mechanism that prevents the timer from completing the count and therefore resetting the system.

The table below summarizes how to implement the assumptions related to the WDOG section of the Safety Manual in section 5.6.9.

Assumptions		
	Description	**Code**
SM_067	Before the safety function is executed, the WDOG must be enabled and configuration registers hard-locked against modification. Additionally, it should be verified that the clock source is configured as LPO.	`S32K1xx_CLK_LPIT_VLPS_WDOG -> main.c` `/* Configure the WDOG */` `WDOG_init();` `S32K1xx_CLK_LPIT_VLPS_WDOG -> WDOG.c` `void WDOG_init(void)` `{` `/* Disable the interrupts */` `DisableInterrupts;` `/* Unlock WDOG */` `WDOG->CNT = WDOG_UNLOCK_VAL;` `/* Wait until registers are unlocked */` `while(False == ((WDOG->CS & WDOG_CS_ULK_MASK) >> WDOG_CS_ULK_SHIFT));`
	Implementation	
	The WDOG should always be a backup in a functional safety application. In case the execution is trapped in a code section, the WDOG will not be refreshed therefore causing a reset. In this example, before entering into the VLPS mode, the watchdog is unlocked, configured and enabled. The LPO is set as the WDOG clock. If the core does not wakeup from VLPS in a certain period of time, the WDOG will trigger a reset.	`/* Set the WDOG timeout value according to the FTTI */` `WDOG->TOVAL = WDOG_PERIOD;` `/* Configure the WDOG */` `WDOG->CS |= WDOG_CS_EN_MASK` `/* Enable the WDOG */` `| WDOG_CS_CLK(WDOG_CLK_LPO)` `/* Set the LPO as the WDGO clock source */` `| WDOG_CS_UPDATE_MASK` `/* Enable the WDOG to allow updates */` `| WDOG_CS_CMD32EN_MASK` `/* Enable the WDOG support for 32 bit refresh/unlock command write words */` `| WDOG_CS_INT_MASK` `/* Enable the WDOG interrupt, the reset is delayed 128 bus clocks */` `| WDOG_CS_STOP_MASK` `/* Enable the WDOG to work on stop mode */` `| WDOG_CS_PRES_MASK` `/* Enable the WDOG prescaler to divide the CLK by 256 */` `| WDOG_CS_DBG_MASK;` `/* Allow the WDOG counter to continue running while debugging */` `/* Wait until the configuration takes effect */`

		`        while(False == ((WDOG->CS & WDOG_CS_RCS_MASK)`
		`>> WDOG_CS_RCS_SHIFT));`
		
		`        /* Enable the interrupts */`
		`        EnableInterrupts;`
		`}`

SM_202	**Description**	**Code**

	The WDOG time window settings must be set to a value less than the FTTI. Detection latency shall be smaller than the FTTI.	`S32K1xx_CLK_LPIT_VLPS_WDOG -> WDOG.c`
		
		`/* Configure the WDOG */`
		`WDOG->CS |= WDOG_CS_EN_MASK /* Enable the WDOG */`
		`          |  WDOG_CS_CLK(WDOG_CLK_LPO) /* Set the LPO as the WDGO clock source */`
		`          |  WDOG_CS_UPDATE_MASK  /* Enable the WDOG to allow updates */`
		`          |  WDOG_CS_CMD32EN_MASK  /* Enable the WDOG support for 32 bit refresh/unlock command write words */`
		`          |  WDOG_CS_INT_MASK /* Enable the WDOG interrupt, the reset is delayed 128 bus clocks */`
		`          |  WDOG_CS_STOP_MASK /* Enable the WDOG to work on stop mode */`
		`          |  WDOG_CS_PRES_MASK /* Enable the WDOG prescaler to divide the CLK by 256 */`
		`          |  WDOG_CS_DBG_MASK; /* Allow the WDOG counter to continue running while debugging */`

	Implementation	
	The FTTI is application specific. It must be calculated and set as the WDOG timeout value. In the SM_211 it is assumed that the FTTI is 100 ms.	
SM_069	**Description**	**Code**

	It is the responsibility of the application software to insert control flow checkpoints with the required granularity as required by the application.	`S32K1xx_CLK_LPIT_VLPS_WDOG -> WDOG.c`
		
		`void LPIT0_Ch0_IRQHandler(void)`
		`{`
		`        /* Stop the LPIT channel 1 and get the counter value */`
		`        LPIT_CH1_current_value = LPIT_MAX_PERIOD - LPIT_CH1_stop();`
		
		`        /* Verify if the LPIT interrupt flag caused the interruption */`
		`        if(True == ((LPIT0->MSR & LPIT_MSR_TIF0_MASK) >> LPIT_MSR_TIF0_SHIFT))`
		`        {`
		`                /* Disable the interrupts */`
		`                DisableInterrupts;`
		
		`                /* Reconfiguration to restore the RUN mode after VLPS */`
		`                SCG->RCCR  = SCG_RCCR_SCS(2)`
		`        /* Set the SIRC as the system clock source */`
		`                                   |`
		`SCG_RCCR_DIVCORE(1)      /* Set the CORE_CLK  to (4MHz) by dividing the SIRC     (8MHz) by 2 */`
		`                                   |`
		`SCG_RCCR_DIVBUS(1)              /* Set the BUS_CLK to (4MHz) by dividing the CORE_CLK (4MHz) by 1 */`
		`                                   |`
		`SCG_RCCR_DIVSLOW(3);     /* Set the FLASH_CLK to (1MHz) by dividing the CORE_CLK (4MHz) by 4 */`
		
		`                /* Set the PTD16 pin */`
		`                PTD->PTOR |= 1 << RED_LED;`
		
		`                /* Clear the flag which produced the interrupt */`
		`                LPIT0->MSR |= LPIT_MSR_TIF0_MASK;`
		
		`                /* Refresh the WDOG */`
		`                WDOG->CNT = WDOG_REFRESH_VAL;`

	Implementation	
	The WDOG should be used to monitor the application, therefore it should be refreshed periodically to ensure the correct functionality. The watchdog refresh is done periodically using the LPIT interrupt. This period should always be smaller than the application FTTI.	

		```
		                /* Enable the interrupts */
		                EnableInterrupts;
		        }

		        /* Restart the LPIT channel 1 */
		        LPIT_CH1_start();
		}
		```

2.8.8. Low Power Periodic Interrupt Timer (LPIT)

The LPIT is a timer with multiple channels that have a configurable 32 bit count which can trigger other modules on the device or trigger interrupts periodically. It has two modes of operation, the compare and the capture modes.

The table below summarizes how to implement the assumptions related to the LPIT section of the Safety Manual in section 5.6.10.

Assumptions		
	Description	**Code**

SM_107	When using the LPIT module, it should be used in such a way that a possible functional safety-relevant failure is detected by the Watchdog Timer (WDOG).
	Implementation
	The LPIT handler is used to refresh the WDOG one a FTTI. Therefore if a fault occurs during the execution, the WDOG will timeout, trigger an interruption and reset the system.
		S32K1xx_CLK_LPIT_VLPS_WDOG -> WDOG.c
		
		void LPIT0_Ch0_IRQHandler(void)
		{
		        /* Stop the LPIT channel 1 and get the counter value */
		        LPIT_CH1_current_value = LPIT_MAX_PERIOD - LPIT_CH1_stop();
		
		        /* Verify if the LPIT interrupt flag caused the interruption */
		        if(True == ((LPIT0->MSR & LPIT_MSR_TIF0_MASK) >> LPIT_MSR_TIF0_SHIFT))
		        {
		                /* Disable the interrupts */
		                DisableInterrupts;
		
		                /* Reconfiguration to restore the RUN mode after VLPS */
		                SCG->RCCR = SCG_RCCR_SCS(2)
		        /* Set the SIRC as the system clock source */
		                |
		SCG_RCCR_DIVCORE(1) /* Set the CORE_CLK to (4MHz) by dividing the SIRC (8MHz) by 2 */
		                |
		SCG_RCCR_DIVBUS(1) /* Set the BUS_CLK to (4MHz) by dividing the CORE_CLK (4MHz) by 1 */
		                |
		SCG_RCCR_DIVSLOW(3); /* Set the FLASH_CLK to (1MHz) by dividing the CORE_CLK (4MHz) by 4 */
		
		                /* Set the PTD16 pin */
		                PTD->PTOR |= 1 << RED_LED;
		
		                /* Clear the flag which produced the interrupt */
		                LPIT0->MSR |= LPIT_MSR_TIF0_MASK;
		
		                /* Refresh the WDOG */
		                WDOG->CNT = WDOG_REFRESH_VAL;
		
		                /* Enable the interrupts */
		                EnableInterrupts;
		        }
		
		        /* Restart the LPIT channel 1 */
		        LPIT_CH1_start();
		}

2.8.9. Low Power Mode Monitoring

The table below summarizes how to implement the assumption related to the Low Power Mode Monitoring section of the Safety Manual in section 5.6.11.

Assumptions		
	Description	**Code**
SM_082	If application uses Low Power mode, it is required to monitor the duration of LP mode. If the system does not wakeup within a specified period, the system will be reset by the monitoring circuitry.	S32K1xx_CLK_LPIT_VLPS_WDOG -> WDOG.c
	Implementation	
	While the systems is in Very Low Power Stop, the LPIT continues running and it is configured to wake up the processor once every FTTI. While the system is awake the watchdog is refreshed. If a problems occurs with the LPIT the watchdog will timeout and trigger an interruption and then a reset.	

```
S32K1xx_CLK_LPIT_VLPS_WDOG -> WDOG.c

void LPIT0_Ch0_IRQHandler(void)
{
        /* Stop the LPIT channel 1 and get the counter
value */
        LPIT_CH1_current_value = LPIT_MAX_PERIOD -
LPIT_CH1_stop();

        /* Verify if the LPIT interrupt flag caused the
interruption */
        if(True == ((LPIT0->MSR & LPIT_MSR_TIF0_MASK)
>> LPIT_MSR_TIF0_SHIFT))
        {
                /* Disable the interrupts */
                DisableInterrupts;

                /* Reconfiguration to restore the RUN
mode after VLPS */
                SCG->RCCR  = SCG_RCCR_SCS(2)
        /* Set the SIRC as the system clock source */
                                  |
SCG_RCCR_DIVCORE(1)     /* Set the CORE_CLK  to (4MHz)
by dividing the SIRC     (8MHz) by 2 */
                                  |
SCG_RCCR_DIVBUS(1)              /* Set the BUS_CLK
to (4MHz) by dividing the CORE_CLK (4MHz) by 1 */
                                  |
SCG_RCCR_DIVSLOW(3);    /* Set the FLASH_CLK to (1MHz)
by dividing the CORE_CLK (4MHz) by 4 */

                /* Set the PTD16 pin */
                PTD->PTOR |= 1 << RED_LED;

                /* Clear the flag which produced the
interrupt */
                LPIT0->MSR |= LPIT_MSR_TIF0_MASK;

                /* Refresh the WDOG */
                WDOG->CNT = WDOG_REFRESH_VAL;

                /* Enable the interrupts */
                EnableInterrupts;
        }

        /* Restart the LPIT channel 1 */
        LPIT_CH1_start();
}
```

2.8.10. Cyclic Redundancy Check (CRC)

The Cyclic Redundancy Check module generates 16/32 bits CRC code for error detection without using the CPU. It is useful for detecting corrupted data during transmission or storage. It has a programable polynomial and a transpose feature.

The table below summarizes how to implement the assumption related to the Cyclic Redundancy Check (CRC) section of the Safety Manual in section 5.6.12.

Assumptions		
	Description	**Code**

SM_070	The safety-relevant configuration registers shall be checked at least once per FTTI to verify their proper content.	**S32K1xx_ADC_CRC_PMC -> ADC.c**
		
		`/* Calculate the CFG2 CRC */`
		`CRC_CFG2_result = CRC_32_bit(CRC_polynomial, CRC_seed, ADC0->CFG2);`
		
		`/* Verify the CRC */`
		`if(CFG2_CRC_OFFLINE_VAL != CRC_CFG2_result)`
		`{`
		`/* If they are different jump to the safe state */`
		`ADC_safe_state();`
		`}`
		
		`/* Calculate the SC1 CRC */`
		`CRC_SC1_result = CRC_32_bit(CRC_polynomial, CRC_seed, ADC0->SC1[0]);`
		
		`/* Verify the CRC */`
		`if(SC1_CRC_OFFLINE_VAL != CRC_SC1_result)`
		`{`
		`/* If they are different jump to the safe state */`
		`ADC_safe_state();`
		`}`

	Implementation	
	For the registers of the ADC a CRC is calculated offline and then recalculated after the registers are configured. A comparison is done between the offline and the online CRC result. This is done to verify that they match, if the do not match the system jumps to a safe state.	

Peripheral

2.9.1. Communications

For any communication peripheral used in a safety relevant application the software needs to provide the safety measures to ensure they meet the safety requirements. A proper fault tolerant communication layer should be implemented for each protocol.

The table below summarizes how to implement the assumption related to the Communications section of the Safety Manual in section 5.7.1.

Assumptions		
	Description	**Code**
SM_051	It is recommended that communication over CAN interfaces is to be protected by a fault-tolerant communication protocol.	NA
	Implementation	
	The CAN physical layer in the S32K1xx family is capable of handling errors. The CAN frame has checksums that detect if there is an erroneous bit in the data transfer. There is also an Error Counter that disables the node when it detects 255 errors.	

2.9.2. I/O functions

The functional safety relevant peripherals must be ensured by the application. They are assumed to be used redundantly, to ensure that the information is transmitted or received correctly. Many approaches exist, therefore, it can be chosen the one that best fits the application requirements.

The table below summarizes how to implement the assumption related to the I/O functions section of the Safety Manual in section 5.7.2.

Assumptions		
	Description	**Code**

SM_133	Comparison of redundant operation of I/O modules is the responsibility of the application software, as no hardware mechanism is provided for this.	S32K1xx_FTM -> ADC.c
		
		/* Infinite loop */
		for(;;)
		{
		        /* Save the last edges */
		        Past_rising_edge_0 = Rising_edge_0;
		        Past_rising_edge_1 = Rising_edge_1;
		
		        /* Verify if a rising edge was detected */
		        Rising_edge_0 = FTM0_CH6_input_capture();
		        /* Verify if a rising edge was detected */
		        Rising_edge_1 = FTM1_CH3_input_capture();
		
		        /* Verify if edges were detected */
		        if(False != (Rising_edge_0 & Rising_edge_0))
		        {
		                /* Calculate the PWM period */
		                Edge_to_edge_0 = Rising_edge_0 -
		Past_rising_edge_0;
		                Edge_to_edge_1 = Rising_edge_1 -
		Past_rising_edge_1;
		
		                if(Edge_to_edge_0 != Edge_to_edge_1)
		                {
		                        /* Jump to the safe state if
		the periods are not equal */
		                        Safe_state();
		                }
		        }
		}

	Implementation	
	To achieve redundancy, multiple peripherals are used to read the same signal. Then the values are compared to verify the correct functionality of those GPIOs. For this assumption three FTM channels were used. One as PWM generator and two as input capture. They are used to measure the period of a PWM signal. Both periods are compared, they should be equal.	
	Description	Code

SM_137	When safety functions use digital input, system level functional safety mechanisms have to be implemented to achieve required functional safety integrity.	S32K1xx_FTM -> ADC.c
		
		/* Infinite loop */
		for(;;)
		{
		        /* Save the last edges */
		        Past_rising_edge_0 = Rising_edge_0;
		        Past_rising_edge_1 = Rising_edge_1;
		
		        /* Verify if a rising edge was detected */
		        Rising_edge_0 = FTM0_CH6_input_capture();
		        /* Verify if a rising edge was detected */
		        Rising_edge_1 = FTM1_CH3_input_capture();
		
		        /* Verify if edges were detected */
		        if(False != (Rising_edge_0 & Rising_edge_0))

	Implementation	
	In this case it is consider that the PWM that is being produced with the FTM is safety relevant. Therefore it should be covered	

	with a functional level mechanism. A redundancy check between two channels from different FTMs is performed to ensure the correct functionality.	
		```
		        {
		                /* Calculate the PWM period */
		                Edge_to_edge_0 = Rising_edge_0 -
		Past_rising_edge_0;
		                Edge_to_edge_1 = Rising_edge_1 -
		Past_rising_edge_1;

		                if(Edge_to_edge_0 != Edge_to_edge_1)
		                {
		                        /* Jump to the safe state if
		the periods are not equal */
		                        Safe_state();
		                }
		        }
		}
		```

2.9.3. Analog to Digital Converter (ADC)

The ADC module is not fully covered by functional safety, therefore, the software application must ensure that the needed safety level is achieved.

The table below summarizes how to implement the assumption related to the Analog to Digital Converter (ADC) section of the Safety Manual in section 5.7.4.

Assumptions		
	Description	**Code**

SM_130	When Analog-to-Digital Converter (ADC) of the S32K1xx are used in a safety function, suitable system level functional safety integrity measures must be implemented once per L-FTTI.	`S32K1xx_ADC_CRC_PMC -> main.c`
		
		`/* Start the ADC self-calibration sequence */`
		`ADC_self_calibration();`
		
		`S32K1xx_ADC_CRC_PMC -> ADC.c`
		
		`void ADC_self_calibration(void)`
		`{`
		`        /* Enable FIRCDIV 1 and 2 */`
		`        SCG->FIRCDIV |= SCG_FIRCDIV_FIRCDIV2(1) |`
		`SCG_FIRCDIV_FIRCDIV1(1);`
		
		`        /* Disable the clock for the ADC0 to make`
		`changes */`
		`        PCC->PCCn[PCC_ADC0_INDEX] &=`
		`~PCC_PCCn_CGC_MASK;`
		`        /* Select the FIRCDIV2_CLK as clock source */`
		`        PCC->PCCn[PCC_ADC0_INDEX] |= PCC_PCCn_PCS(3);`
		`        /* Enable the clock for the ADC0 */`
		`        PCC->PCCn[PCC_ADC0_INDEX] |= PCC_PCCn_CGC_MASK;`
		
		`        ADC0->SC3 = ADC_SC3_CAL_MASK     /* Start`
		`calibration sequence */`
		`                  | ADC_SC3_AVGE_MASK    /* Enable`
		`hardware averaging */`
		`                  | ADC_SC3_AVGS(3);     /* Average 32`
		`samples */`
		
		`        /* Wait for the ADC self-calibration sequence`
		`to finish */`
		`        while( 0 == ((ADC0->SC1[0] &`
		`ADC_SC1_COCO_MASK)>>ADC_SC1_COCO_SHIFT));`
		
		`}`

	Implementation	
	When the ADC is used in a safety relevant application, the calibration function should be used. This is done to ensure the correct functionality of the ADC. This calibration should be done once after every reset.	

2.9.4. Asynchronous Wake-up Interrupt Controller (AWIC) / External NMI

The table below summarizes how to implement the assumption related to the Asynchronous Wake-up Interrupt Controller (AWIC) / External section of the Safety Manual in section 5.7.5.

Assumptions		
	Description	**Code**
SM_126	If external NMI and Wake-up are used as a safety mechanism, especially if waking up within a certain timespan or at all is considered safety-relevant, it is required to implement corresponding system level measures to detect latent faults in the AWIC.	S32K1xx_CLK_LPIT_VLPS_WDOG -> main.c
	Implementation	
	A LPIT channel is used to measure the timespan it takes to another LPIT channel to wake up the MCU from VLPS. The timespan value is application dependent but should be one per FTTI. The time measured using the LPIT should be less than the application timespan.	

```
void LPIT0_IRQHandler(void)
{
        /* Stop the LPIT channel 1 and get the counter
value which is equal to the time it took the MCU to
wake up from VLPS */
        LPIT_CH1_current_value = LPIT_MAX_PERIOD -
LPIT_CH1_stop();

        /* Verify if the LPIT interrupt flag caused the
interruption */
        if(True == ((LPIT0->MSR & LPIT_MSR_TIF0_MASK)
>> LPIT_MSR_TIF0_SHIFT))
        {
                /* Disable the interrupts */
                DisableInterrupts;

                /* Reconfiguration to restore the RUN
mode after VLPS */
                SCG->RCCR  = SCG_RCCR_SCS(2)
        /* Set the SIRC as the system clock source */
                                  |
SCG_RCCR_DIVCORE(1)     /* Set the CORE_CLK  to (4MHz)
by dividing the SIRC     (8MHz) by 2 */
                                  |
SCG_RCCR_DIVBUS(1)              /* Set the BUS_CLK
to (4MHz) by dividing the CORE_CLK (4MHz) by 1 */
                                  |
SCG_RCCR_DIVSLOW(3);    /* Set the FLASH_CLK to (1MHz)
by dividing the CORE_CLK (4MHz) by 4 */

                /* Set the PTD16 pin */
                PTD->PTOR |= 1 << RED_LED;

                /* Clear the flag which produced the
interrupt */
                LPIT0->MSR |= LPIT_MSR_TIF0_MASK;


                /* Refresh the WDOG */
                WDOG->CNT = WDOG_REFRESH_VAL;

                /* Enable the interrupts */
                EnableInterrupts;

        }

        /* Restart the LPIT channel 1 */
        LPIT_CH1_start();
}
```

3. References

1. S32K1xx Series Safety Manual Rev. 4, 09/2018, by NXP Semiconductors.
2. S32K1xx Series Reference Manual Rev. 9, 09/2018, by NXP Semiconductors.
3. S32K1xx Data Sheet Rev. 9, 09/2018, by NXP Semiconductors.
4. S32K1xx ADC guidelines, spec and configuration Rev. 0, 08/2018, by NXP Semiconductors.
5. S32K1xx ECC Error Handling Rev. 0, 07/2019, bye NXP Semiconductors.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: AN00000000000001

Rev. n