Safe application development for AURIX™ Application Kit TC3xx Safety

32-bit TriCore™ AURIX™ TC3xx microcontroller

About this document

Scope and purpose

This application note describes product safety mechanisms and the actions that must be taken by the product system integrator to ensure the correct operation of the device. This document should be read in conjunction with the AURIX™ User's Manual and with the AURIX™ Safety Manual. The associated source code includes methods for fault injection used for testing purposes, alarm triggering, and the implementation of various safety mechanisms listed in the Safety Manual.

Attention: ***This document does not provide any code suitable for production. This document does not give a legally binding example for the implementation of safety-critical functions.***

Intended audience

This application note is written for system engineers, software engineers, and functional safety managers involved in the design or development of a safety-related system who are considering integrating the AURIX™ TC3xx microcontroller hardware as a Safety Element out of Context (SEooC) into their system.

Table of contents

Table of contents

Table of contents

1 Introduction

The development of a safe application can be a challenge when it comes to following strict safety rules. To ease the development of such applications, Infineon provides the TC3xx Safety Manual. The Safety Manual defines the safety mechanisms as an activity or a technical solution to avoid or control systematic failures and to detect random hardware failures or control random hardware failures.

Safety mechanisms are classified in two main types:

- The technical solution, which is internal to the microcontroller by hardware (HW) or software (SW)
- The technical solution, which is either in HW or SW, implemented at the *system level* by the system integrator

During the design phase of AURIX™ MCUs, the most common use cases have been taken into account, and safety requirements have been derived from these. For the implementation of these specific safety-related functions, different SMs must be implemented according to the specification in the Safety Manual. The specific SM is required depends on the modules used and the safety level required.

The intention of this application note is to provide implementation hints and code examples for many of these safety mechanisms. Therefore, this document is provided with example software optimized for the Application Kit Safety hardware, which is composed of an Application Kit - AURIX™ TC397 TFT (KIT_A2G_TC397_5V_TFT) from Infineon and the new Safety Demo Add-on Shield Board, which is called “Evaluation Board - AURIX™ TC3xx Safety” (EVABOARD_A2G_SAFETY) and the combination of both is called “AURIX™ Application Kit - TC3xx Safety” (APPKIT_A2G_SAFETY). The kit is used for demonstrating the implementation of the safety mechanisms and other diagnostics information.

In addition to the touchscreen and an ASCLIN shell interface, the add-on shield provides several buttons and switches to trigger the injection of faults into the system. An overview of all safety-related functions and SMs covered by the application note can be found in the tables ranging from Table 1 to Table 6.

1.1 Key features

The following key features are implemented and supported:

- Boot and startup procedure including all safety mechanisms involved
- Full SMU driver implementation including the following:
  - SMU core and SMU standby
  - Fault Signaling Protocol (FSP)
  - Emergency stop (ES)
  - Recovery timer (RT)
- Implementation of safety-related functions and the required safety mechanisms
- Fault injection for testing of various safety mechanisms:
  - PFlash ECC error injection
  - DMA error injection
  - Analog and digital acquisition error injection
  - Undervoltage error injection
  - Broken wire, etc.
- TFT touchscreen driver

- TLF35584 PMIC driver
- ASCLIN shell interface
- STM used for basic task scheduling

Table 1 Overview of Safety Related Functions covered by this application note

Safety Related Function	Covered (Yes/Partly/No)	Covered SMs/Functional Use Case (FUC)
Safe computation	Yes	• Safety Mechanism AMU*:* • Safety Mechanism CPU.*:* • Safety Mechanism STM:* • Safety Mechanism * • Safety Mechanism DMA.*:* • Safety Mechanism NVM.*:* • Safety Mechanism EMEM*:* • Safety Mechanism LMU*:* • Safety Mechanism SRI*:*
Analog acquisition	Yes	• FUC 0: Analog acquisition with redundant EVADC channels • FUC 1: Analog acquisition with redundant EDSADC channels • FUC 2: Analog acquisition with one EVADC channel and one EDSADC channel • FUC 3: Single analog acquisition with EVADC channels • FUC 4: Single analog acquisition with one EDSADC and one EVADC channel
Digital acquisition	Yes	• FUC 0: Digital acquisition with redundant TIM/TIM channels • FUC 1: Digital acquisition with redundant CCU6/TIM channels • FUC 2: Digital acquisition with redundant CCU6/GPT12 channels
Digital actuation	Yes	• FUC 0: Digital actuation with redundant TOM channels and IOM comparison • FUC 1: Digital actuation with redundant TOM/CCU6 channels and IOM comparison • FUC 2: Digital actuation with redundant TOM/TIM channels and application SW comparison • FUC 3: Digital actuation with redundant CCU6/GPT12 channels and application SW comparison
Sensor acquisition	Partly	• Safety mechanism SENT:CHANNEL_REDUNDANCY

Safety Related Function	Covered (Yes/Partly/No)	Covered SMs/Functional Use Case (FUC)
External communication	Partly	• Safety mechanism QSPI:SAFE_COMMUNICATION
Avoidance or detection of common-cause failures	Partly	• Safety mechanism PMS:VEXT_VEVRSB_ABS_RATINGS • Safety mechanism PMS:VEXT_VEVRSB_OVERVOLTAGE • Safety mechanism PMS:VX_FILTER • Safety mechanism :DTS_RESULT • Safety mechanism WATCHDOG_FUNCTION • Safety mechanism CLOCK:PLAUSIBILITY • Safety mechanism MONBIST_RESULT • Safety mechanism PORT:LOOPBACK • Safety mechanism PORT:REDUNDANCY
Safe state support	Partly	• Internal failure reporting • External failure reporting (FSP) • Alternate failure reporting (As FSP_ERROR_PIN_MONITOR is implemented)
Coexistence of HW/SW elements	Yes	• Safety mechanism ISR_MONITOR

Table 2 Overview of implemented Safety Mechanisms - Safe startup

Safety Mechanism (SM)	App. Note section	C function name
LBIST_CFG	3.3.2 & 6.1.2	safetyKitSswLbist
LBIST_MONITOR	3.3.2 & 6.1.2	safetyKitSswLbist
LBIST_RESULT	3.3.2 & 6.1.2	safetyKitSswLbist
MONBIST_CFG	3.3.3 & 6.1.3	Ifx_Ssw_Monbist
MONBIST_RESULT	3.3.3 & 6.1.3	Ifx_Ssw_Monbist
MCU_FW_CHECK	3.3.4	safetyKitSswMcuFwCheck
MCU_STARTUP	3.3.5	safetyKitSswMcuStartup
ALIVE_ALARM_TEST	3.3.6	safetyKitSswAliveAlarmTest
REG_MONITOR_TEST	3.3.7	safetyKitSswRegMonitorTest
MBIST	3.3.8 & 6.1.4	safetyKitSswMbist

Table 3 Overview of implemented Safety Mechanisms - Analog acquisition

Safety Mechanism (SM)	App. note section	C function name
CONVCTRL:CONFIG_CHECK	6.2.10	initCONVCTRL
EVADC:CONFIG_CHECK	6.2.10	initEVADCGroups
EVADC:DIVERSE_REDUNDANCY	6.2.10	initAAcqFuc0 initAAcqFuc2 initAAcqFuc4BrokenWR
EVADC:PLAUSIBILITY	6.2.10	plausibilityCheck
EVADC:VAREF_PLAUSIBILITY	6.2.10	evadcVarefPlausibilityCheck
EDSADC:DIVERSE_REDUNDANCY	6.2.10	initAAcqFuc1 initAAcqFuc2 initAAcqFuc4BrokenWR
EDSADC:PLAUSIBILITY	6.2.10	plausibilityCheck
EDSADC:VAREF_PLAUSIBILITY	6.2.10	edsadcVarefPlausibilityCheck

Table 4 Overview of implemented Safety Mechanisms - Digital acquisition

Safety Mechanism	App. note section	C function name
TIM_REDUNDANCY	6.2.11 & 6.2.11.2	checkRedundancyGTMTIM
GTM_CCU6_REDUNDANCY	6.2.11 & 6.2.11.2	checkRedundancyTIMCCU6
CCU6_CAPTURE_MON_BY_GPT12	6.2.11 & 6.2.11.2	plausibiltyCheckDAcqFuc2
TIM_CLOCK_MONITORING	6.2.11 & 6.2.11.2	initEclkMonitoring

Table 5 Overview of implemented Safety Mechanisms - Digital actuation

Safety Mechanism	App. note section	C function name
IOM_ALARM_CHECK	6.2.11 & 6.2.11.3	alarmCheckGTMIOM
CCU6_GPT12_MONITORING	6.2.11 & 6.2.11.3	plausibilityCheckDActFuc3
TIM_CLOCK_MONITORING	6.2.11 & 6.2.11.3	initEclkMonitoring
TOM_TIM_MONITORING	6.2.11 & 6.2.11.3	gtmTimPwmMissionIsrDActFuc2

Table 6 Overview of implemented Safety Mechanisms – Other

Safety Mechanism	App. note section	C function name
.:REG_MONITOR_TEST	6.2.3.3	safetyKitRunRegMonitorTest
DMA: *	6.2.7	initAndRunDmaTransaction
DTS_CFG	6.2.8.6	initDieTemperatureSensors
DTS_RESULT	6.2.8.6	dtsMeasurementISR
CONVCTRL:ALARM_CHECK	6.2.10	initCONVCTRL
APPLICATION_SW_ALARM	6.2.13.1	softwareCoreAlarmTriggerSMU
STM:MONITOR	6.2.1.3	runStmMonitoring
CLOCK:OSC_MONITOR	6.2.8.2	IfxScuCcu_init
GTM_CONFIG_FOR_GTM	6.2.11 & 6.2.11.2	initDAcqFuc0
IOM_CONFIG_FOR_GTM	6.2.11 & 6.2.11.3	initDActFuc0
MON_REDUNDANCY_CFG	6.2.8.1	initVoltageMonitors
VX_MONITOR_CFG	6.2.8.1	initVoltageMonitors
SMU:CONFIG	4 & 6.2.13.1	initSMUModule enableFSPcoreSMU enableFSPstdbySMU
PFLASH: *	6.2.2.1	runInterityCheckPFLASH, runUpdateCheckPFLASH, runWordlineFailDetectPFLASH
EMEM:DATA_INTEGRITY	6.2.3.1	runDataIntegrityEMEM
ISR_MONITOR	6.2.7.1	isrMonitor
CLOCK:PLAUSIBILITY	6.2.8.2	initQSPI5ClockPlausibility
QSPI:SAFE_COMMUNICATION	6.2.9.1	initQSPISafeCommunication
PORT:LOOPBACK, PORT:REDUNDANCY	6.2.9.2	runPortLoopback, runPortRedundancy
SENT:CHANNEL_REDUNDANCY	6.2.9.3	initTLE5012Modules checkRedundancySENT

1.2 Abbreviations and acronyms

Note: For a list of abbreviations, acronyms, and safety-related definitions, see [1], [2], [3], and [4].

Note: In this document, the AURIX™ Application Kit - TC3xx Safety is also referred to as "Application Kit Safety" or "Safety Kit". Also, "Evaluation Board - AURIX™ TC3xx Safety" is also called "Safety Evaluation Board" or "Evaluation Board".

2 Demonstrator presentation

2.1 Hardware

The dedicated full name of the demonstrator is "AURIX™ Application Kit - TC3xx Safety" and it is composed of two PCBs:

- Application Kit - AURIX™ TC397 TFT (KIT_A2G_TC397_5V_TFT)
- Evaluation Board - AURIX™ TC3xx Safety (EVABOARD_A2G_SAFETY)

The Application Kit - AURIX™ TC397 TFT itself features a TC397 microcontroller, a TFT touch display, and a TLF35584 safe system power supply. The add-on Evaluation Board - AURIX™ TC3xx Safety is composed of multiple additional sensors, undervoltage protection circuitry, buttons, and switches allowing to demonstrate the behavior of AURIX™ MCUs in the presence of a fault. Different faults such as a lockstep error can be directly injected through the touch display of the Application Kit - AURIX™ TC397 TFT, PFlash error can be injected via a dedicated button on the Evaluation Board.

Note: Do not run the demo without the Evaluation Board - AURIX™ TC3xx Safety (add-on shield board) as multiple alarms will be reported.

Figure 1 shows the board with dedicated naming.

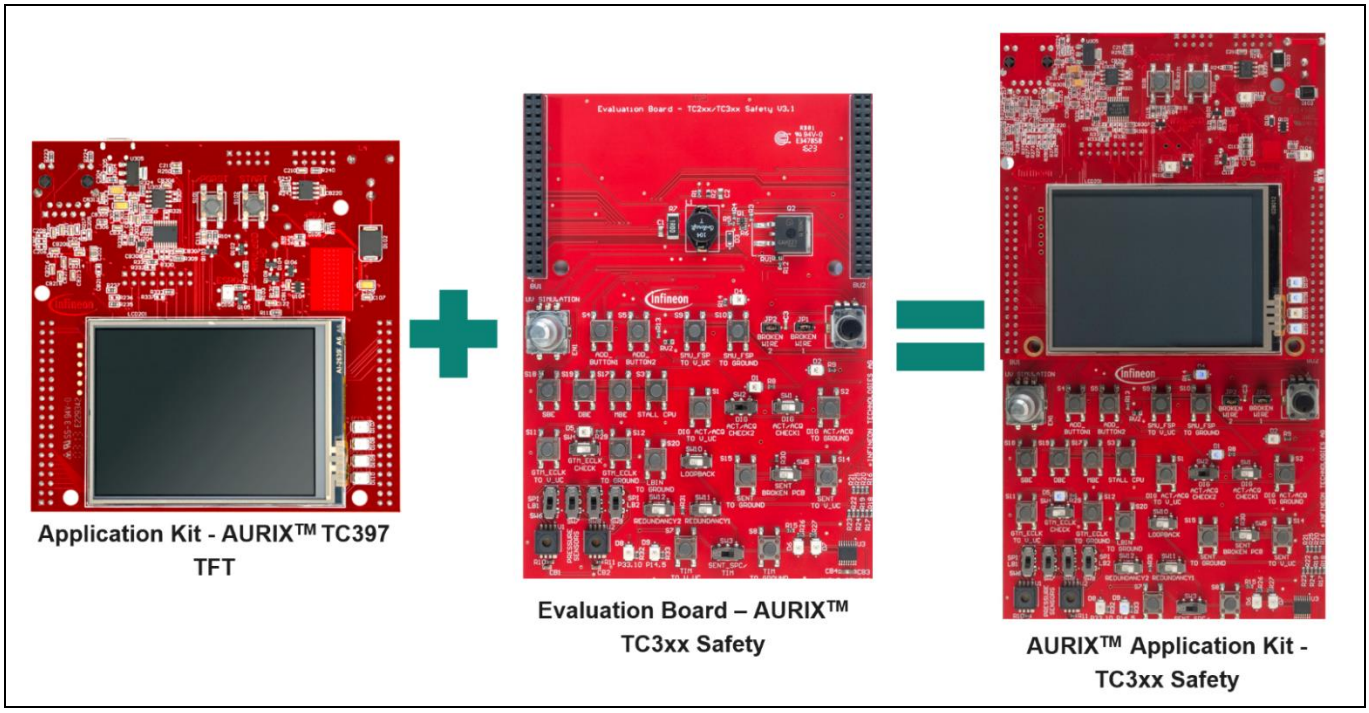


Figure 1 AURIX™ Application Kit - TC3xx Safety

Figure 2 shows the schematic of Evaluation Board - AURIX™ TC3xx Safety. See the Application Kit Manual TC3x7 for the schematic of Application Kit - AURIX™ TC397 TFT.

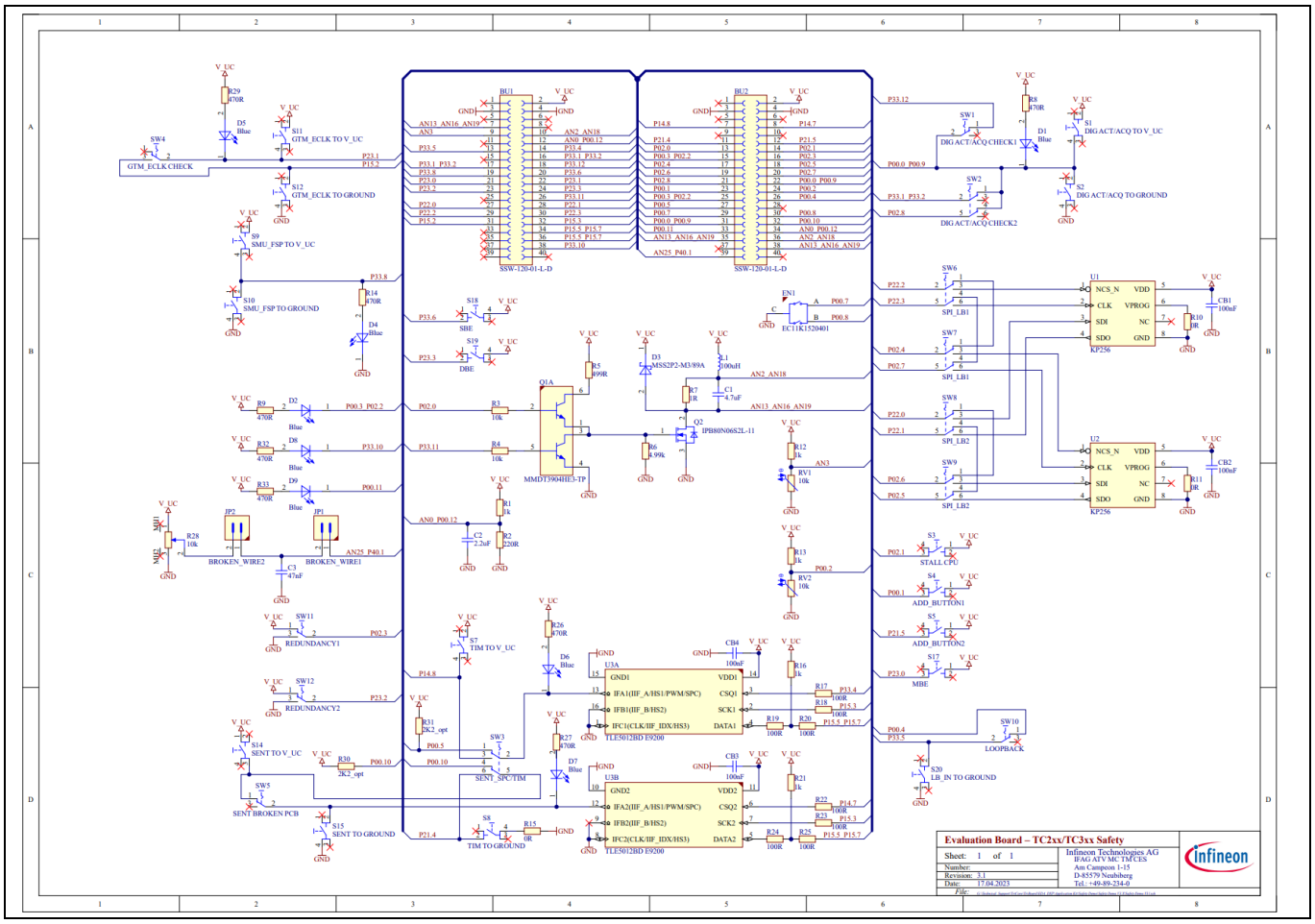


Figure 2 Schematic of Evaluation Board - AURIX™ TC3xx Safety (add-on shield board)

2.1.1 Safe components

Application Kit - AURIX™ TC397 TFT including TLF35584 itself has been designed to follow the safety guidelines. To demonstrate the safety application capability, the Safety Evaluation Board also contains other safety components, such as sensors and actuators. The following section provides a detailed overview of all components available on both PCBs.

2.1.1.1 TLF35584 power supply

TLF35584 is a multi-voltage safety supply for safety-relevant applications supplying 5 V or 3.3 V MCUs, transceivers, and sensors by an efficient and flexible pre-/post-regulator concept over a wide input voltage range. The multiple built-in safety features enable easy realization of microcontroller applications, fulfilling the highest Automotive Safety Integrity Level (ASIL-D):

- Independent voltage monitoring block with reset function
- Configurable functional and window watchdog
- Error monitoring
- 16-bit SPI interface
- Safe state control with two safe state signals with programmable delay
- Input voltage monitoring (overvoltage switch-off)

TLF35584 provides a windowed or question-and-answer watchdog; it must be configured for the required functionality. The application configures the power supply via SPI; by default, TLF35584 will start as a windowed watchdog via SPI. If not triggered, it will reset the system. The chip provides different voltages for ADC reference, communication, controller supply, and other functionalities. SPI communication to and from TLF35584 is protected with an odd parity bit.

See Section 5.1 for more information on the usage of TLF35584. See the latest datasheet for more information about TLF35584 [4].

Note: TLF35584 provides a Microcontroller Programming Support (MPS) mode, which is enabled by default via a hardware pull-up of the TLF35584 MPS pin to 5 V on the evaluation board. While being in MPS mode, the contribution of a watchdog or error monitoring failure to an AURIX™ reset via ROT is blocked.

2.1.1.2 TLE5102BD E9200 magnetic angle sensor

TLE5102BD magnetic angle sensor is a dual-die position sensor based on the giant magnetoresistance (GMR) effect. All connections and sensors are implemented twice. The sensor supports various interfaces to a controller, such as SPC (based on SENT SAE J2716), SPI, Hall, incremental interface, and PWM. SPC and SPI communication is end-to-end protected with a CRC checksum. SPI can be used in parallel with the other interfaces.

By default, the magnetic angle sensor is configured to the SPC interface mode but can be reconfigured to the necessary interface at runtime using SPI. For more information, see the TLE5102 BD datasheet.

2.1.1.3 KP256 pressure sensor

The KP256 pressure sensor is connected to the microcontroller via SPI. It also houses a temperature sensor. The communication is protected with an odd parity bit. This sensor demonstrates the QSPI redundancy for future use case.

2.1.1.4 Other Safety Board components

- Pin-to-pin circuitry with switches to inject hardware faults
- Redundant temperature sensors for safe analog measurement demonstration
- Potentiometer with switch for analog measurement with hardware fault injection
- Encoder
- Single Bit Error (SBE), Double Bit Error (DBE) and Multiple Bit Error (MBE) buttons to inject NVM PFlash error
- Debug LEDs for different purposes
- Action switches (error pin, Emergency Stop, etc.)
- Current pump circuitry to demonstrate undervoltage safety mechanisms
- Four parallel switches to change the functionality between pressure sensor and safe QSPI communication (where two QSPI of the AURIX™ TC3xx MCU are connected with each other (loopback) to simulate a safe communication safety mechanism)
- Connector for broken-wire case simulation

2.1.2 Default switch configuration

This section describes the default hardware configuration where no fault injection is active. In general, the default switch configuration does not interrupt each signal.

Attention: This default configuration must be observed to avoid any unintended alarm directly after power-on reset.

Table 7 Default state of switches and LEDs

Switch	Default state	Comment
SW1	Right	Used
SW2	Right	Used
SW3	Right	Do not care/not used
SW4	Left	Used
SW5	Right	SENT/SPC
SW6	Down	Used
SW7	Down	Used
SW8	Down	Used
SW9	Down	Used
SW10	Right	Used
SW11	Right	Used
SW12	Right	Used
D1	On/~50% brightness	PWM out
D2	Depends on the selected IO example	On when running the TOM_IOM demo
D4	On brightness	SMU_FSP
D5	On/50% brightness	GTM_EXCLK
D6	On	SENT Protocol
D7	On	SENT Protocol
D8	Off	Spare
D9	Toggling	QSPI Protocol
JP1	Connected	BWD
JP2	Connected	BWD

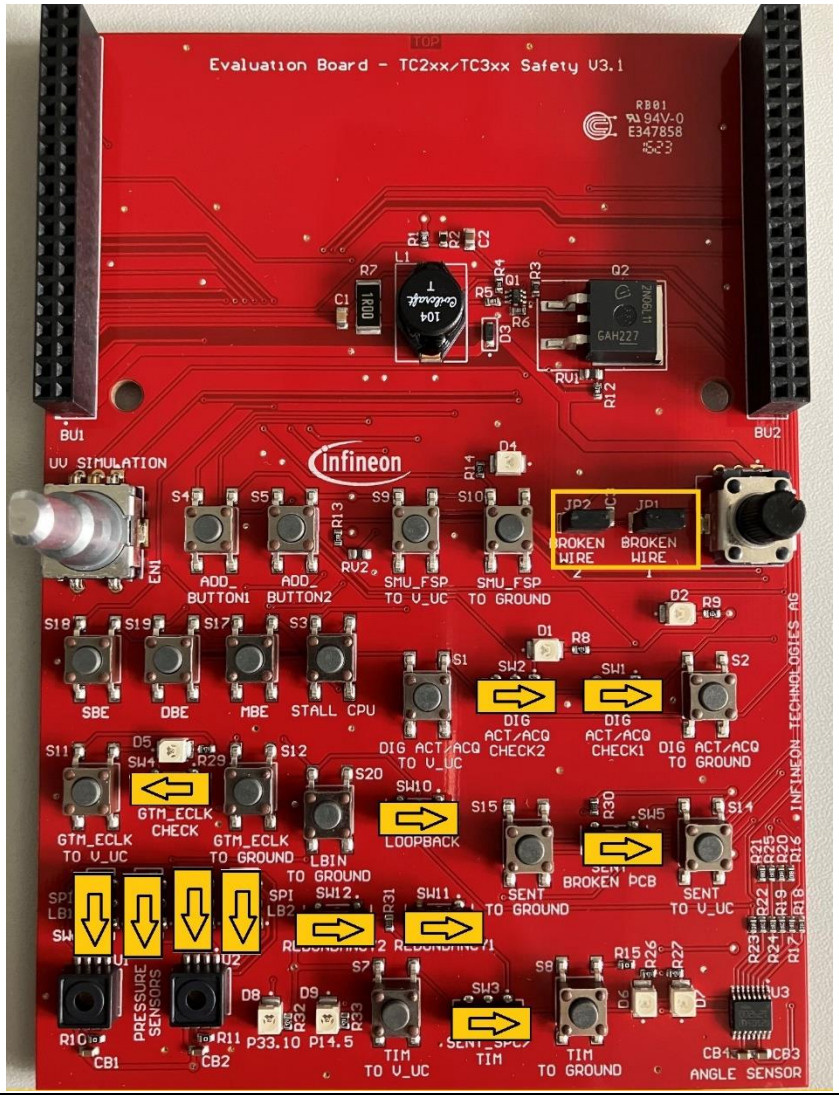


Figure 3 Default state of switches and jumper on the Evaluation Board - AURIX™ TC3xx Safety

2.1.3 Application Kit - TC397 TFT LEDs

Application Kit - TC397 TFT offers four LEDs (D107 to D110) which can be used by the application software. For Application Kit Safety, they are used for the following purposes:

- **LED0 (D107):** Signals the successful initialization of the application software
- **LED1 (D108):** Life hold indication, blinking serviced by CPU5.
- **LED2 (D109):** Indicates that a background task is running the infinite while loop to determine if any bit of an SMU alarm status register is set. If yes, which means an SMU alarm is active, this LED is turned on.
- **LED3 (D110):** Port Emergency Stop. LED “off” when ES is activated.

See the Application Kit TC397 TFT Manual TC3x7 for further information about all nine LEDs on the PCB.

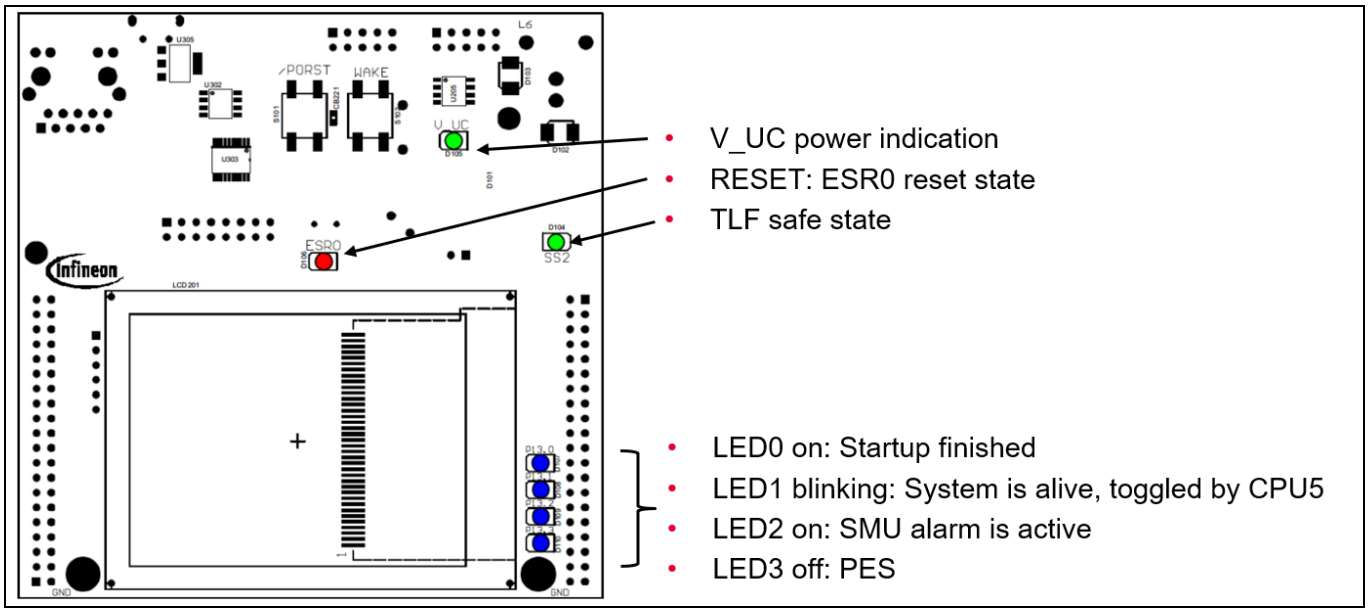


Figure 4 Application Kit TC397 TFT LED usage

2.2 AURIX™ MCU resource allocation

For more information about the AURIX™ MCU resource allocation and project configuration, see the following configuration files:

- *\AppSw\AppKit\Cfg_Illd\Configuration.h*
- *\AppSw\AppKit\Cfg_Illd\ConfigurationIsr.h*
- *\AppSw\AppKit\AppKit_Cfg.h*
- *\AppSw\SafetyKit\SafetyKit_Cfg.h*

See Table 9 and Table 10 for additional information on resource allocation.

2.3 Software overview

Figure 5 shows the project structure. In addition to the six *CpuX_Main.c (x=0...5)* files, the relevant application software is stored in the *AppSw* folder, which consists of the two folders *AppKit* and *SafetyKit* and the two *Stm_Scheduler* files used as System Timer (STM) for basic periodic task scheduling.

The *AppKit* (for Application Kit - AURIX™ TC397 TFT) folder contains software specific for the display, touch application, and ASCLIN shell Interface. The *SafetyKit* folder includes all the code for the safety features implemented and required for this specific application note.

The software project is free-of-cost without any legal binding, and it is developed in the free-of-charge AURIX™ Development Studio (ADS) integrated development environment (IDE). The project can be easily found in the ADS via import project functionality. It is not tested on other IDE platforms. Some software implementations are based on other AURIX™ - TC3xx Microcontroller expert trainings, which can be found on the Infineon webpage.

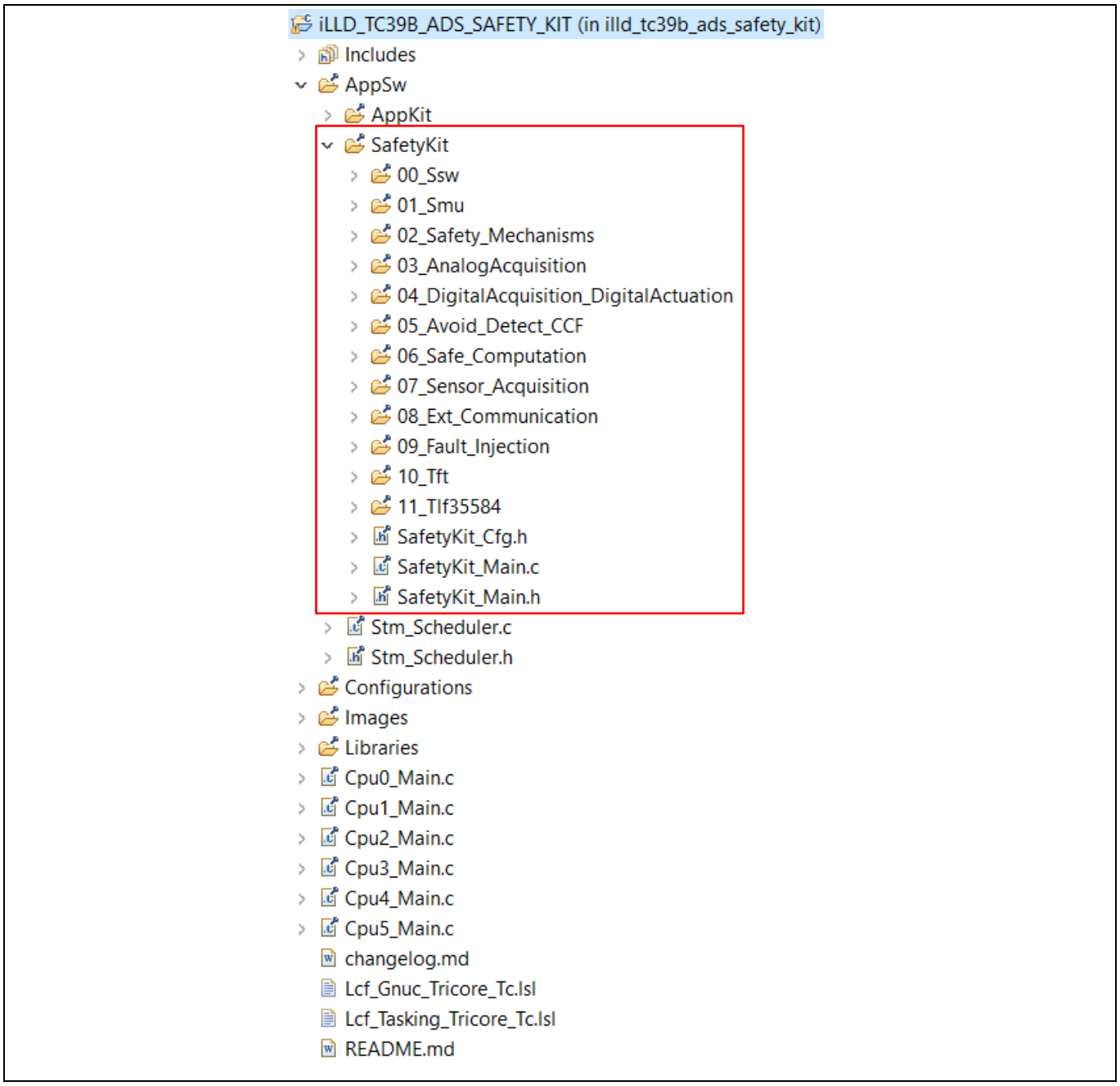


Figure 5 Project folder structure

2.4 Touch display interface presentation

The Application Kit - AURIX™ TC397 TFT features a touchscreen display, which is used to show the status information, real-time data, or to trigger different actions such as error injection. The default TFT view and an alarm pop-up window are shown in Figure 6. The alarm pop-up has four options to select:

- Reset alarm
- Ignore alarm
- Reset SMU
- Reset system

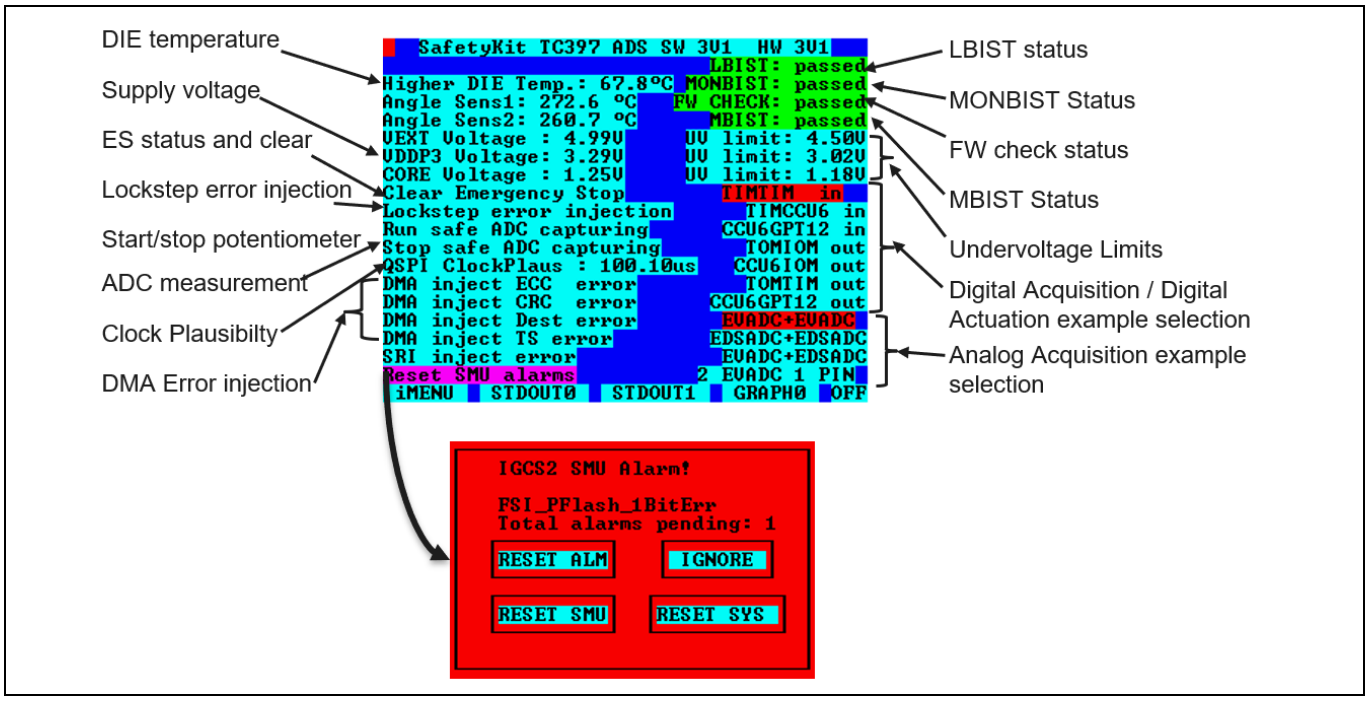


Figure 6 Demonstrator TFT display overview

The TFT display has some more menus available:

- SMU Test Result
- Switch off option
- Keyboard for updating voltage value

STDOUT1 and GRAPH0 are not used currently but keep there just for future or user use.

The Figure 7 below is an illustration of the above three options.

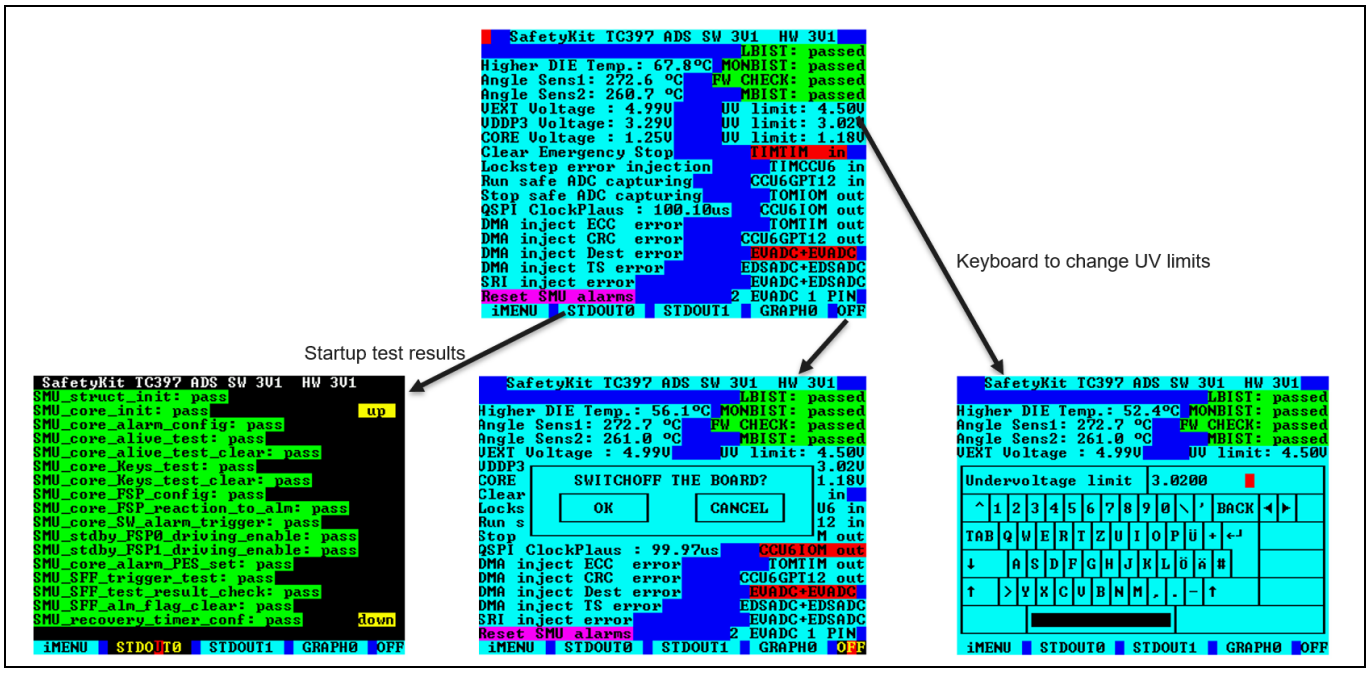


Figure 7 TFT Display menu options

2.5 ASCLIN shell interface

The AURIX™ Application Kit TC3xx Safety has another option as ASCLIN shell interface (terminal use). Currently, the following three options are available:

- **standby**: Switch TFL35584 to standby or TLF30682 to disabled state
- **Showtlf**: Show the status of the TLF35584 register
- **trigAlalrm**: Trigger a SMU alarm

To use the shell interface, power the Application Kit and connect it to the PC via a USB connector. Open a terminal in ADS (see Figure 8) and adjust the configuration as given below (user serial port can be different).

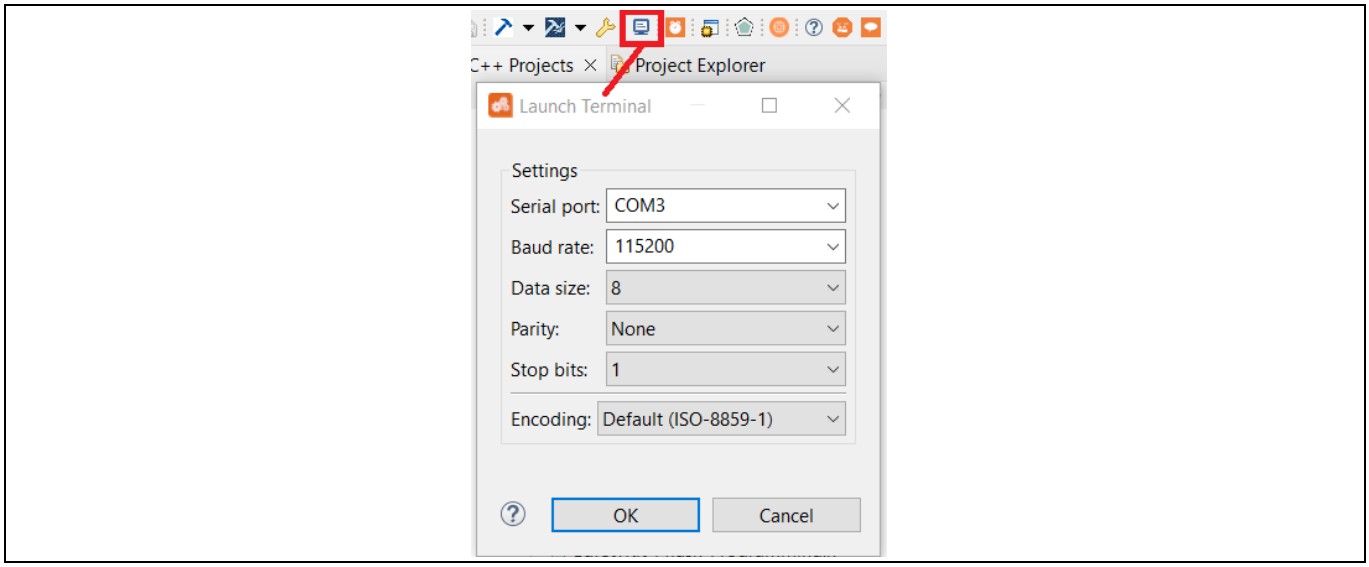


Figure 8 ASCLIN shell interface configuration

After a successful USB connection, the following messages will be displayed in the terminal:

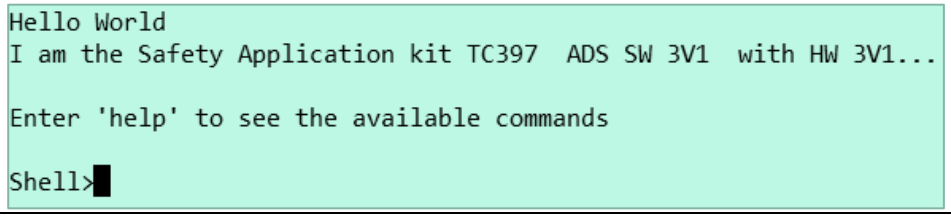


Figure 9 ASCLIN shell interface successful connection

Type “help” and press **Enter** to display the list of functionalities available via this interface. See Figure 10.

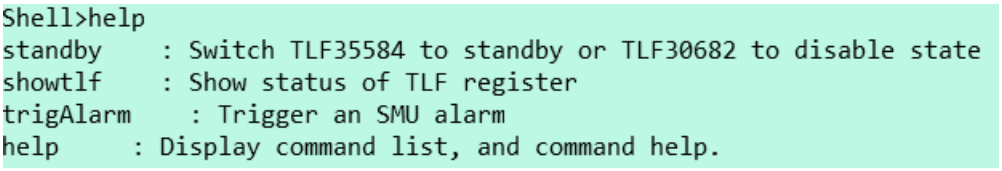


Figure 10 ASCLIN shell interface available commands

To get the syntax of a command type the command name and then type “?”(question mark) to list the syntax of the command:

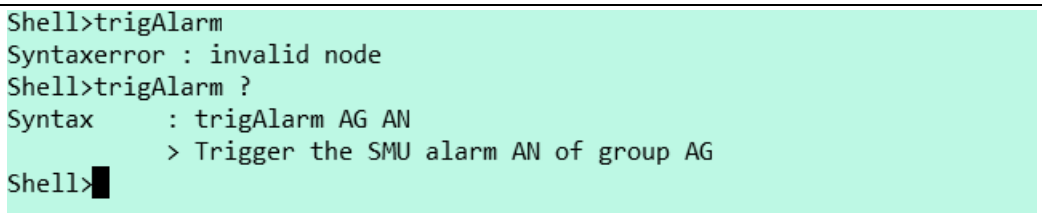


Figure 11 ASCLIN shell interface syntax correction

3 Boot and startup procedure

3.1 Analog power-up

When the device comes from an unpowered state and the external power supply reaches 2.4 V, the internal circuitry is activated, and the power management system (PMS) checks the 100 MHz backup clock source (fBACK). If the clock is stable, PBIST (*Safety Mechanism PBIST*) is automatically executed. The device is released from the cold PORST reset only if the PBIST is executed successfully.

3.2 Boot firmware

After PORST release, the system firmware (FW) execution is started by CPU0. The firmware (FW) is a code stored in Boot ROM that is automatically executed by the device after every reset. The FW is configurable via UCB registers and is composed of the following parts (among others):

Startup Software (SSW)

The internal SSW firmware contains procedures for device initialization. Depending on the type of reset the device is coming from, the startup software will execute different tasks. Some parts may already be initialized when coming from a reset other than cold PORST. The start-up software takes care of:

- Flash ramp-up
- Device Configuration
- RAM Initialization
- Selection and execution of Startup Modes
- LBIST execution
- Lockstep configuration
- Ending the startup software and starting the User Code

For more information about the firmware execution flow, see the figure “AURIX™ TC3xx Platform Firmware: main flow” in the AURIX™ TC3xx User’s Manual [1].

Checker Software (CHSW)

The checker software verifies that all safety-critical aspects of the startup software have been executed correctly and thus everything is prepared for the execution of the user code.

For more information about the CHSW execution flow, see “AURIX™ TC3xx Platform Checker Software Overview” in the AURIX™ TC3xx User’s Manual [1].

Bootstrap Loaders (BSL)

The BSL routines provide mechanisms to load a user program into the RAM of CPU0. This loaded code is started after exiting the Boot ROM. Therefore, after a successful execution of the firmware, the application software is started.

3.3 Application SW startup

During the execution of the application SW startup from a lockstep, the user code is responsible for the execution of several operations for ensuring that latent faults are not present, and for the correct initialization of the MCU before starting the runtime execution.

In particular, the application SW must:

- Execute the logic build-in self-test (LBIST) if not already performed during FW
- Evaluate the result of LBIST
- Configure, run, and check the result of MONBIST for ensuring the absence of latent faults in the secondary voltage monitors and standby SMU alarm path
- Execute MCU_FW_CHECK
- Verify the correct configuration settings installed by FW as described in MCU_STARTUP
- Test the functionality of the SMU core alive monitor by the *Safety Mechanism ALIVE_ALARM_TEST*
- Configure, run, and check the result of MBIST for ensuring the absence of faults in RAM
- Ensure to enable all SMU alarms relevant for the application. In particular, the user code must re-enable alarms that were disabled during the configuration procedure

Figure 12 shows the sequence of safety mechanisms involved from an unpowered state to a full operational state of (see the legend on the bottom right for the respective color information):

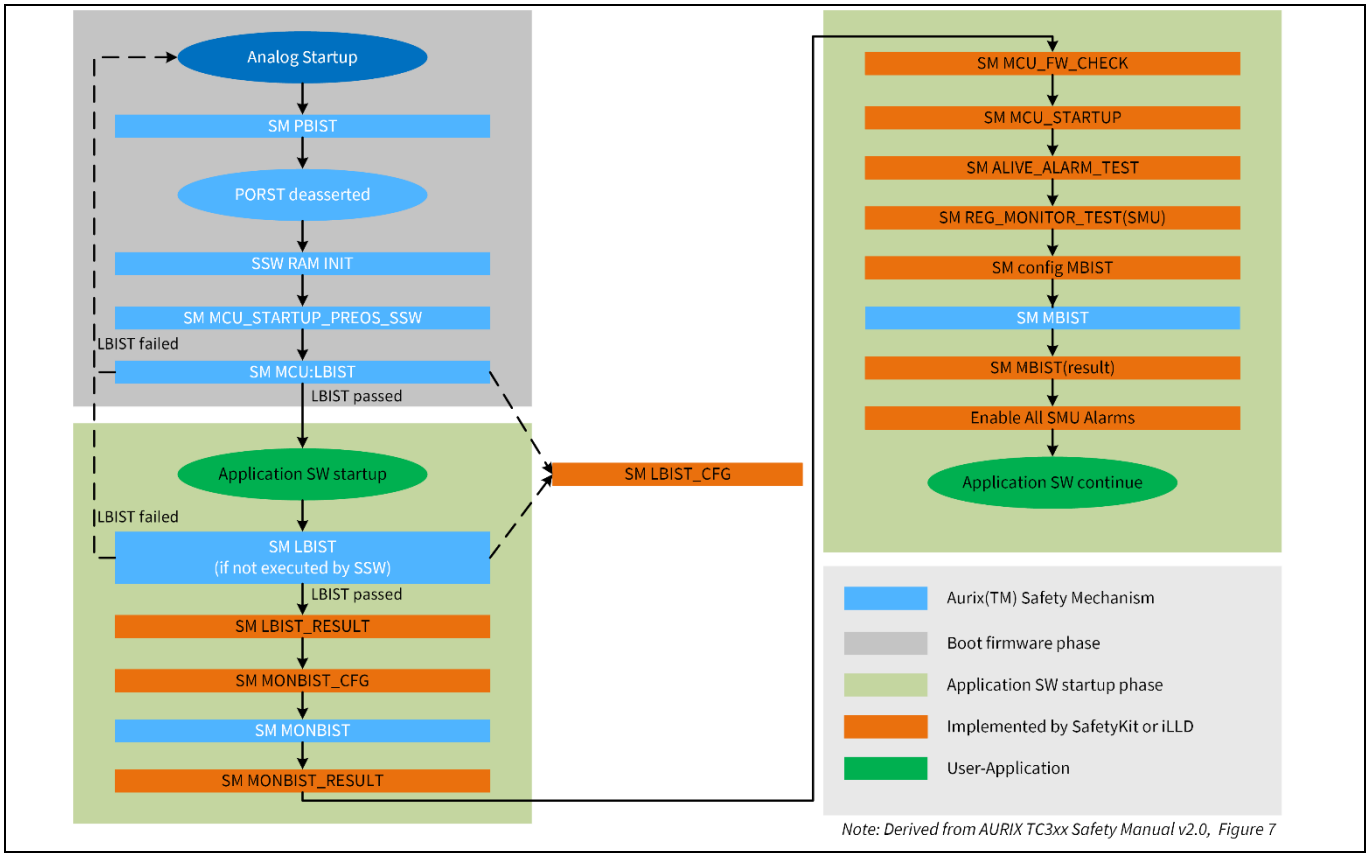


Figure 12 Safety mechanisms sequence during startup

Note: Depending on the application, additional safety mechanisms at startup are required. For example, if a non-lockstep CPU is used for safety-relevant tasks, all safety mechanisms of the non-lockstep CPU must be considered.

3.3.1 Safety Kit implementation of the application SW startup

The full application SW startup sequence can be observed in the following two code snippets. The first one shows the sequential execution of all steps involved; the corresponding macros to enable or disable individual steps can be observed in the second one.

Code Listing 1 Safe application software startup code example

```
/* Execute the sequence of SMs when coming from an un-powered state. */
void runSafeAppSwStartup(void)
{
#if SAFETYKIT_CFG_SSW_ENABLE_LBIST_BOOT || SAFETYKIT_CFG_SSW_ENABLE_LBIST_APPSW
    safetyKitSswLbist();
#endif

    /* Evaluate reset after LBIST execution */
    g_SafetyKitStatus.resetCode = safetyKitEvaluateReset();
    /* Evaluate if coming from standby mode */
    g_SafetyKitStatus.wakeupFromStandby = safetyKitEvaluateStandby();

#if SAFETYKIT_CFG_SSW_ENABLE_MONBIST
    g_SafetyKitStatus.sswStatus.monbistStatus = failed;

    /* MONBIST Tests and evaluation
     * SM:PMS:MONBIST_CFG and SM:MONBIST_RESULT */
    Ifx_Ssw_Monbist();
    g_SafetyKitStatus.sswStatus.monbistStatus = passed;
#endif /* SAFETYKIT_CFG_SSW_ENABLE_MONBIST */

#if SAFETYKIT_CFG_SSW_ENABLE_MCU_FW_CHECK
    /* SM:MCU_FW_CHECK */
    safetyKitSswMcuFwCheck();
#endif /* SAFETYKIT_CFG_SSW_ENABLE_MCU_FW_CHECK */

#if SAFETYKIT_CFG_SSW_ENABLE_MCU_STARTUP
    /* SM:MCU_STARTUP */
    safetyKitSswMcuStartup();
#endif /* SAFETYKIT_CFG_SSW_ENABLE_MCU_STARTUP */

#if SAFETYKIT_CFG_SSW_ENABLE_ALIVE_ALARM_TEST
    safetyKitSswAliveAlarmTest();
#endif /* SAFETYKIT_CFG_SSW_ENABLE_ALIVE_ALARM_TEST */

#if SAFETYKIT_CFG_SSW_ENABLE_REG_MONITOR_TEST
    /* SM:SMU:REG_MONITOR_TEST */
    safetyKitSswSmuRegMonitorTest();
#endif /* SAFETYKIT_CFG_SSW_ENABLE_REG_MONITOR_TEST */

#if SAFETYKIT_CFG_SSW_ENABLE_MBIST
    /* SM:MBIST */
    safetyKitSswMbist();
#endif /* SAFETYKIT_CFG_SSW_ENABLE_MBIST */
    /* Configure the alarm action for all SMU alarms with default configuration.
     * Some configuration might get overwritten with specific configuration later
in the function initSMUModule() */
    safetyKitEnableAllSMUAlarms();
}
```

\AppSw\SafetyKit\00_Ssw\SafetyKit_SSW.c

Code Listing 2 Macros to enable or disable steps of application SW startup

```
/*********************************************************************************/
/*-----------------------------------Macros--------------------------------------*/
/*********************************************************************************/
#define SAFETYKIT_CFG_SSW_ENABLE_LBIST_BOOT             1
#define SAFETYKIT_CFG_SSW_ENABLE_LBIST_APPSW            1
#define SAFETYKIT_CFG_SSW_ENABLE_MONBIST                1
#define SAFETYKIT_CFG_SSW_ENABLE_MCU_FW_CHECK           1
#define SAFETYKIT_CFG_SSW_ENABLE_MCU_STARTUP            1
#define SAFETYKIT_CFG_SSW_ENABLE_ALIVE_ALARM_TEST       1
#define SAFETYKIT_CFG_SSW_ENABLE_REG_MONITOR_TEST       1
#define SAFETYKIT_CFG_SSW_ENABLE_MBIST                  1

\AppSw\SafetyKit\SafetyKit_Cfg.h
```

3.3.2 LBIST

As already mentioned in the sections 3.2 Boot firmware and 3.3 Application SW startup, the LBIST can be either executed by hardware during the boot firmware or by the application software during the application SW startup. Both ways are supported by Application Kit Safety.

To execute the LBIST during the firmware, LBIST must be enabled via the LBIST enable bit (LBISTENA) in the boot mode index (BMI), see Code Listing 3.

As stated in Safety Mechanism LBIST_CFG, LBIST can be executed with different configurations. The only recommended configuration, which should be used is ‘Configuration A’, which is given in the device specific User’s Manual Appendix [2]. This configuration obtains the best coverage with an execution time less than 6 ms and reasonable power consumption. In fact, with the wrong configuration, LBIST execution can drain too much current that power monitoring will reset the device. To avoid any kind of misconfiguration of the LBIST, use the proper configuration in the User Configuration Block (UCB) UCB04.

The correct execution of the LBIST and the resulting signature is always verified by the application SW (*Safety Mechanism LBIST_RESULT*).

See Section 6.1.2 Logic built-in self-test (LBIST) for more information.

Code Listing 3 Eor disabling LBIST execution during boot firmware

```
const Ifx_Ssw_Bmhd bmhd_0_orig =
{
#if SAFETYKIT_CFG_SSW_ENABLE_LBIST_BOOT
    0x01FE,          /* 0x000: .bmi: Boot Mode Index (BMI)                        */
    0xB359,          /* 0x002: .bmhdid: Boot Mode Header ID (CODE) = B359H        */
    0xA0000000,      /* 0x004: .stad: User Code start address                     */
    0xFA2586D5,      /* 0x008: .crc: Check Result for the BMI Header (offset 00   */
    0x05DA792A,      /* 0x00C: .crcInv: Inverted Check Result for the BMI Header */
#else
    0x00FE,          /* 0x000: .bmi: Boot Mode Index (BMI)                        */
    0xB359,          /* 0x002: .bmhdid: Boot Mode Header ID (CODE) = B359H        */
    0xA0000000,      /* 0x004: .stad: User Code start address                     */
    0x31795570,      /* 0x008: .crc: Check Result for the BMI Header (offset 00   */
    0xCE86AA8F,      /* 0x00C: .crcInv: Inverted Check Result for the BMI Heade   */
#endif

[..]

const Ifx_Ssw_Bmhd bmhd_0_copy =
{
#if SAFETYKIT_CFG_SSW_ENABLE_LBIST_BOOT
    0x01FE,          /* 0x000: .bmi: Boot Mode Index (BMI)                       */
    0xB359,          /* 0x002: .bmhdid: Boot Mode Header ID (CODE) = B359H       */
    0xA0000000,      /* 0x004: .stad: User Code start address                    */
    0xFA2586D5,      /* 0x008: .crc: Check Result for the BMI Header (offset 00 */
    0x05DA792A,      /* 0x00C: .crcInv: Inverted Check Result for the BMI Heade */
#else
    0x00FE,          /* 0x000: .bmi: Boot Mode Index (BMI)                       */
    0xB359,          /* 0x002: .bmhdid: Boot Mode Header ID (CODE) = B359H       */
    0xA0000000,      /* 0x004: .stad: User Code start                            */
    0x31795570,      /* 0x008: .crc: Check Result for the BMI Header (offset 00 */
    0xCE86AA8F,      /* 0x00C: .crcInv: Inverted Check Result for the BMI Heade */
#endif

[..]
```

\Configurations\Ifx_Cfg_SswBmhd.c

Note: Check whether BMHD has been programmed successfully by verifying the value of the "bmi" variable of the "bmhd_0_orig" structure (bmi = 0x01FE if LBIST is enabled and bmi = 0xFE if LBIST is disabled).

3.3.3 MONBIST

See Section 6.1.3 Monitor built-in self-test (MONBIST).

3.3.4 Firmware check

The *Safety Mechanism MCU_FW_CHECK* is required for detecting failures (random hardware and systematic hardware and software), which may have affected the correct execution of the firmware.

In particular:

- Random hardware faults (RHF) can appear as transient or permanent faults and affect one of the hardware parts (CPU, buses, FLASH) that are needed by the firmware
- Systematic faults (hardware/software) can be generated by incorrectly programming a UCB and other registers, which control the firmware execution. Because the number of possible combinations cannot be tested during firmware development, the application needs to verify that firmware execution was correct.

Depending on the AURIX™ device and the kind of reset performed, the firmware execution varies. Therefore, the application software must verify the correct content of the registers and flags that are relevant for the specific type of reset. For the complete list of registers and SMU alarms expected content, see “Appendix A” of the AURIX™ TC3xx Safety Manual [3].

If one of the relevant registers or SMU flags do not report the expected value (“fail” condition), it can be concluded that the firmware execution has been corrupted by:

- **Firmware systematic fault:** Can be caused by a specific UCB register combination, leading to an incorrect firmware sequence and MCU initialization
- **Hardware transient fault:** Can be caused by EM interference or other transitory events affecting some hardware part (for example CPU0) during the firmware execution
- **Hardware permanent fault:** Can be caused by faults in the hardware (undetected by LBIST), affecting some hardware part (for example, CPU0) which lead to a firmware execution misbehavior

For firmware systematic faults and hardware permanent faults, the firmware check will fail after every reset, while hardware transient faults have an extremely low probability to affect the device in the same way again during a second attempt. Therefore, if the firmware check fails, it is recommended to perform a second attempt by triggering a device reset (ideally of the same type). If the check fails again, it should be concluded that the device is affected by a permanent hardware fault, and the device should be considered as not reliable.

3.3.4.1 FW_CHECK implementation

The implementation of *Safety Mechanism MCU_FW_CHECK* requires various steps. In this example, the main steps are as follows (see Code Listing 4, for the Application Kit Safety implementation of the firmware check):

- Check that the expected SMU alarms have been triggered (registers SMU_AG[0→11])
- Verification of the content of the SCU_STMEM[3→6] registers
- Verification of the content of the SCU_LCLCON[0→1] registers
- Verification of the content of the SSH registers (MCi_ECCD, MCi_FAULTSTS, and MCi_ERRINFO[0], with “i” representing every SRAM used in the application)

Attention: ***Register values depend on the AURIX™ device type and reset type. For the full list of registers and their expected values, see “Appendix A” of the AURIX™ TC3xx Safety Manual.***

If all registers and SMU alarms report the expected values, the application SW must:

- Clear the content of the registers mentioned in the tables in “Appendix A” of the Safety Manual [3]
- Clear the SMU alarms SMU_AG [0 ... 11]
- Clear the corresponding reset status bits in RSTSTAT register
- Proceed further

As already mentioned, if any of the registers do not report the expected values, it shall be assumed that the firmware has not been executed as expected. It is recommended to perform a second attempt by triggering a device reset (ideally of the same type).

STMEM check

The execution performed by the SSW is checked by the CHSW that indicates the status of several modules in these four registers at the end of each reset:

- SCU_STMEM3
- SCU_STMEM4
- SCU_STMEM5
- SCU_STMEM6

For more details, see Section 3.1.2.2 “Checks performed by CHSW and exit information” in the AURIX™ TC3xx User Manual. The expected values of these registers are available in Section 3.1 “Checker Software exit information for ALL CHECKS PASSED” of the device-specific User’s Manual Appendix [2].

LCLCON check

During SSW execution, lockstep cores are enabled according to the UCB_BMHD registers. After each reset (except an application reset), the application software must verify the status of the LCLCON[0→1] registers according to the enabled lockstep cores:

- SCU_LCLCON0
- SCU_LCLCON1

SSH register check

For each memory controller (MC) used in the application, this routine verifies that the alarm and error status registers are reporting the expected values. The values mainly depend on the reset type. The following registers are verified:

- ECCD
- FAULTSTS
- ERRINFO [0]

Code Listing 4 Code snippet of firmware check implementation

```
 * SM:MCU_FW_CHECK
  * */
void safetyKitSswMcuFwCheck(void)
{
    if(g_SafetyKitStatus.resetCode.resetType == safetyKitResetTypeColdpoweron)
    {
        /* Initialize the g_sswStatusXram data if it was a Cold PORST */
        g_sswStatusXram->mcuFwcheckRuns  = 0;
    }
[..]

    /* Increment the firmware check execution counter */
    g_sswStatusXram->mcuFwcheckRuns++;
    if  (
            /* Read SMU alarm register values and compare with expected ones(listed in Appendix A of the
             * Safety Manual)
             * Note: depending on the device and reset type different register values are expected */
            (TRUE == safetyKitFwCheckSmuStmemLclcon(fwCheckSMUTC39B, fwCheckSMUTC39BSize,
                    g_SafetyKitStatus.resetCode.resetType, fwCheckVerificationSMU))   &&
            /* Read SCU_STMEM register values and compare with expected ones(listed in Appendix A of the
             * Safety Manual) */
               (TRUE == safetyKitFwCheckSmuStmemLclcon(fwCheckSTMEMTC39B, fwCheckSTMEMTC39BSize,
                    g_SafetyKitStatus.resetCode.resetType, fwCheckVerificationSTMEM))  &&
            /* Read SCU_LCLCON register values and compare with expected ones (listed in Appendix A of the
             * Safety Manual) */
            (TRUE == safetyKitFwCheckSmuStmemLclcon(fwCheckLCLCONTC39B,  fwCheckLCLCONTC39BSize,
                    g_SafetyKitStatus.resetCode.resetType, fwCheckVerificationLCLCON)) &&
            /* Read SSH register values of all RAM and compare with expected ones (listed in Appendix A of
             * the Safety Manual)*/
            (IfxMtu_MbistSel_none == safetyKitFwCheckSsh(g_SafetyKitStatus.resetCode.resetType))
        )
    {
        /* If all four checks have passed set FW check status variable to "passed" */
        g_SafetyKitStatus.sswStatus.mcuFwcheckStatus = passed;

        /* If all registers and SMU alarm registers have reported the expected values .. */
        /* .. clear the content of the registers mentioned in the Appendix table */
        safetyKitFwCheckClearSSH(g_SafetyKitStatus.resetCode.resetType);
        /* .. clear the SMU alarms SMU_AG0..11 */
        safetyKitFwCheckClearSmuAlarms(fwCheckSMUTC39B, fwCheckSMUTC39BSize);
        /* .. clear the corresponding reset status bits in RSTSTAT register */
        IfxScuRcu_clearColdResetStatus();
    }
    else
    {
        g_SafetyKitStatus.sswStatus.mcuFwcheckStatus = failed;
        /* If FW check has failed during its first execution trigger the check again */
        if(g_sswStatusXram->mcuFwcheckRuns < SAFETKIT_FW_CHECK_MAX_RUNS)
        {
            /* Clear COLD PORST reason to preserve the data on the SCR XRAM */
            IfxScuRcu_clearColdResetStatus();
            safetyKitFwCheckRetriggerCheck(g_SafetyKitStatus.resetCode.resetType);
        }
    }
[..]

}
\AppSw\SafetyKit\00_Ssw\SafetyKit_SSW_02_MCU_FW_CHECK.c
```

Attention: ***Debugger influence***
Debuggers might have an (undefined) interference to the firmware behavior and alarms set during firmware execution. Therefore, it is mandatory to disconnect the debugger before performing any reset and reconnect it at the end of the firmware execution. Otherwise, the SMU alarms and/or SSH registers may not report expected values.

Attention: ***"Early" execution***
The firmware check (Safety Mechanism MCU_FW_CHECK) is recommended to perform its execution in the earliest stage possible of the startup. One reason for that is that SMU ALMs and/or SSH registers might be accidentally influenced by actions within the application software. Another reason is that, to meet the startup timing requirements, the application must be able to detect a fault in the firmware check and trigger a second reset (if required) as soon as possible.

3.3.4.2 Reset triggering

If one of the checks shown in the previous sections is not met, the application must reset the microcontroller at least once. As described in Section 9.1.2.2 "Reset Types" of the AURIX™ TC3xx User Manual [1], there are four kinds of resets that can be used in an application:

- Cold power-on reset (cold PORST)
- Warm power-on reset (warm PORST)
- System reset
- Application reset

There are several methods for generating a reset. In general, a PORST (cold or warm) requires the external power supply to either remove the power supply voltage (cold PORST) or assert the PORST pin (warm PORST). System and application resets can be generated by application software through the SWRSTCON and RSTCON registers, respectively. It is possible to store the number of reset events (except for cold PORST) in the Standby Controller Extension RAM, which retains its contents during all resets except cold PORST.

Cold reset through LBIST

If the external power supply is unable to generate a cold PORST by removing the supply voltage, it is possible to simulate a cold PORST by triggering a "fake" LBIST. This can be achieved by setting LBISTCTRL0.PATTERNS = 0 and triggering a new LBIST execution by software. The LBIST controller will start and immediately stop the LBIST scan chain. At the end of the LBIST execution, the MCU internally automatically triggers an internal reset, which from a firmware execution point of view is equivalent to a cold PORST.

Ensure the following before launching a new LBIST execution:

- Reset the LBIST controller to its initial state and set the "done" bit to zero again by setting LBISTCTRL0.LBISTRES = 1.
- Clear the cold reset status bits by setting LBISTCTRL2.CLRC = 1.

At the next startup, the application software can detect that an LBIST reset has been executed by checking RSTSTAT.STBYR = 0 and RSTSTAT.LBTERM = 1. The registers and SMU alarms will report the values expected for a cold reset. If one of the checks fails again (it does not necessarily have to be the same condition that failed at the cold PORST), the device must be considered faulty.

See Code Listing 5 for the Safety Application Kit implementation and Section 9.3.3 “LBIST Support” in the AURIX™ TC3xx User’s Manual [1] for more details.

Code Listing 5 Code snippet of function safetyKitTriggerLbist

```
/*
 * SM:LBIST_CFG
 * */
void safetyKitTriggerLbist(void)
{
    /* Increment counter variable which counts the LBIST requests via Application SW
*/
    g_sswStatusXram->lbistAppSwReq++;

    /* Clear COLD PORST reason to preserve the data on the SCR XRAM */
    IfxScuRcu_clearColdResetStatus();

    /* Trigger LBIST */
    if ( IFX_SCU_CHIPID_CHREV_TC39X_BD == MODULE_SCU.CHIPID.B.CHREV)
    {
        /* Default signature for TC39X-BD device */
        IfxScuLbist_triggerInline(&IfxScuLbist_defaultConfig_tc39x_bd);
    }
    else
    {
        IfxScuLbist_triggerInline(&IfxScuLbist_defaultConfig);
    }

    while(1)
    {
        __nop(); /* After triggering LBIST wait for warm reset */
    }
} \AppSw\SafetyKit\00_Ssw\SafetyKit_SSW_00_LBIST.c
```

Warm PORST

A warm PORST is generated by an assertion of the PORST pin while keeping the external power supply voltage stable. After a warm PORST, it is possible to configure the firmware to perform a RAM initialization (through the PROCONRAM register) and even to retrigger an LBIST execution (although this is not necessary). The standby domain remains functional during a warm PORST, so the standby RAM (Memory Controller instances 77 and 78) are not accessed during the consequent firmware execution.

System and application reset

System and application resets can be easily triggered by the application software during runtime by configuring the following registers. See Code Listing 6 for the Application Kit Safety implementation.

- Configure RSTCON.SW with the required reset (0x1 system, 0x2 application)
- Configure SWRSTCON.SWRSTREQ = 0x1

Code Listing 6 Code snippet of function safetyKitTriggerSwReset

```
/*
 * This function triggers either a SW Application Reset or a SW System Reset, based
on the parameter resetType
 * */
void safetyKitTriggerSwReset(SafetyKitResetType resetType)
{
    /* Get the CPU EndInit password */
    uint16 cpuEndinitPw = IfxScuWdt_getCpuWatchdogPassword();

    /* Configure the request trigger in the Reset Configuration Register */
    IfxScuRcu_configureResetRequestTrigger(IfxScuRcu_Trigger_sw, (IfxScuRcu_Reset-
Type)resetType);

    /* Clear CPU EndInit protection to write in the SWRSTCON register of SCU */
    IfxScuWdt_clearCpuEndinit(cpuEndinitPw);

    /* Trigger a software reset based on the configuration of RSTCON register */
    IfxCpu_triggerSwReset();

    /* The following instructions are not executed if a SW reset occurs */
    /* Set CPU EndInit protection */
    IfxScuWdt_setCpuEndinit(cpuEndinitPw);
} \AppSw\SafetyKit\00_Ssw\SafetyKit_SSW.c
```

3.3.5 MCU_STARTUP

Before entering run mode, the application SW must check for any corruption of data stored in the safety-relevant registers.

To verify whether the contents of these registers is correct, the user code must store a data checksum accumulated over all safety-relevant registers and application-dependent safety registers. When this calculation is completed, the application software must compare the calculated result with an expected checksum stored in the NVM.

See the implemented function `void safetyKitSswMcuStartup(void)` and to the *Safety Mechanism MCU_STARTUP* section in the Safety Manual [3] for more information.

3.3.6 SMU ALIVE_ALARM_TEST

If a malfunction occurs, the primary part of the SMU (SMU_core) generates an internal *alive* signal to the standby part of the SMU (SMU_stdby). The SMU Core Alive signal is generated if one of the following conditions is met:

- An alarm event occurs while SMU_core is in RUN or FAULT state and the SMU_core alive monitor detects that a reaction has not been generated by SMU_core.
- A watchdog or recovery timer alarm event occurs while the SMU_core is in START state and the SMU_core alive monitor detects that a reaction has not been generated by SMU_core.
- The application issues an SMU_ActivateFSP or SMU_ActivatePES command, but the appropriate reaction is not generated by SMU_core.
- The alarm configuration is changed while this alarm is being processed.

According to *Safety Mechanism ALIVE_ALARM_TEST*, the application software must, at least once per driving cycle, test the SMU core alive monitor and its connection to the SMU standby by triggering the alive alarm.

See the Application Kit Safety implementation (Code Listing 8) and Section 15.3.1.2.5 “Interface to SMU_stdby” in the User’s Manual [1] and *Safety Mechanism ALIVE_ALARM_TEST* in the Safety Manual [3] for more information.

Code Listing 7 Code snippet of function safety mechanism ALIVE_ALARM_TEST

```
/* SM:ALIVE_ALARM_TEST */
void safetyKitSswAliveAlarmTest(void)
{
    g_SafetyKitStatus.smuStatus.smuCoreAliveTestSts        = NA;
    g_SafetyKitStatus.smuStatus.smuCoreAliveTestClearSts   = NA;

    if((g_SafetyKitStatus.resetCode.resetType == safetyKitResetTypeColdpoweron) ||
       (g_SafetyKitStatus.resetCode.resetType == safetyKitResetTypeLbist))
    {
        /* Start SMU Alive Test */
        IfxSmu_startAliveTest();

        /* Poll for command status (success) */
        uint8 timeout = 0xFF;
        while (SMU_STS.B.RES != 0U && timeout > 0)
        {
            timeout--;
        }
        /* Wait for ALM21[16] - SMU Alive Monitor Alarm */
        while(IfxSmuStdby_getSmuStdbyAlarmStatus(21, 16) != IfxSmuStdby_AlarmStatusFlag_faultExist)
        {
            __nop();
        }
        /* Set smuCoreAliveTestSts SMU status variable which is displayed on TFT */
        g_SafetyKitStatus.smuStatus.smuCoreAliveTestSts = pass;
        /* Stop alive test */
        IfxSmu_stopAliveTest();
        /* Clear the ALM21[16] which is triggered by the Core alive test */
        IfxSmuStdby_setSmuStdbyAlarmStatusFlag(21, 16, IfxSmuStdby_AlarmStatusFlag_faultExist);

        /* Poll for command status (success) */
        timeout = 0xFF;
        while (SMU_STS.B.RES != 0U && timeout > 0)
        {
            timeout--;
        }
        /* Check if alarm is cleared */
        if(IfxSmuStdby_getSmuStdbyAlarmStatus(21, 16) == IfxSmuStdby_AlarmStatusFlag_noFaultExist)
        {
            g_SafetyKitStatus.sswStatus.aliveAlarmTestStatus = passed;
            /* Set smuCoreAliveTestClearSts SMU status variable which is displayed on TFT */
            g_SafetyKitStatus.smuStatus.smuCoreAliveTestClearSts = pass;
        }
        else
        {
            g_SafetyKitStatus.sswStatus.aliveAlarmTestStatus = failed;
            /* Set smuCoreAliveTestClearSts SMU status variable which is displayed on TFT */
            g_SafetyKitStatus.smuStatus.smuCoreAliveTestClearSts = fail;
        }
    }
} \AppSw\SafetyKit\00_Ssw\SafetyKit_SSW_04_ALIVE_ALARM_TEST.c
```

3.3.7 SMU REG_MONITOR_TEST

According to *Safety Mechanism REG_MONITOR_TEST*, the application software must execute the register monitor test of all relevant functional blocks by setting the related bit in the RMCTL register. The register monitor test should have a timeout greater than the value listed in the Safety Manual. At the end of the test, the application software must check the result by reading the RMEF register.

See the Application Kit Safety implementation (Code Listing 8) and *Safety Mechanism REG_MONITOR_TEST* in the Safety Manual [3] for more information.

Code Listing 8 Code snippet of Safety Mechanism REG_MONITOR_TEST

```
void safetyKitSswSmuRegMonitorTest(void) {
    [..]
/* Iterate through all modules listed in the SmuRegMonitorModule enumeration */
    for (SmuRegMonitorModule testModeEnable = smuRegMonitorModuleMTU; testModeEnable < numSmuRegMonitorModule;
         testModeEnable++)
    {
        /* Enable the test */
        IfxSmu_setRegMonTestModeEnable(testModeEnable);
        tStart = IfxStm_get(&MODULE_STM0);
        /* Wait until test is done, as long test is not finished measure the execution time */
        do{
            tExecution = IfxStm_get(&MODULE_STM0) - tStart;
        }
        while(!(IfxSmu_getRegisterMonitorStatus() & (1U << testModeEnable)));

        /* Disable the test */
        IfxSmu_clearRegMonTestModeEnable(testModeEnable);
        /* Convert the time to seconds */
        tExecutionInSec = tExecution / IfxStm_getFrequency(&MODULE_STM0);

        /* Validation */
        if(IfxSmu_getRegisterMonitorErrorFlag() && (1U << testModeEnable))
        {
            smuRegMonitorTestPassed = FALSE;
        }

        if(tExecutionInSec > SMU_REG_MONITOR_TEST_MAX_TIME_SEC)
        {
            smuRegMonitorTestPassed = FALSE;
        }
    }
    /* Set g_SafetyKitStatus.smuStatus variable */
    if(smuRegMonitorTestPassed == TRUE)
    {
        g_SafetyKitStatus.smuStatus.smuSafetyFlipFlopTriggerTestSts       = pass;
        g_SafetyKitStatus.smuStatus.smuSafetyFlipFlopTestResultCheckSts  = pass;
    }
    /* CLear all Safety Flip-Flop uncorrectable errors which are raised during the test */
    /* Required for g_SafetyKitStatus.smuStatus variable */
    boolean alarmClearedSuccessful = FALSE;
    SmuStatusType result = pass;

    for(uint8 errorId = 0; errorId < NUM_UNCORRECTABLE_ERRORS; errorId++)
    {
        if(safetyFfUncorrectableErrors[errorId].smuType == TYPE_SMU_CORE)
        {
            IfxSmu_clearAlarmStatus(safetyFfUncorrectableErrors[errorId].alarmName);
            if(IfxSmu_getAlarmStatus(safetyFfUncorrectableErrors[errorId].alarmName) == FALSE)
            {
                alarmClearedSuccessful = TRUE;
            }
        }
        else if (safetyFfUncorrectableErrors[errorId].smuType == TYPE_SMU_STDBY)
        {
            if (clearBitSMUstdby(safetyFfUncorrectableErrors[errorId].alarmName) == fail)
            {
                result = fail;
            }
        }
    }
[..]
    if(REG_MONITOR_TEST_PASSED)
    {
        g_SafetyKitStatus.sswStatus.regMonitorTestStatus = passed;
    }
    else
    {
        g_SafetyKitStatus.sswStatus.regMonitorTestStatus = failed;
    }
}
\AppSw\SafetyKit\00_Ssw\SafetyKit_SSW_05_SMU_REG_MONITOR_TEST.c
```

3.3.8 MBIST

See Section 6.1.4 for more information on memory built-in self-test (MBIST).

3.3.9 Enable all SMU alarms

The Safety Manual recommends enabling all SMU alarms relevant for the application. In particular, the user must re-enable all alarms that must be disabled during the startup configuration procedure.

See the function `void safetyKitEnableAllSMUAlarms(void)` in *\AppSw\SafetyKit\00_Ssw\SafetyKit_SSW.c*.

4 Failure management

Note: See AURIX™ TC3xx Safety Manual [3], Chapter 6 “Safety Mechanisms” and AURIX™ TC3xx User’s Manual [1], Chapter 15 “Safety Management Unit (SMU)”.

The cornerstone of AURIX™ failure management is the Safety Management Unit (SMU). The *Safety Mechanism SMU:CONFIG* is required to configure the SMU reaction for each possible alarm. The SMU is the central component of the safety architecture, providing a generic interface to manage the behavior of the microcontroller under the presence of faults. The SMU centralizes all the alarm signals related to different hardware and software-based safety mechanisms. The purpose of the SMU is to configure the behavior of each alarm. Those alarms can be individually configured to trigger internal actions and/or notify externally the presence of faults through a fault signaling protocol. The severity of each alarm must be configured according to the needs of the safety application(s): by default, every alarm reaction is disabled, apart from the watchdog timeout alarms.

The SMU is split in two parts:

- **SMU_core**: Located in the core domain; responds to all alarms generated by modules supplied by the SPB clock
- **SMU_stdby:** Located in the stand-by domain; responds to all alarms generated by modules supplied by the BACK clock

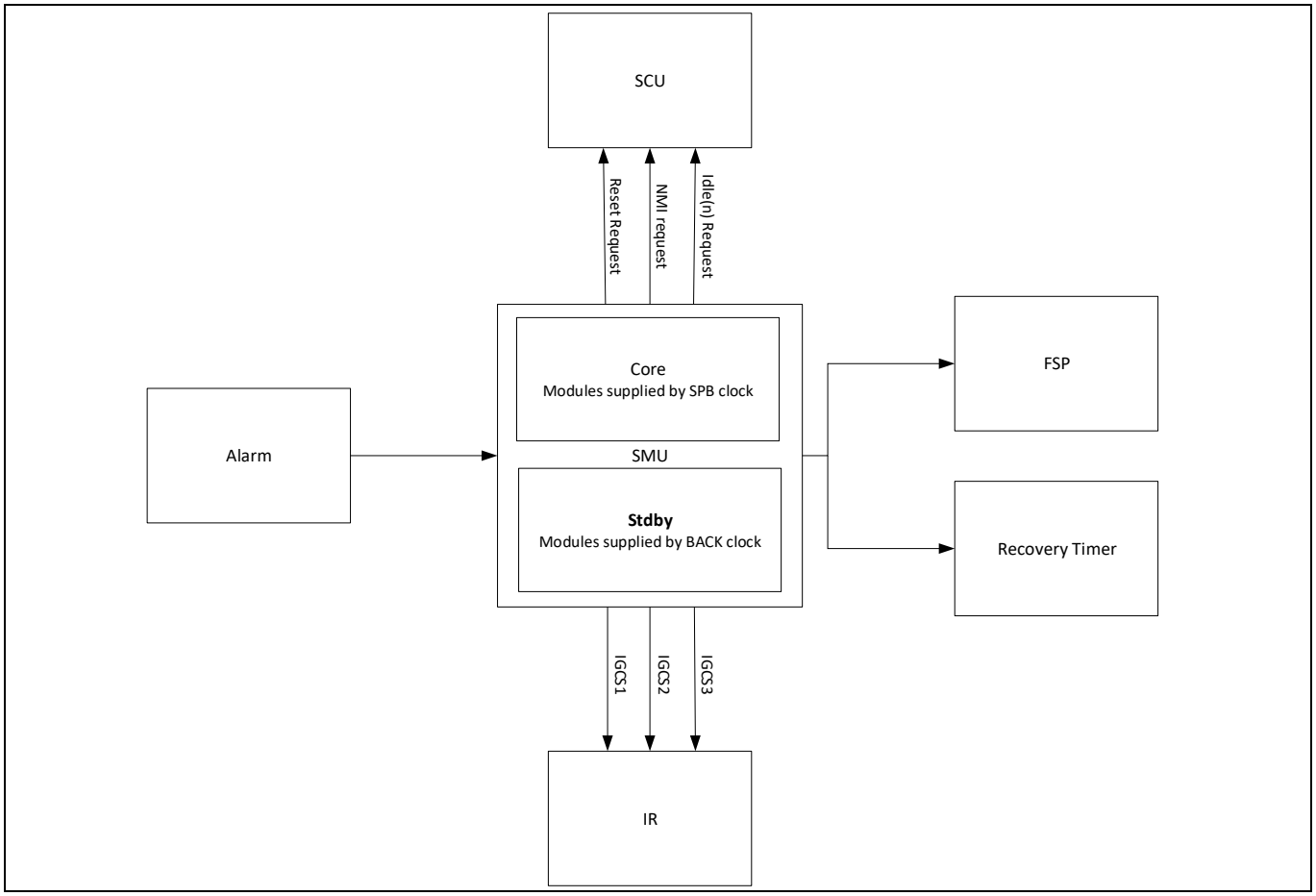


Figure 13 Simplified SMU concept

4.1 Error management concept

To configure the specific action to be taken, the alarm signals at the input of the SMU are mapped with the alarm configuration registers. There is a one-to-one relationship between an alarm group index ALM<n> signal and the alarm configuration and status registers (AG<n>). Each group is made of up to 32 input alarms.

Each alarm can be configured with a different “primary” behavior:

- **SMU interrupt service request:** IGCSx (x = 0 .. 2) with the AGC. IGCSx register controls how the SMU triggers interrupt requests to the interrupt router
- **Non-maskable interrupt (NMI):** High-priority interrupt, which cannot be masked (cannot be ignored by the system)
- CPU reset request
- Application or system reset request
- Disable the alarm

The list can be completed with a “second” behavior:

- **FSP:** See Section 4.2.2
- **Recovery timer:** See Section 4.2.1

4.2 SMU driver implementation

The `initSMUModule` function (see **Error! Reference source not found.**) contains a test of *Safety Mechanism SMU:LOCK* to ensure proper SMU functionality. Afterwards, structures are filled according to the configured alarm configuration (`globalAlarmConfig` structure) of each alarm. For alarm mapping, see the User’s Manual Appendix [2].

Code Listing 9 Fault injection to trigger FSP via SMU_STDBY

```
void initSMUModule(void)
{
    /* Prevention of double SMU reset */
    IfxScuWdt_clearSafetyEndinit(IfxScuWdt_getSafetyWatchdogPassword());
    SCU_WDTSCON1.B.CLRIRF = 1;
    IfxScuWdt_setSafetyEndinit(IfxScuWdt_getSafetyWatchdogPassword());

    /* Initialize the structure containing the execution state of SMU sensitive functions */
    initFunctionExecutionStatusSMU(&g_SafetyKitStatus.smuStatus);

    /* Test if SM:SMU:LOCK is working */
    enableKeysTestSMU();

    /* Reset alarmCounter */
    g_SafetyKitStatus.smuAlarmPending.alarmCounter = 0;

    /* Set result to fail */
    SmuStatusType result = fail;

    /* Fill the RuntimeAlarmHandle structures and configure alarms */
    result = initSMUAlarmsSMU();
    g_SafetyKitStatus.smuStatus.smuCoreAlarmConfigSts = result;

    /* Enable and configure the PES */
    uint8 pesAction = onPESIGCS1;

    g_SafetyKitStatus.unlockConfig &= IfxSmu_unlockConfigRegisters();
    if (g_SafetyKitStatus.unlockConfig == TRUE)
    {
        g_SafetyKitStatus.smuStatus.unlockConfigRegisterSMU = pass;
    }
    else
    {
        g_SafetyKitStatus.smuStatus.unlockConfigRegisterSMU = fail;
    }

    IfxSmu_configAlarmActionPES(pesAction);
    IfxSmu_temporaryLockConfigRegisters();

    /* Validation if PES configuration was successful */
    IfxScuWdt_clearSafetyEndinitInline(IfxScuWdt_getSafetyWatchdogPasswordInline());
    if(MODULE_SMU.AGC.B.PES != pesAction){
        result = fail;
    }
    IfxScuWdt_setSafetyEndinitInline(IfxScuWdt_getSafetyWatchdogPasswordInline());
    g_SafetyKitStatus.smuStatus.smuCoreAlarmPESSetSts = result;

    /* Enable and configure the recovery Timer (maximum value for the duration is 0xffffff) */
    enableRecoveryTimerSMU(0xffffff);

    /* Enable and configure the FSP */
    enableFSPcoreSMU(IfxSmu_FspMode_TimeSwitchingProtocol, IfxSmu_FspPrescalar1_referenceClockDiv2,
            IfxSmu_FspPrescalar2_referenceClockDiv4096);

    enableFSPstdbySMU();

    /* Enable the SMU */
    result = activateSMU();
    g_SafetyKitStatus.smuStatus.smuCoreInitSts = result;

    /* Check if IGCSx group config is valid and if SMU ISRs are reachable */
    result = checkIsrConfigSMU();
    g_SafetyKitStatus.smuStatus.smuCoreAlarmConfigSts &= result;

} \AppSw\SafetyKit\01_Smu\SMU\SMU.c
```

4.2.1 Recovery Timer (RT) and watchdog alarms

Note: See the section "Safety Mechanism SMU:RT" in the AURIX™ TC3xx Safety Manual [3], Section 15.3.1.5.7 "Recovery timer" and Section 15.3.1.5.8 "Watchdog alarms" in the AURIX™ TC3xx User's Manual [1].

The SMU implements two recovery timers (RT0 and RT1) to monitor the response time of a RESET, NMI, or interrupt triggered by an alarm. If the RT is not serviced before it times out, an alarm is generated.

Two independent instances (RT0 and RT1) are available. The recovery timer duration (identical for all instances) is configured in the SMU_RTC register. It is possible to enable or disable each instance; however, both instances are enabled by default because recovery timers are required for the operation of the CPU watchdog timers.

The alarm mapping consists of a pair of parameters {GIDi, ALIDi} (with i = 0..3), where GIDi is a group identifier and ALIDi is the alarm identifier belonging to the group. Four {GIDi, ALIDi} pairs can be configured per recovery timer instance.

If a recovery timer is enabled and for any of the {GIDi, ALIDi} pairs an alarm event occurs and if an internal action is configured leading to an internal action (the alarm status must be cleared), the recovery timer is automatically started by hardware.

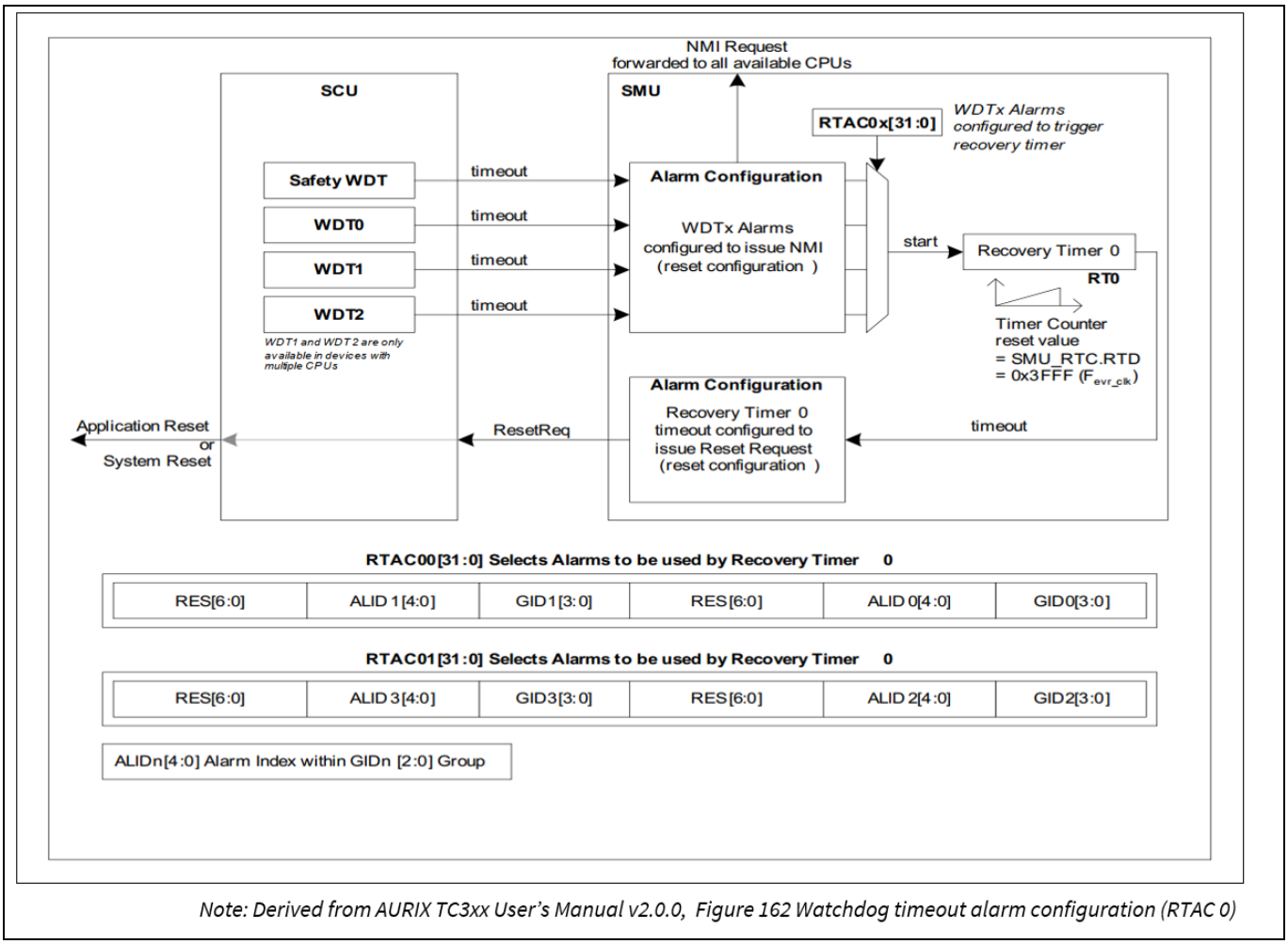
Once a recovery timer event has occurred, the recovery timer starts and counts until the software stops it with `IfxSmu_stopRT()`. If the timer expires, an internal SMU alarm (Recovery Timer Timeout) is issued.

Watchdog alarm processing

The safety manual states that a special processing including the recovery timer is required to ensure the correct microcontroller behavior if the watchdog timers (WDT) are not serviced by software or firmware. It must be ensured that the microcontroller is reset after a pre-warning phase, where the software can still perform some critical actions:

- Every timeout alarm must activate an NMI.
- Recovery Timer 0 must be configured to service WDT timeout alarms for Safety WDT, CPU0 WDT, CPU1 WDT and CPU2 WDT.
- Recovery Timer 1 must be configured to service WDT timeout alarms for CPU3 WDT, CPU4 WDT and CPU5 WDT.
- Recovery Timer 0 and Recovery Timer 1 timeout alarms must be configured to issue a reset request and activate the fault signaling protocol.

Figure 14 shows an example for RTAC 0 and the related four WDT (Safety WDT, CPU0 WDT, CPU1 WDT and CPU2 WDT). As shown, if any of the four mentioned watchdog timers has on overflow (timeout), an SMU alarm will be raised which is configured to issue an NMI. The SMU is also configured to automatically start Recovery Timer 0 if an NMI is triggered; therefore, Recovery Timer 0 starts incrementing directly after one of the watchdog timeouts. This pre-warning phase enables the application to react to the fault. If the system can solve the issue and to stop Recovery Timer 0, a Recovery Timer overflow is prevented; therefore, a reset request to the SCU (application or system reset) which would have occurred after the overflow is also avoided.



Note: Derived from AURIX TC3xx User's Manual v2.0.0, Figure 162 Watchdog timeout alarm configuration (RTAC 0)

Figure 14 Watchdog timeout alarm configuration (RTAC 0)

4.2.2 Fault Signaling Protocol (FSP)

The Fault Signaling Protocol (FSP) enables the microcontroller to report a critical situation to an external safety controller device to control the safe state of the safety system.

The fault signaling protocol is configured through the FSP register and has three possible states (STS register):

- Power-on reset
- Fault-free (FSP not asserted)
- Fault (alarm asserting FSP)

FSP can be configured with three possible protocols, which define the FSP behavior for fault-free and fault state:

- Bi-stable protocol (default)
- Dynamic dual-rail protocol
- Time-switching protocol (recommended protocol for the usage of AURIX™ MCUs together with the TLF35584 PMIC)

As shown in Code Listing 28, all four lockstep comparator alarms are enabled to trigger the FSP (“enable external reaction” column). It is also possible to configure SMU_stdby alarms to trigger the FSP (see the `alarmStdbySMUReactionEnabledFSP` structure).

Most safety applications will reset the system after an SMU alarm has occurred. For the Application Kit Safety application, many alarms will clear the alarm and resume. This FAULT to RUN state transition is possible only if SMU_AGC.EFRST is enabled (see the iLLD function `IfxSmu_enableFaultToRunState` which is called inside the `configSMUFSPcore` function).

After an alarm occurs, the Application Kit Safety application clears the alarm and releases FSP in the `coreAlarmReactionClearSMU` function *(SMU.c).*

SMU_stdby alarms must be handled differently compared to SMU_core alarms. If an SMU_stdby alarm event has occurred, it can be cleared by writing to the SMU_stdby registers rather than the core SMU_AGx registers. See the example in Code Listing 10.

Code Listing 10 Fault injection to trigger FSP via SMU_STDBY

```
/*
 * Test the FSP for the stdby alarm
 * */
void testFSPstdbySMU()
{
    IfxScuWdt_clearSafetyEndinitInline (IfxScuWdt_getSafetyWatchdogPasswordInline ());
    /*
    * An over-voltage event is triggered when the threshold is crossed in
    * a lower to higher voltage transition. Greater than or equal compare
    * is used.
    */
    PMS_EVRMONCTRL.B.EVRCOVMOD = 0x1;

    /* Value to trigger the fault */
    PMS_EVROVMON.B.EVRCOVVAL = 0x00;

    IfxScuWdt_setSafetyEndinitInline (IfxScuWdt_getSafetyWatchdogPasswordInline ());

    /* Clear SMU_stdby ALM for VDD over voltage fault */
    IfxScuWdt_clearSafetyEndinitInline (IfxScuWdt_getSafetyWatchdogPasswordInline ());

    /* Default value when the micro controller is reset via Power on reset */
    PMS_EVROVMON.B.EVRCOVVAL = 0xFE;

    /* Preparing for bit clearing, with ASCE */
    PMS_CMD_STDBY.U |= 0x40000008;

    /* Clear the status flag error related to over voltage */
    PMS_AG20_STDBY.B.SF4 = 1;

    IfxScuWdt_setSafetyEndinitInline (IfxScuWdt_getSafetyWatchdogPasswordInline ());
} \AppSw\SafetyKit\01_Smu\SMU\FSP\SMU_stdby_FSP.c
```

Note: Write access to PMS_EVRMONCTRL.EVRCOVMOD is Safety Endinit protected.

Note: See the section “Safety Mechanism SMU:FSP_MONITOR” in the AURIX™ TC3xx Safety Manual [3] and Section 15.3.1.8 “Fault Signaling Protocol (FSP)” in the AURIX™ TC3xx User’s Manual [1].

4.2.3 Port Emergency Stop (PES)

This feature provides a fast reaction without the intervention of SW. The selected output ports are immediately set-switched to the input function if an alarm occurs. The Port Emergency Stop request to the SCU can be activated by any of the following situations:

- Port Emergency Stop configured for the port via Pn_ESR – see TC3xx User Manual, Section 14.4.13 “Emergency Stop Register”
- Activate PES
- An alarm event with SMU_AG<x>FSP enabled and FSP.PES enabled
- An alarm event with an internal action configured in SMU_AG<x>CFx registers and SMU_AGC.PES enabled for that action

The PES feature can be configured for an internal action via the iLLD function `IfxSmu_configAlarmActionPES`. See Section 6.2.8.4.

5 System-level hardware requirements

The Application Kit - TC3xx Safety utilizes the Application Kit TC397 TFT as a mainboard for running the safety demonstration. This board already implements the system-level hardware elements required when developing a safe application (see Section 3.3 “System Level Hardware Requirements” in the Safety Manual [3]).

The system-level hardware requirements are focused on:

- External voltage supply
- Error monitor
- External time-window watchdog

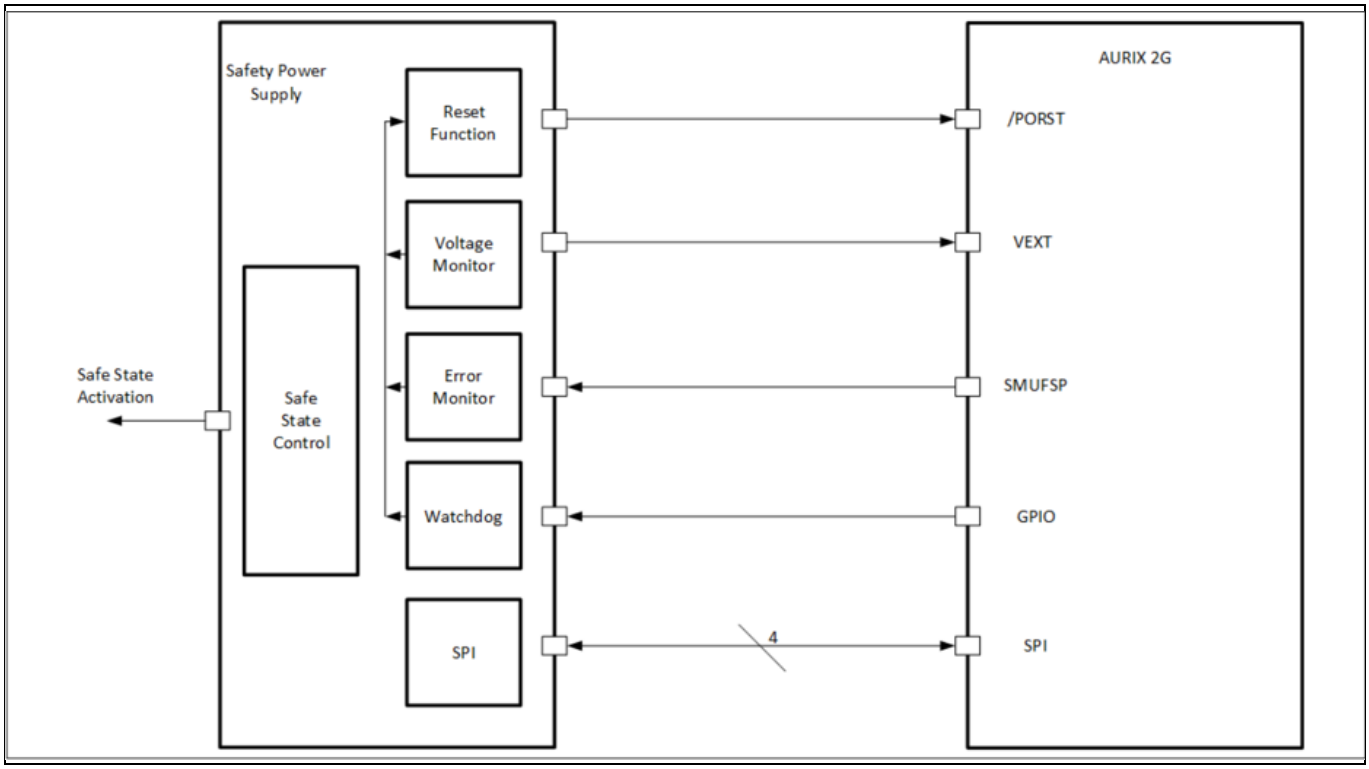


Figure 15 System-level hardware requirements overview

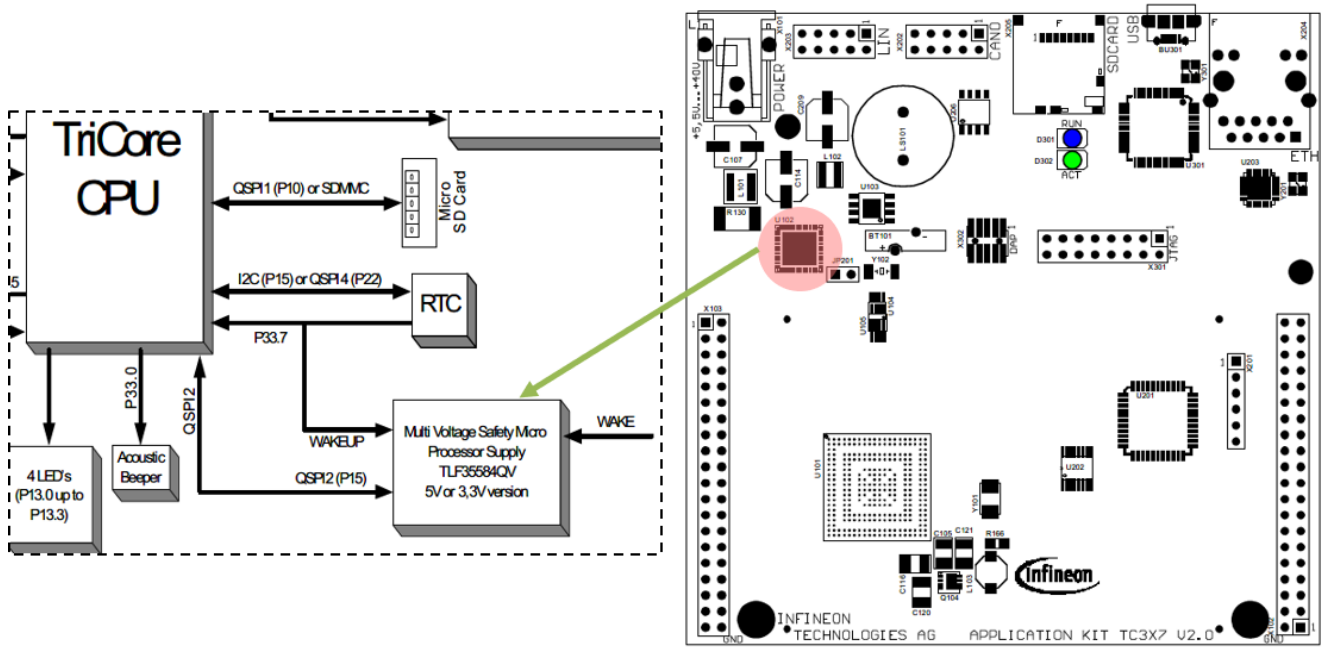
All the system-level hardware requirements are fulfilled by one external component, the Infineon TLF35584QV multi-voltage safety microprocessor supply, which was already introduced in Section 2.1.1.

5.1 External voltage supply

In order to operate safely, AURIX™ MCUs should be powered with a stable voltage supply (see *Safety Mechanism PMS:VEXT_VEVRSB_ABS_RATINGS* in the Safety Manual). TLF35584QV features a low-drop post regulator 3.3 V/600 mA or 5.0 V/600 mA for MCU supply. An external device, which can be different from the supply voltage device, should monitor the supply voltage and disable it in case of a limit violation (see “*Safety Mechanism PMS:VEXT_VEVRSB_OVERVOLTAGE”* in the Safety Manual). TLF35584 includes an independent voltage-monitoring function of all output voltages for overvoltage and undervoltage conditions. An overvoltage condition detected for a predefined time will trigger the shutdown of the related regulator. Every overvoltage and undervoltage event is stored in a SPI status register.

If the MCU is exposed to a voltage level exceeding the operating conditions for longer than the specified time, a permanent fault can occur. Those faults would be detected by startup self-tests such as MONBIST or LBIST, or even a failure to start up.

Note: See the latest TLF35584 datasheet for more information [4].



Note: Derived from Application Kit Manual TC3X7 v2.1, Figure 2-1 Application Kit TC3X7 Block Schematic and Figure 2-2 Application Kit TC3X7 V2.0 Top Placement

Figure 16 TLF35584 power supply on the Application Kit TC397 TFT

Electromagnetic noise or other sources of disturbances can degrade the quality of the supply voltage (see *Safety Mechanism PMS:VX_FILTER*). TLF35584 does not detect such events. Decoupling capacitors must be installed near the AURIX™ MCU to avoid effects from short voltage spikes and high-frequency oscillations. See the application notes for BGA and TQFP/LQFP packages on PCB design and layout that list recommended capacitor values and placement.

The Application Kit TC397 schematic provides the following example (Figure 17):

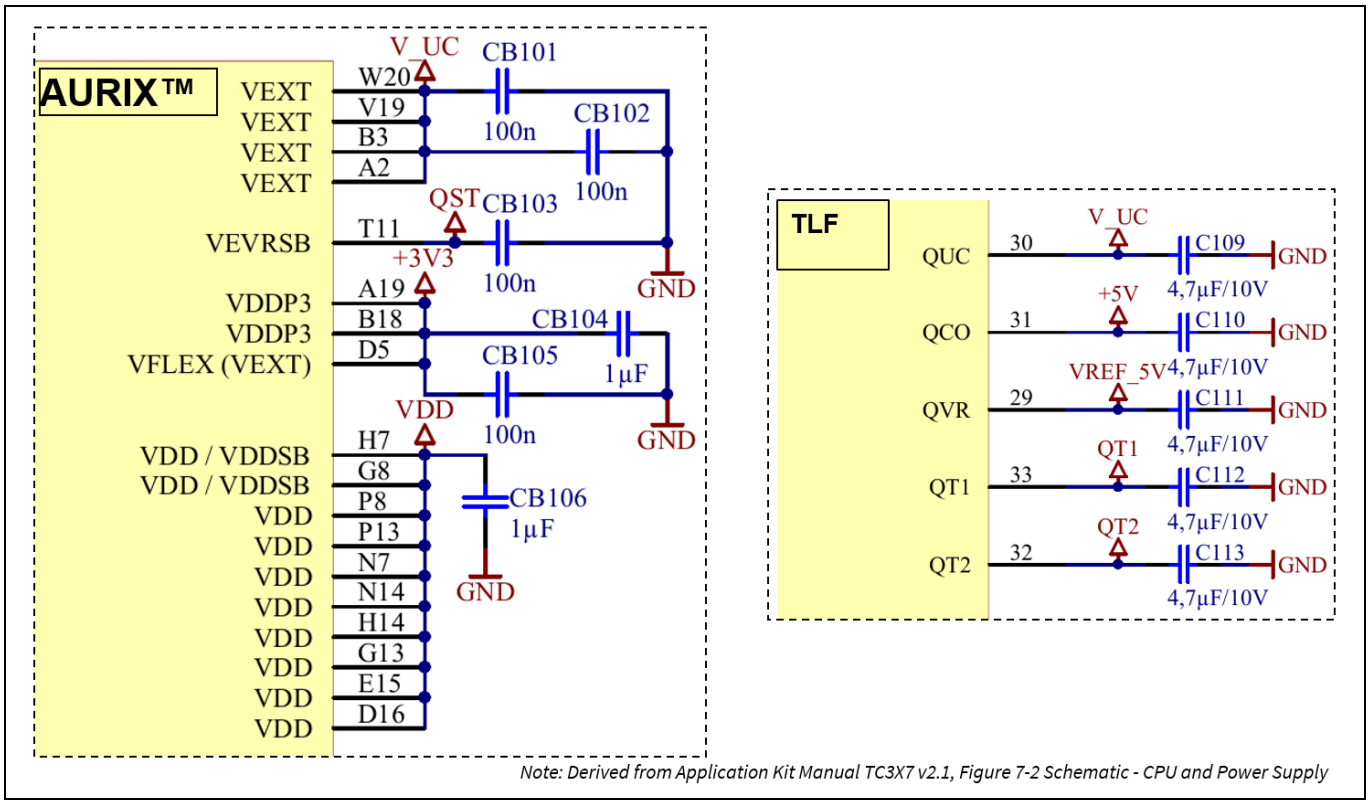


Figure 17 Decoupling capacitors for supply voltage filtering

5.2 Error monitoring

Some critical internal errors should be reported externally to create an external response such as warm reset. There are three ways of signaling an error:

- Fault signaling protocol (FSP) pin activation
- Emergency Stop
- Application software notification via NMI or ISR

5.2.1 FSP activation

The Fault Signaling Protocol (FSP) feature can be used to report a critical situation to an external component (*Safety Mechanism FSP_ERROR_PIN_MONITOR*). On the Application Kit - AURIX™ TC397 TFT, the external device used to monitor the AURIX™ MCU is the TLF35584 PMIC. In particular, the FSP pin P33.8 (SMU_FSP0) is used as the TLF35584 ERR input. It is also routed to the Evaluation Board - AURIX™ TC3xx Safety (add-on shield board) for error injection (see Figure 18).

The SMU can be configured to trigger an FSP reaction on a per-alarm basis. At the register level, this is done by setting the SMU_AGxFSP bit field corresponding to the alarm. At a higher level, using the provided SMU driver, it can be done by enabling the external reaction of the specific alarms inside the `globalAlarmConfig` array.

As already mentioned in Section 4.2.2, in Application Kit Safety, all lockstep comparator errors are configured to trigger the FSP (see Code Listing 28). For more information, see the section "*Safety Mechanism FSP_ERROR_PIN_MONITOR"* in the AURIX™ TC3xx Safety Manual [3].

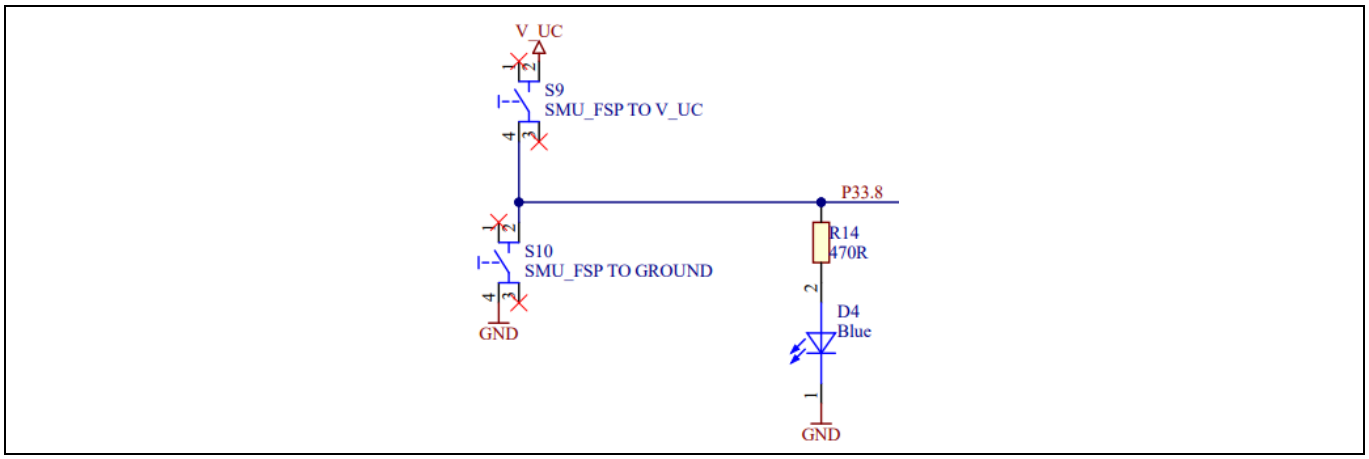


Figure 18 Circuitry for FSP signal fault injection

Note: By default, TLF35584 does not react to any fault. This is because the microcontroller programming support pin is asserted by hardware, putting TLF35584 in a debug mode and disabling the switch to the FAILSAFE mode. This allows the user to debug the system without worrying about safety features (e.g., error pin monitoring or watchdogs). By removing the resistor R127 on the Application Kit TC397 TFT PCB and by assembling a 2-pin jumper on JP201, debug mode can be enabled and disabled in a more flexible way. For more information, see the Application Kit TC397 manual [5] and the TLF35584 datasheet [4].

5.2.2 Emergency Stop activation

The Emergency Stop (ES) feature can also be used to report a critical situation to an external component (*Safety Mechanism ES_ERROR_PIN_MONITOR*). Any combination of the FSP pin activation, ES pin activation, NMI request, and ISR request can be used depending on the system environment.

As FSP features are already used for monitoring by an external device, the ES feature is not implemented to report a critical situation to an external component; therefore, it is also not implemented on Application Kit Safety. The system integrator is responsible to determine the **_ERROR_PIN_MONITORING* safety mechanism to implement.

5.2.3 Application software notification via NMI or ISR

The third way to signal an erroneous behavior is to use the software-based external failure reporting interface (*Safety Mechanism SW_ERROR_PIN_MONITOR*). It can be done through any kind of reporting interface such as ES activation, FSP pin activation, NMI request, or Interrupt Service Request (ISR).

5.3 External time-window watchdog

An external device with an independent reference clock and with the functionality of a time-window watchdog supervises the AURIX™ microcontroller. This external device must initiate the transition to the system safe state if a fault, which can lead to the violation of a system safety goal is detected. This is necessary to control common-cause failures initiated by external stress conditions or internal failures of the microcontroller, which may lead to a state where the microcontroller cannot signal an internal failure.

Some systems implement external watchdogs supporting program flow monitoring with a question-answer protocol. Although these functions can be taken over by the internal watchdogs so there is no need for such a

device; complex external watchdogs are still supported by the AURIX™ microcontroller. In this case, a separate window watchdog may not be required.

For more information, see the "*Safety Mechanism WATCHDOG_FUNCTION*" section in the Safety Manual [3].

Note: *By default, TLF35584 does not react to any fault. This is because the microcontroller programming support pin is asserted by hardware, putting TLF35584 in a debug mode and disabling the switch to the FAILSAFE mode. This allows the user to debug the system without worrying about safety features (e.g., error pin monitoring or watchdogs). By removing the resistor R127 on the Evaluation Board TC397 PCB and by assembling a 2-pin jumper on JP201, debug mode can be enabled and disabled in a more flexible way. For more information, see the Evaluation Board Manual [5] and the TLF35584 datasheet [4].*

6 Architecture for management of faults

6.1 Self-tests for latent fault metric support

The AURIX™ TC3xx platform supports automatic and user-triggered built-in self-test. The following section describes the available built-in self-test in AURIX™ TC3xx microcontroller.

6.1.1 Power built-in self-test (PBIST)

The power BIST (*Safety Mechanism PBIST*) is implemented in the hardware to identify power-related faults: in particular, faults related to voltage monitoring mechanisms. This SM is automatically executed before PORST, and releases the AURIX™ MCU from reset only after the test succeeds.

PBIST has no possible configuration from the user.

6.1.2 Logic built-in self-test (LBIST)

The LBIST module oversees the testing of the digital logic of the MCU. The logic BIST (LBIST) executes structural tests (the tests use scan chains) and checks the digital logic of the MCU to detect hard errors. The LBIST is used as a safety mechanism (*Safety Mechanism LBIST*) and contributes to achieving the latent fault metric (LFM) target.

The idea behind the LBIST is that, for a given chain of flip-flops (FF) with some combinational logic in between, if an input sequence is shifted through the chain, the chain should be in a predictable state when the input sequence is fully loaded. That state can then be captured in a single cycle. The relationship between the input and the captured state of the chain can be computed into an expected fixed signature. In case of an unexpected signature, the presence of a fault can be deduced.

For a flip-flop chain of length n, 2^n possible combinations should be tested to be sure to detect any latent fault. Unfortunately, this would take an unreasonable time, considering the number of flip-flops. To cope with that issue, randomized patterns are used. Multiple patterns can be tested to maximize the chance of finding a fault. The completed logic circuitry to be tested is divided into scan chains. Each scan chain is loaded with the pattern. The result of each scan chain is compacted into a single signature.

The software integrator is responsible for checking the proper execution of the LBIST and for checking the generated signature against the expected signature. The signature values can be found in the variant-specific User Manual Appendix.

As the complete circuitry of the chip is filled with a random pattern, a reset is required to bring the chip back to the default state. A warm PORST will be automatically generated at the end of the LBIST execution. The AURIX™ internal firmware follows the path for a cold power-on reset.

All digital modules in AURIX™ TC3xx, except for the PMS sub-block, are covered by LBIST scan chains. All CPUs, peripherals, ports (excluding pads), and the HSM are tested by the LBIST. PMS modules (EVR, SCR WUT, SMU_STDBY. etc.) are not covered by the LBIST. The PMS register interface is covered. Non-covered modules can be used to store the number of LBIST executions in case of an error (e.g., the standby controller extension RAM (XRAM)).

All safety mechanisms (for example: SRAM and PFlash ECC, MPUs, bus SMs, and all SMU alarms and the SMU_CORE) are covered through the LBIST. Therefore, there is no need to test the safety mechanisms again (for example by software) after a proper LBIST execution. Analog and mixed signal modules are not covered by

LBIST. Simply repeating the LBIST (for example with the same or different seed) does not improve the coverage.

LBIST does not test the SRAMs, PFlash, or DFlash memories. The RAMs have to be tested separately by the MBIST. However, SRAM redundancy registers are part of the scan chain and therefore corrupted. Therefore, SRAMs contents are not reliable after the LBIST and must be initialized after the LBIST before use. By default, SRAMs are initialized after the cold power-on reset followed by the internal firmware, unless changed in HF_PROCONRAM.RAMIN. Similarly, LBIST does not test the flash itself, but the surrounding/digital logic (for example, FSI or ECC blocks) is tested.

The execution of the LBIST should be monitored as stated by *Safety Mechanism LBIST_MONITOR*. A normal LBIST completes within 6 ms. Because the MCU digital logic is not available during LBIST execution, an external device is responsible for making sure that the LBIST completes. An external WDT can be used for this purpose, as highlighted in *Safety Mechanism WATCHDOG_FUNCTION*. After PORST deassertion, the external power supply waits for a first SPI message from the MCU, which will serve the external WDT. If the wait time exceeds the allowed time window because of a problem during the startup sequence, the appropriate reaction must be taken at the system level.

The ESR0 and ESR1 pins are put into weak pull-down state during the LBIST execution state. Therefore, these pins can used as an alternative option to check the LBIST progress.

6.1.3 Monitor built-in self-test (MONBIST)

The secondary monitor BIST (*Safety Mechanism MONBIST*) is a user-triggered test that provides a higher latent fault coverage for secondary monitors, associated alarms, and error pin fault logic routed to the standby SMU. The test should be triggered at startup immediately after enabling the SMU.

As mentioned in *Safety Mechanism MONBIST_CFG* in the AURIX™ TC3xx Safety Manual [3] and in the corresponding section in the User Manual [1], the test procedure requires 16 steps. After the procedure has finished, the application software must check the MONBIST results in MONBISTSTAT (*Safety Mechanism MONBIST_RESULT*). That procedure is fully covered by the Infineon Low Level Drivers (iLLDs), either as a function as part of the application SSW (`void Ifx_Ssw_Monbist(void)`) or as part of the SMU_stdby (`void IfxSmuStdby_startSmuStdbyMonBist(void)`).

As shown in Code Listing 1 Safe application software startup code example, the function of the application SSW is used within this application note.

6.1.4 Memory built-in self-test (MBIST)

As mentioned in the “Safety Mechanism MBIST” section in the Safety Manual [3], the MBIST checks the SRAM integrity. The SRAM areas are not covered by the LBIST. To increase the latent fault detection coverage, this memory check is necessary.

The full non-destructive test (NDT) described in Section 13.3.8.1 “Non-Destructive Test (NDT)” of the User’s Manual [1] is fully implemented in the iLLD function `IIfxMtu_runMbistAll(mbistGangConfig)`.

As also stated in the “Safety Mechanism MBIST” section of the Safety Manual [3], the application software must prevent any attempt of read or write operations on the memory during the execution. The respective preparation steps to prevent these read or write attempts are shown in Code Listing 11.

Note: During an MBIST, the memories being tested are not accessible. To prevent unintended memory accesses, bus masters like CPU1…CPU5, etc. are kept disabled during the test. Because this MBIST example is executed on CPU0, it is required that CPU0.DSPR and CPU0. PSPR should not be accessible during the test; otherwise, the MBIST will fail. Therefore, during this test, no RAMs of CPU0 are accessed because the function and struct etc. are stored in PFlash and the local variables are handled by the core registers. The number of local variables used should not exceed the core registers, because any violation will be stored on the stack and cause the MBIST to fail (e.g., using local vector also or struct definitions). Therefore, it is the responsibility of the user to ensure adherence to this constraint while performing MBIST.

Code Listing 11 Code snippet of function safetyKitSswMbist

```
/* SM:VMT:MBIST */
void safetyKitSswMbist(void)
{
    g_SafetyKitStatus.sswStatus.mbistStatus = notEvaluated;
    /* Preparations for MBIST */
    /* Disable all other CPUs, but of course not the one which is executing the function */
    IfxCpu_ResourceCpu coreIndex  = IfxCpu_getCoreIndex();
    if(coreIndex != IfxCpu_ResourceCpu_0){
        IfxCpu_setCoreMode(&MODULE_CPU0, IfxCpu_CoreMode_idle);
    }
#if (IFXCPU_NUM_MODULES > 1)
    if(coreIndex != IfxCpu_ResourceCpu_1){
        IfxCpu_setCoreMode(&MODULE_CPU1, IfxCpu_CoreMode_idle);
    }
#endif
#if (IFXCPU_NUM_MODULES > 2)
    if(coreIndex != IfxCpu_ResourceCpu_2){
        IfxCpu_setCoreMode(&MODULE_CPU2, IfxCpu_CoreMode_idle);
    }
#if (IFXCPU_NUM_MODULES > 3)
#endif
    if(coreIndex != IfxCpu_ResourceCpu_3){
        IfxCpu_setCoreMode(&MODULE_CPU3, IfxCpu_CoreMode_idle);
    }
#if (IFXCPU_NUM_MODULES > 4)
#endif
    if(coreIndex != IfxCpu_ResourceCpu_4){
        IfxCpu_setCoreMode(&MODULE_CPU4, IfxCpu_CoreMode_idle);
    }
#if (IFXCPU_NUM_MODULES > 5)
#endif
    if(coreIndex != IfxCpu_ResourceCpu_5){
        IfxCpu_setCoreMode(&MODULE_CPU5, IfxCpu_CoreMode_idle);
    }
#endif

    /* Disable CPU caches */
    IfxCpu_setDataCache    (0);
    IfxCpu_setProgramCache (0);

    /* If DMA master is enabled disable it */
    boolean dmaWasEnabled = FALSE;
    if(MODULE_DMA.CLC.B.DISS == 0)
    {
        dmaWasEnabled = TRUE;
        uint16 passwd = IfxScuWdt_getCpuWatchdogPassword();
        IfxScuWdt_clearCpuEndinit(passwd);
        MODULE_DMA.CLC.B.DISR = 1;
        IfxScuWdt_setCpuEndinit(passwd);
    }

    /* MBIST Tests and evaluation */
    boolean nBistError = TRUE;
    nBistError = IfxMtu_runMbistAll(mbistGangConfig);

    /* check if there was any error */
    if (nBistError == FALSE)
    {
        g_SafetyKitStatus.sswStatus.mbistStatus = passed;
    }
    else
    {
        g_SafetyKitStatus.sswStatus.mbistStatus = failed;
    }

    /* Clear all ECCD and FAULTSTS registers of the tested memory */
    safetyKitClearMbistSshRegisters();

    /* Enable DMA module again if it got disabled before */
    if(dmaWasEnabled)
    {
        uint16 passwd = IfxScuWdt_getCpuWatchdogPassword();
        IfxScuWdt_clearCpuEndinit(passwd);
        MODULE_DMA.CLC.B.DISR = 0;
        IfxScuWdt_setCpuEndinit(passwd);
    }
[…]
} \AppSw\SafetyKit\00_Ssw\SafetyKit_SSW_06_MBIST.c
```

6.2 Functional blocks and safety-related functions

6.2.1 MCU function - processing

6.2.1.1 CPU

The TC3xx family utilizes the TriCore™ 1.62P (TC1.6P) core hardware, which is based on TC1.6P core with enhancements in memory distribution and protection and other aspects. Additionally, depending on the variant, up to four CPUs are protected by a lockstep mechanism, which allows to run up to ASIL-D applications. From a functional perspective, the two CPU categories (lockstep and non-lockstep) offer the same performance.

For more information, see the “CPU” section in the Safety Manual [3].

6.2.1.1.1 CPU memory and time protection

The CPU offers several hardware measures for protection on memory and resource accessed, as well as timer-based mechanisms for detecting timing violations of software:

- Safety mechanisms based on master TAG ID
- Safety mechanisms based on access privilege levels
- Safety mechanisms based on ENDINIT and SAFETY_ENDINIT
- Safety mechanisms based on program memory and data memory regions
- Safety mechanism to detect unintended interrupt request from the HW elements

For more information on these safety mechanisms, see the “Coexistence of HW/SW elements” chapter in the AURIX™ Safety Manual [3].

6.2.1.1.2 Lockstep CPU

AURIX™ TC3xx offers up to four lockstep CPUs; these lockstep CPUs are an important part of supporting safety. Lockstep is an optional safety mechanism that can be enabled during startup (see BMI.LSENAx in the AURIX™ TC3xx User Manual [1]).

A lockstep CPU is composed of a master core and a checker core. To detect a transient fault on the CPU (for example, due to charged particle interference), the checker core executes the same instructions as the master core. The checker core cannot be utilized for purposes other than performing the same tasks as the master core. The same code is executed by both the master core and the checker core but for the checker core, it is delayed by two clock cycle to reduce common-cause failures. In addition, the checker core is physically positioned in a different silicon area from the master core to avoid common-cause failures related to the layout. The result of the master and checker core is compared after the realignment. If the comparison fails, an alarm is raised to the SMU.

The logic covered by this hardware redundancy does not only cover the CPU, but also the associated logic. These are the logic areas duplicated for each master core and checker core:

- TriCore™ TC1.62P core
- CPU SFR and CSFR
- Master and slave interfaces to SRI
- Master interface to SPB

- Interface to the interrupt router
- Interface to the SCU
- Interface to the program memories (PMI)
- Interfaces to the data memories (DMI)
- Interfaces to the program flash (PFI)

Figure 19 shows an overview of the architectural elements of a lockstep CPU.

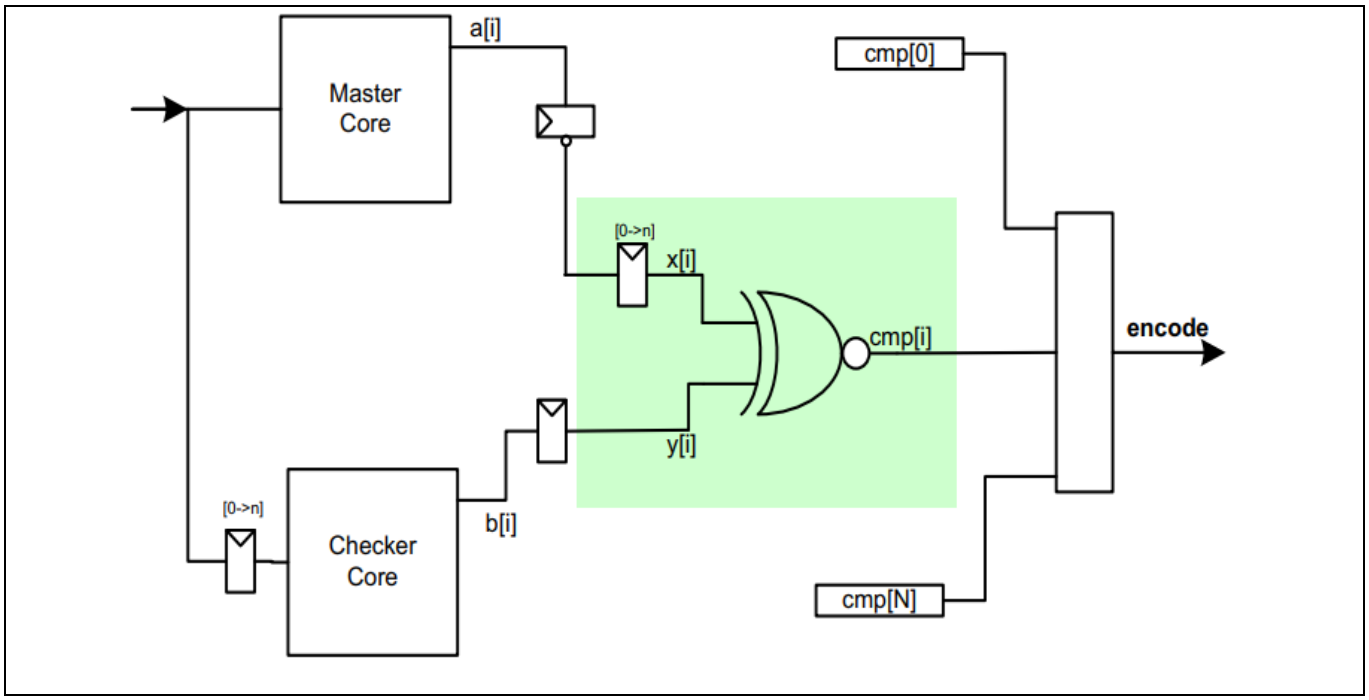


Figure 19 Simplified overview of lockstep architecture

Fault coverage and hardware self-test

Latent faults are covered by the LBIST, while transient faults in the comparison logic can be detected by the continuously running background self-test of the lockstep comparator. This self-test is executed every 8192 clock cycles on the master core, and then on the checker core. The complete self-test cycles are therefore repeated every 16,384 clock cycles. If there is no discrepancy, the self-test will not signal an alarm to the SMU.

Safety Kit implementation of lockstep error injection

To ensure that the lockstep mechanism is working correctly and that an alarm will be transmitted correctly to the SMU, lockstep provides a feature to manually inject faults in the comparator logic. This self-test is implemented in Application Kit Safety to demonstrate the lockstep safety mechanism. This feature can be started by the application software by writing '1' to the LCLT1 bitfield of the LCLTEST register. This in turn reports a lockstep comparator error alarm to the SMU. Because the master and checker cores are covered by the LBIST and the background comparator self-test, the fault injection test is not required in typical applications: it is provided here as a means of demonstrating lockstep error detection.

By selecting "Lockstep error injection" on the TFT main menu, a lockstep fault will be injected in CPU1. The fault will be only injected if no alarm is pending and if the lockstep is enabled. The result is a triggering of SMU alarm, which is shown on the TFT display (see Figure 20).

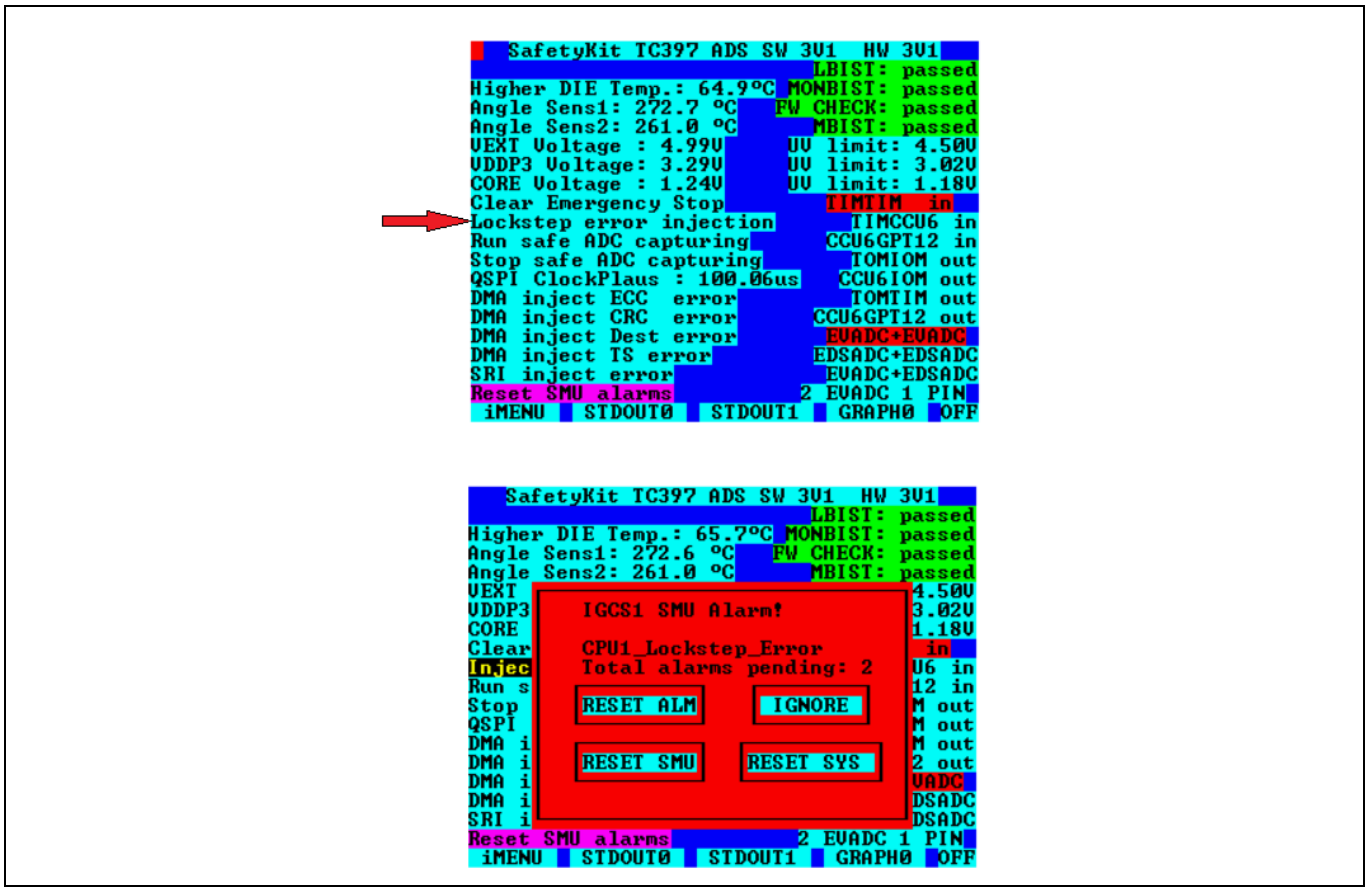


Figure 20 CPU1 lockstep error injection via TFT

Note: Two SMU alarms are pending because the SMU is configured to trigger both the lockstep alarm as well as the Emergency Stop on a lockstep error event. See sections 4.2.3 and 5.2.1 for more details.

Code Listing 12 Lockstep error injection example code

```
void injectLockstepError(IfxSmu_Alarm lockstepAlarm)
{
    /* Step 1: Check if alarm is pending */
    if(IfxSmu_getAlarmStatus(lockstepAlarm) == TRUE)
    {
        conio_ascii_printfxy (DISPLAYSTDOUT1, 1, 2, (uint8 *)"Lockstep error pending");
        return;
    }

    boolean errorInjected = FALSE;

    switch(lockstepAlarm)
    {
        case IfxSmu_Alarm_CPU0_Lockstep_ComparatorError:
            /* Step 2: Check if lockstep is enabled */
            if(SCU_LCLCON0.B.LS0 == 1){
                /* Step 3: Inject lockstep error */
                SCU_LCLTEST.B.LCLT0 = 1;
                errorInjected = TRUE;
            }
            break;
        case IfxSmu_Alarm_CPU1_Lockstep_ComparatorError:
            if(SCU_LCLCON1.B.LS1 == 1){
                SCU_LCLTEST.B.LCLT1 = 1;
                errorInjected = TRUE;
            }
            break;
        case IfxSmu_Alarm_CPU2_Lockstep_ComparatorError:
            if(SCU_LCLCON0.B.LS2 == 1){
                SCU_LCLTEST.B.LCLT2 = 1;
                errorInjected = TRUE;
            }
            break;
        case IfxSmu_Alarm_CPU3_Lockstep_ComparatorError:
            if(SCU_LCLCON1.B.LS3 == 1){
                SCU_LCLTEST.B.LCLT3 = 1;
                errorInjected = TRUE;
            }
            break;

        default:
            __debug();
    }

    if(errorInjected)
    {
        conio_ascii_printfxy (DISPLAYSTDOUT1, 1, 2, (uint8 *)"Lockstep error injected");
    }
    else
    {
        conio_ascii_printfxy (DISPLAYSTDOUT1, 1, 2, (uint8 *)"Lockstep not running");
    }
}
```

\AppSw\SafetyKit\04_Fault_Injection\SafetyKit_Lockstep.c

6.2.1.1.3 Non-lockstep CPU

The only architectural difference between a non-lockstep CPU and a CPU with lockstep is the missing redundant CPU (called the “checker core”). While there is no performance difference between both, the non-lockstep CPUs do not have dedicated mechanisms for fault detection. Therefore, software safety mechanisms may be needed. Infineon’s AUTOSAR MCAL package offers a software-based self-test (SM[SW]:CPU:SBST) to support applications on non-lockstep CPUs.

6.2.1.1.4 CPU RAM

Each CPU utilizes different RAM blocks as local memories, which can be affected by transient or permanent faults. Therefore, the CPU RAM has the same safety mechanisms common to all SRAM blocks, as described in

the “MCU Function – Volatile Memory” section of the AURIX™ TC3xx Safety Manual [3] or in Section 6.2.3.3 of this document.

6.2.1.2 Processing – FCE

The Flexible CRC Engine (FCE) is a hardware unit connected as a slave to the System Peripheral Bus (SPB) that provide an acceleration engine for CRC algorithms. For more information, see the “FCE” section in the AURIX™ TC3xx Safety Manual [3].

In Application Kit Safety, the FCE module is used for CRC calculation for a different data integrity purpose and therefore it contains implementation of the *Safety Mechanism CRC_CFG*, as shown in Code Listing 13.

Code Listing 13 FCE initialization

```
/* SM:CRC_CFG */
void initFCECRC()
{
    /* Disable interrupts */
    boolean interruptState = IfxCpu_disableInterrupts();

    /* Create FCE module configuration */
    IfxFce_Crc_Config fceConfig;
    IfxFce_Crc_initModuleConfig(&fceConfig, &MODULE_FCE);

    /* ISR priorities and interrupt target */
    fceConfig.isrPriority = ISR_PRIORITY_FCE_ER;
    fceConfig.isrTypeOfService = (IfxSrc_Tos)IfxCpu_getCoreIndex();

    /* Initialize module */
    IfxFce_Crc_initModule(&g_fceCrc.fce, &fceConfig);

    /* Initialize CRC kernel with default configuration */
    IfxFce_Crc_CrcConfig crcConfig;
    IfxFce_Crc_initCrcConfig(&crcConfig, &g_fceCrc.fce);

    /* For All CRC calculations */
    /* Enable Interrupt in case of CRC Mismatch */
    crcConfig.enabledInterrupts.crcMismatch = TRUE;
    crcConfig.useDma = TRUE;

    /* Initialize FCE CRC */
    crcConfig.crcKernel = IfxFce_CrcKernel_0;
    crcConfig.fceChannelId = IfxDma_ChannelId_20;
    IfxFce_Crc_initCrc(&g_fceCrc.fceCrc, &crcConfig);

    /* Enable interrupts again */
    IfxCpu_restoreInterrupts(interruptState);
} \AppSw\SafetyKit\06_Safe_Computation\SafetyKit_Fce.c
```

6.2.1.3 Processing - system timer (STM)

The STM is a free-running 64-bit timer that is enabled immediately after application reset and can be read by the application SW. Each CPU has a dedicated STM. The STM can be configured to generate compare-match ISR by using dedicated registers.

Monitoring concept

Permanent and transient hardware faults in the STM may corrupt the timer value or interrupt generation. Because the STM is not a part of the duplication area of the CPU, the application software is responsible for executing plausibility checks using an independent timer (see *Safety Mechanism STM:MONITOR* in the AURIX™ TC3xx Safety Manual [3]).

Safety Kit implementation of the safety mechanism STM:MONITOR

The *Safety Mechanism STM:MONITOR* states that the plausibility check should be implemented using a secondary timing module. Because there is one dedicated system timer per CPU, one STM can be used to monitor the other, and vice-versa. In this example, it is implemented in such a way that every CPU should call the `runStmMonitoring` function periodically. By calling this function, the value of the STM timer of the calling CPU and the value of the STM timer of another CPU will be compared. For example, CPU0 validates the STM timer value of CPU0 and CPU1, while CPU1 compares the values of STM1 and STM2, and so on.

If the deviation of two STM timer values is higher than two milliseconds, an appropriate reaction must be triggered (e.g., application reset).

Code Listing 14 Safety Mechanism STM:MONITOR code example

```
void runStmMonitoring(IfxCpu_ResourceCpu cpuIndex)
{
    Ifx_STM *stmX;
    Ifx_STM *stmY;

    switch (cpuIndex) {
            case IfxCpu_ResourceCpu_0:
                stmX = &MODULE_STM0;
                stmY = &MODULE_STM1;
                break;
            case IfxCpu_ResourceCpu_1:
                stmX = &MODULE_STM1;
                stmY = &MODULE_STM2;
                break;
            case IfxCpu_ResourceCpu_2:
                stmX = &MODULE_STM2;
                stmY = &MODULE_STM3;
                break;
            case IfxCpu_ResourceCpu_3:
                stmX = &MODULE_STM3;
                stmY = &MODULE_STM4;
                break;
            case IfxCpu_ResourceCpu_4:
                stmX = &MODULE_STM4;
                stmY = &MODULE_STM5;
                break;
            case IfxCpu_ResourceCpu_5:
                stmX = &MODULE_STM5;
                stmY = &MODULE_STM0;
                break;
            default: while (1) {}; break;
    }

    uint8 repeatNtimes = 5;
    while(repeatNtimes)
    {
        /* Compare tick counter of STM module X and STM module Y, raise alarm if deviation is greater
than 2 milliseconds */
        /* Get STM counter values */
        uint64 stmXvalue = IfxStm_get(stmX);
        uint64 stmYvalue = IfxStm_get(stmY);

        uint64 stmDifference = stmXvalue > stmYvalue ? stmXvalue - stmYvalue : stmYvalue - stmXvalue;

        if(stmDifference > (2*IFX_CFG_STM_TICKS_PER_MS) )
        {
            /* Try N times, if difference is still too high trigger appropriate reaction. */
            repeatNtimes--;
            if(repeatNtimes == 0)
            {

/* Safety manual recommends an application reset but it is disabled for Safety Kit demonstration */
#if (STM_APP_RESET == 0)
                /* Trigger appropriate reaction */
                softwareCoreAlarmTriggerSMU(SOFT_SMU_ALM_STM);
#else
                /* Trigger appropriate reaction */
                safetyKitTriggerSwReset(safetyKitResetTypeApplication);
#endif
            }
        }
        else
        {
            break;
        }
    }
} \AppSw\SafetyKit\06_Safe_Computation\SafetyKit_StmMon.c
```

6.2.1.4 Processing – HSM

The Hardware Security Module (HSM) is a separate processor subsystem dedicated for security tasks. Because it is not meant for safety-related applications, it is not part of this application note.

6.2.2 MCU function – Non-volatile memory

The non-volatile memory (NVM) subsystem consists of the data memory unit (DMU), program flash interface (PFI), and non-volatile memory module (comprising the flash standard interface (FSI), program and data flash memories and program flash read write buffer (PFRWB)). For more details, see Section 6 of the AURIX™ TC3xx User's Manual [1].

6.2.2.1 PFlash NVM

The program flash (PFlash) is divided into one or more banks, each connected to a CPU. It is used by the application to store program code and data constants. Compute performance is optimized by using a point-to-point interface to minimize the latency and maximize the bandwidth. Each PFlash is connected to a PFlash read write buffer (PFRWB) that performs the ECC correction and detection, and provides the read data to the system.

The following structure terms are used for the PFlash module.

- **Flash module:** Contains non-volatile memory (NVM) with its own digital logic; in the PFlash, there are one more PF banks
- **Bank:** Supports concurrent operation with some limitations. One bank is equal to three physical sectors: PS0, PS1, and PS2.
- **Physical sector:** One PS is equal to 64 logical sectors, i.e., S0 to S63. Each physical sector is equal to 1 MB.
- **Logical sector:** One logical sector is equal to 16 KB, and each logical sector is equal to 32-word line. It is the smallest unit to erase PFlash.
- **Word line:** One word line is equal 512 bytes and is equal to 16 pages
- **Page**: Smallest unit that can be programed. One page is equal to 32 bytes

For detailed description, features, and multiple sector partition, see the AURIX™ TC3xx User's Manual [1].

Safety Kit implementation

The following external safety mechanisms are implemented in AURIX™ Application Kit - TC3xx Safety and belong to safe computation.

Safety mechanism PFlash: INTEGRITY_CHECK

For the implementation of this SM, dummy safety data is stored at PF0 in array at the address 0xA010 0000. At each run cycle, the CRC of this dummy safety data is calculated via the FCE module and compared with the expected CRC value. If the calculated CRC does not match the expected CRC, a software alarm will be generated.

Code Listing 15 Safety Mechanism PFLASH: INTEGRITY_CHECK

```
#define DATA_LENGTH_WORDS_IC   25U      /* Message data size in words (32-bit)  */
#define PFLASH0_START_ADDRESS_IC      0xA0100000   /* Cached address where the
user wants to store the safety data i.e. Bank Pf0 */


/* data stored at Pf0 memory space for SM:PFLASH:INTEGRITY_CHECK */
const uint32 dummySafetyDataIc [DATA_LENGTH_WORDS_IC] __at(PFLASH0_START_AD-
DRESS_IC)=
{
0xbe9957bb, 0x1c706c1e, 0x14c3db3f, 0x7fb17a93, 0xb0d9d5a7, 0x768093e0,
0x88b206a0, 0xc51299e4, 0xe8a97d48, 0x89367f27, 0x70095984, 0xec030f75,
0xdc22f8d4, 0xd951407b, 0x34ae18c6, 0x4d47ba7d, 0x0e2e4622, 0x4a2e90d3,
0xdaec3752, 0xcd3ed11c, 0x36b416b7, 0x8ea28658, 0xdd37eee3, 0x23928b62,
0x84eb4b22,
};

/*
 * Data integrity by check by CRC calculation comparison
 * SM:PFLASH:INTEGRITY_CHECK
 */
void runIntegrityCheckPFLASH(void)
{
    uint32 *safetyDataIntegrityCheck = (uint32 * )PFLASH0_START_ADDRESS_IC;
    /* CRC calculation with FCE Kernel 0 (CRC32) */
    runCrcCheckFCE(CRC_EXPECTED_RESULT_IC, safetyDataIntegrityCheck,
DATA_LENGTH_WORDS_IC);
} \AppSw\SafetyKit\6_Safe_Computation\SafetyKit_NvmPflash.*
```

Safety Mechanism PFLASH: UPDATE_CHECK

For the implementation of this SM, dummy safety data is stored at PF1 in array at the address 0xA040 0000 where the expected CRC is also known. Every time there is an update of this safety data, the CRC of this updated safety data must be calculated via the FCE module. After the calculation, the application SW compares whether the calculated CRC matches the expected CRC; software alarm will be generated if a mismatch occurs.

There is also the possibility to inject single-bit errors, double-bit errors, and multiple-bit errors via a dedicated button on Application Kit Safety. Single-bit and double-bit errors can be detected and corrected, while multiple-bit errors can only be detected in the AURIX™ TC3xx microcontroller.

Injecting an error is done with the following steps:

1. Erase and write flash to the dedicated data.
2. Read the PFlash.
3. Automatically calculate the ECC value of one page and store it in *MODULE_PFI1.ECCR.B.RCODE*.
4. Store the ECC in WCODE i.e., *MODULE_DMU.HF_ECCW.B.WCODE = MODULE_PFI1.ECCR.B. RCODE*.
5. Change the configuration so that the next time when a read occurs, the flash takes the ECC value from WCODE.
6. Write the inject error data to flash. You can only enter error from ‘0’ to ‘1’ but not the other way around.
7. Change the configuration again to automatic ECC.
8. Make dummy read again to trigger the ECC error.

Code Listing 16 Safety Mechanism PFLASH:UPDATE_CHECK

```
#define PF1_SBE_STARTING_ADDRESS     0xA0400000 //0x80400000     /* Address of the
PFLASH 1 where the data is written  */
#define PF1_DBE_STARTING_ADDRESS     0xA0400020 //0x80400000     /* Address of the
PFLASH 1 where the data is written  */
#define PF1_MBE_STARTING_ADDRESS     0xA0400040 //0x80400000     /* Address of the
PFLASH 1 where the data is written  */

void injectPflashEccError(uint8 err_inj_type)
{
[…]
    /* Get the current password of the CPU WatchDog module */
    passwordEndInit = IfxScuWdt_getCpuWatchdogPasswordInline(&MOD-
ULE_SCU.WDTCPU[IfxCpu_getCoreIndex()]);

    /* Write data at dedicated address in PFLASH */
    writePFLASH(injectErrorAddressPF1, &dataPflashUpdate[0]);

    /* SM:PFLASH:UPDATE_CHECK */
    runUpdateCheckPFLASH(injectErrorAddressPF1);

    /* Make dummy read to get ECC value of working Page */
    dummyRead = MEM(injectErrorAddressPF1);

    IfxScuWdt_clearCpuEndinitInline(&MODULE_SCU.WDTCPU[IfxCpu_getCoreIndex()],
passwordEndInit);

    /* Write in WCODE field the valid ECC value of working Page */
    MODULE_DMU.HF_ECCW.B.WCODE = MODULE_PFI1.ECCR.B.RCODE;

    /* Change ECC encoding configuration so that ECC code is taken from WCODE */
    MODULE_DMU.HF_ECCW.B.PECENCDIS = 3U;

    IfxScuWdt_setCpuEndinitInline(&MODULE_SCU.WDTCPU[IfxCpu_getCoreIndex()], pass-
wordEndInit);

    /* write to Pflash with error value */
    writePFLASH(injectErrorAddressPF1, &injectErrorPflashData[0]);

    /* Change ECC encoding configuration so that ECC code is automatically calcu-
lated */
    IfxScuWdt_clearCpuEndinitInline(&MODULE_SCU.WDTCPU[IfxCpu_getCoreIndex()],
passwordEndInit);
    MODULE_DMU.HF_ECCW.B.PECENCDIS = 0U;
    IfxScuWdt_setCpuEndinitInline(&MODULE_SCU.WDTCPU[IfxCpu_getCoreIndex()], pass-
wordEndInit);

    /* Make dummy read to trigger the ECC error */
    dummyRead = MEM(injectErrorAddressPF1);

    /* Restore the interrupts state */
    IfxCpu_restoreInterrupts(interruptState); */

}\AppSw\SafetyKit\6_Safe_Computation\SafetyKit_PflashProgramming.c
```

The Evaluation Board - AURIX™ TC3xx Safety V3.1 has three dedicated buttons (see Figure 21) to inject SBE, DBE, and, MBE.

Figure 21 SPE, DBE, and MBE dedicated buttons

Pressing the SBE button injects a single-bit error; as a result, the following alarm is generated:

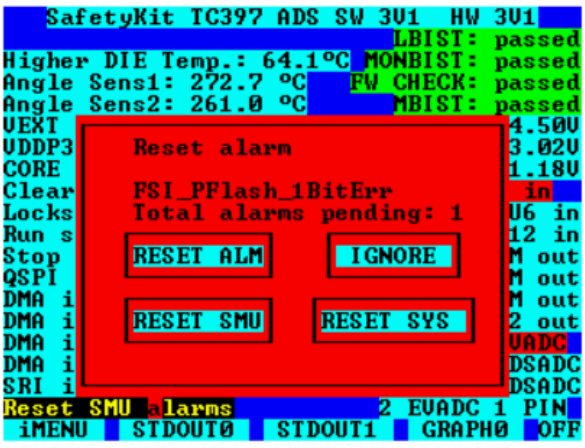


Figure 22 SBE SMU alarm

The DBE button can be used to inject a double-bit error. When pressed, the following alarm is generated. AURIX™ can correct single-bit and double-bit errors.

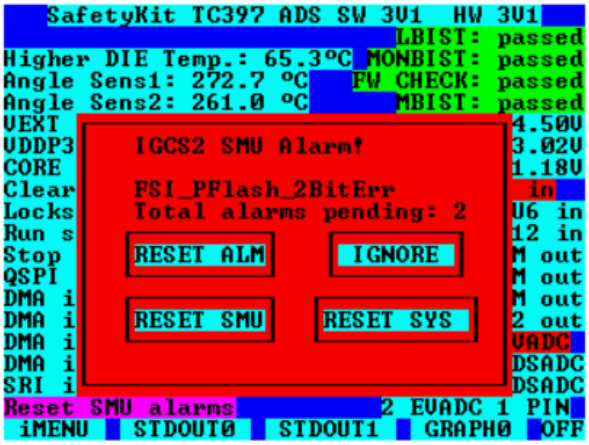


Figure 23 DBE SMU alarm

Pressing the MBE button injects a multiple-bit error; the corresponding alarm is generated. In AURIX™, multiple-bit error is only detected but not corrected.

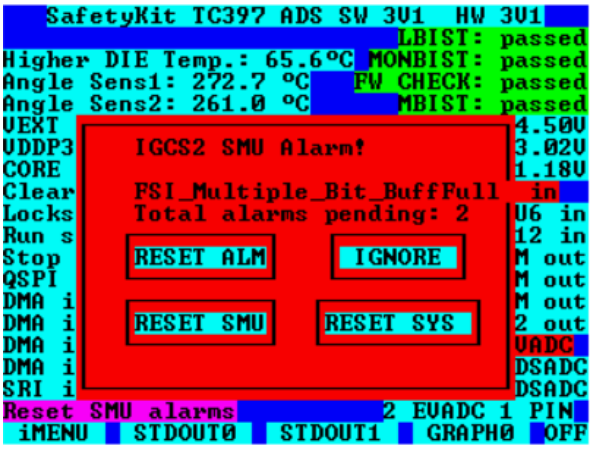


Figure 24 MBE SMU alarm

Safety Mechanism PFlash: WL_FAIL_DETECT

When a DBE occurs, there is high probability that the complete word line (WL) is corrupted. Therefore, when an SMU alarm is generated indicating DBE correction, the application software must verify the integrity of PFlash word line in which the DBE occurred by reading the next 10 pages and monitor the multiple-bit error.

Code Listing 17 Safety Mechanism PFLASH:WL_FAIL_DETECT

```
const AlarmConfigStruct globalAlarmConfig[USER_ALARM_NUMBER] =
{
[…]
    {IfxSmu_Alarm_FSI_PFlash_SingleBitError,              IfxSmu_InternalA-
larmAction_igcs2,   FALSE,      FALSE,      NULL_PTR},

    {IfxSmu_Alarm_FSI_PFlash_DoubleBitError,              IfxSmu_InternalA-
larmAction_igcs2,   FALSE,      FALSE,      &enablePflashWlFailDetect},
    {IfxSmu_Alarm_FSI_Multiple_BitErrorDetectionTrackingBufferFull, IfxSmu_Inter-
nalAlarmAction_igcs2,   FALSE,      FALSE,      NULL_PTR},

[…]
};

/* function to enable flag when dbe occur */
void enableWlFailDetectPFLASH (void)
{
    doubleBitErrorOccur = TRUE;
}

/* read the 10 pages after double bit error injected address
 * SM:PFLASH:WL_FAIL_DETECT
 */
void runWordlineFailDetectPFLASH(void)
{
    uint32 i;
    uint32 *dataWlFaildetectPFLASH[NUM_OF_PAGES];

    /* read 10 pages to check if any page is corrupted */
    for(i=0; i < NUM_OF_PAGES; i++)
    {
        dataWlFaildetectPFLASH[i] = (uint32 * )(PFLASH1_DBE_ADDRESS + (i*0x10));
    }

}\AppSw\SafetyKit\6_Safe_Computation\SafetyKit_NvmPflash.c
```

6.2.3 MCU function – Volatile memory

6.2.3.1 Extension Memory (EMEM)

Extension Memory (EMEM) can be used for ADAS applications, calibration, or trace data storage alternatively; it contains the RAM blocks (EMEM tiles). EMEM has interface to Multi Core Debug Solution (MCDS), SPU to EMEM Protocol (SEP), Shared Resource Interconnect (SRI), and Back Bone Bus (BBB) protocol. EMEM has the following features:

- Software may configure the operation mode of each individual EMEM tile
- 16 KB eXtra Trace Memory (XTM) for trace data only
- Standby power supply for TCM and XCM

Application Kit Safety implementation

The Safety Application Kit contains implementation of *Safety Mechanism EMEM:DATA_INTEGRITY*. Here, if the system integrator wants to use EMEM in a safety-critical application, the application SW must check the corruption of data stored at the EMEM caused by random hardware faults by using information redundancy. The application SW must compare the CRC expected with the measured one and respond accordingly.

Code Listing 18 shows the implementation of this SM in Application Kit Safety. The dummy safety data is stored at start address of the EMEM at 0xB900 0000, and the expected CRC valued is calculated. While at the start of the program, the CRC will measure via the FCE module and compare it with the expected value; a software alarm is generated if a mismatch occurs.

Code Listing 18 Safety Mechanism EMEM: DATA_INTEGRITY

```
#define CRC_EXPECTED_RESULT_EMEM     0xA7EE4C1C    /* EMEM Integrity Check */
#define DATA_LENGTH_WORDS_EMEM       8U           /* Message data size in words
(32-bit) */

uint32 dummySafetyDataEmem [DATA_LENGTH_WORDS_EMEM] =
{
        0x0e9957bb, 0x1c706c1e, 0x14c3db3f, 0x7fb17a93, 0xb0d9d5a7, 0x768093e0,
0x88b206a0, 0xc51299e4,
};

/*
 *  SM:EMEM:DATA_INTEGRITY */
/* put data to EMEM
 */
void initDataEMEM(void)
{
    uint32 *ptrEmemStartAdd  = (uint32 *)IFXEMEM_START_ADDR_CPU;

    /* put the dummy data to EMEM */
    uint32 i;
    for (i=0;i<(DATA_LENGTH_WORDS_EMEM);i++)
    {
        *ptrEmemStartAdd++ = dummySafetyDataEmem[i];
    }
}

/* Start the calculation
 * SM:EMEM:DATA_INTEGRITY *
 */
void runDataIntegrityEMEM(void)
{
    uint32 *ememSafetyData = (uint32 *)IFXEMEM_START_ADDR_CPU;

    /* CRC calculation with FCE Kernel 0 (CRC32) */
    runCrcCheckFCE(CRC_EXPECTED_RESULT_EMEM, ememSafetyData,
DATA_LENGTH_WORDS_EMEM);

} \AppSw\SafetyKit\6_Safe_Computation\SafetyKit_Emem.c
```

6.2.3.2 LMU

Currently not implemented.

6.2.3.3 SRAM

There are separate instances of SRAMs in the MCU. SRAM blocks implement various hardware SMs to assert data integrity using error detection and correction code (ECC/EDC). The SMs also includes a verification of the address logic. See the AURIX™ TC3xx Safety Manual [3] for the complete list of the internal hardware SMs.

The application software is responsible for running and checking the BIST for latent fault detection in the SRAM Safety Flip-Flop (SFF) once per driving cycle using *Safety Mechanism MCi:REG_MONITOR*. The software integrator should identify all the relevant MCU blocks that need the REG_MONITOR test, execute it in software, check the execution time, and check the result of the BIST. See *Safety Mechanism MCi:REG_MONITOR_TEST* of each RAM module, where “MCi” designates the specific RAM module under test.

A dedicated SMU alarm is triggered for every MONITOR_TEST if an error occurs. The application software is responsible for clearing the affected register after running the test:

- MCi.ECCD
- MCi.FAULTSTS
- SMU alarm (if raised)

Safety Kit implementation of the safety mechanism MCi: REG_MONITOR_TEST

The REG_MONITOR test is executed immediately after the application SW startup (see Section 3.3).
The test consists of the following steps, which is shown in Code Listing 19:

1. Start the test.
2. Monitor the execution time with a simple timeout counter.
3. Check if an error was raised (an SMU alarm is raised by hardware anyway).
4. Clear the ECCD and FAULTSTS registers.
5. Report the error status.

Code Listing 19 Code snippet of the REG_MONITOR_TEST function

```
/* SM:*:REG_MONITOR_TEST */
void safetyKitRunRegMonitorTest(void)
{
    g_SafetyKitStatus.regMonitorTestAllFB = NA;
    boolean regMonitorTestPassed = TRUE;

    for(int i = 0; i < safetyKitRegMonSetSize; i++)
    {
        IfxMtu_MbistSel regToCheck = safetyKitRegMonSet[i];

        /* register monitor test if each SSH */
        regMonitorTestPassed &= safetyKitRunSshRegMonitorTest(regToCheck);
    }

    if(regMonitorTestPassed == TRUE){
        g_SafetyKitStatus.regMonitorTestAllFB = pass;
    }
    else{
        g_SafetyKitStatus.regMonitorTestAllFB = fail;
    }
}
/* run SSH register monitor test */
boolean safetyKitRunSshRegMonitorTest(uint32 sshReg)
{
    Ifx_MTU_MC *mc = &MODULE_MTU.MC[sshReg];

    boolean sshRegMonitorTestPassed = TRUE;
    uint64 tStart, tExecution;
    float64 tExecutionSec;
    uint16 password = IfxScuWdt_getSafetyWatchdogPassword();
    IfxScuWdt_clearSafetyEndinit(password);
    /* Trigger the SFF self test */
    mc->ECCS.B.SFFD = 1;
    IfxScuWdt_setSafetyEndinit(password);

    tStart = IfxStm_get(&MODULE_STM0);
    /* Wait until test is done, as long test is not finished store measure the execution time */
    do{
        tExecution = IfxStm_get(&MODULE_STM0) - tStart;
    }
    while(mc->ECCS.B.SFFD != 0);
    /* Convert the time to seconds */
    tExecutionSec = tExecution / IfxStm_getFrequency(&MODULE_STM0);

    /* Validation */
    uint16 regMISCERR = mc->FAULTSTS.B.MISCERR;
    if(regMISCERR != 0)
    {
        sshRegMonitorTestPassed = FALSE;
    }
    if(tExecutionSec > REG_MONITOR_TEST_MAX_TIME_S)
    {
        sshRegMonitorTestPassed = FALSE;
    }
    /* Clear SSH */
    mc->ECCD.U = 0x0;

    password = IfxScuWdt_getSafetyWatchdogPassword();
    IfxScuWdt_clearSafetyEndinit(password);
    mc->FAULTSTS.U = 0x0;
    IfxScuWdt_setSafetyEndinit(password);

    return sshRegMonitorTestPassed;
} \AppSw\SafetyKit\02_Safety_Mechanisms\SafetyKit_RegMon.c
```

6.2.3.3.1 LMU_DAM

The Default Application Memory (DAM) is a Shared Resource Interconnect (SRI) peripheral providing access to volatile memory resources. Its primary purpose is to provide 64 KB or 32 KB of local memory for general-purpose usage. The amount of memory available depends on the product.

The following features are implemented in the DAM:

- **64 KB of SRAM depending on the product:**
  - Organized as 64-bit words
  - Support for byte, half-word, and word accesses as well as double-word and burst accesses

- **Protection of DAM SRAM contents:**
  - Eight programmable address regions can be protected
  - Each address range has a programmable list of bus masters permitted read or write access based on the unique master Tag ID

The DAM SRAM can be used for code execution or data storage.

Safety Kit implementation

If the system Integrator wants to use LMU_DAM in safety-critical applications, the application software must monitor the corruption of data stored in LMU_DAM caused by random hardware faults. For this purpose, the use of CRC comes in; the CRC expected value of data stored at LMU_DAM must be compared with the calculated CRC at the start of program via the FCE module and activate the SMU software alarm according to the result.

Code Listing 20 Safety Mechanism LMU_DAM: DATA_INTEGRITY

```
#define CRC_EXPECTED_RESULT_LMU_DAM0     0xA7EE4C1C /*lmu Dam0 Integrity Check */
#define DAM0_START_ADDRESS               0xB0400000 /*Cached address of DAM where the user stores the
safety data*/
#define DATA_LENGTH_WORDS_DAM            8U         /* Message data size in words (32-bit) */
uint32 dummySafetyDataLmuDam [DATA_LENGTH_WORDS_DAM] =
{
        0x0e9957bb, 0x1c706c1e, 0x14c3db3f, 0x7fb17a93, 0xb0d9d5a7, 0x768093e0, 0x88b206a0,
0xc51299e4,
};
/* SM:AMU.LMU_DAM:DATA_INTEGRITY
 * Initialization of DLMU
 * This function is called from main during initialization phase
 */
void initDataLMUDAM(void)
{
    uint32 *ptrStartAddrDAM0   = (uint32 *)DAM0_START_ADDRESS;
    uint32 i;
    for (i=0; i < DATA_LENGTH_WORDS_DAM; i++)
    {
        /* store the Dummy LMU DAM data to DAM0 */
        *ptrStartAddrDAM0++ = dummySafetyDataLmuDam[i];
    }
}
/*
 * data integrity by checking CRC
 * SM:AMU.LMU_DAM:DATA_INTEGRITY
 */
void runDataIntegrityLMUDAM(void)
{
    uint32 *safetyDataLMUDAM = (uint32 * )DAM0_START_ADDRESS;

    /* CRC calculation with FCE Kernel 0 (CRC32) */
    runCrcCheckFCE(CRC_EXPECTED_RESULT_LMU_DAM0, safetyDataLMUDAM, DATA_LENGTH_WORDS_DAM);
}

\AppSw\SafetyKit\6_Safe_Computation\SafetyKit_LmuDam.c
```

6.2.3.4 Default Application Memory (DAM)

6.2.3.5 Volatile Memory Test (VMT)

The Volatile Memory Test (VMT) module contains the hardware for the MBIST functionality: *Safety Mechanism MBIST*. Faults affecting the functionality are covered by the LBIST. See Section 6.1.4 Monitor built-in self-test (MONBIST) for more information.

6.2.4 MCU function – ADAS

The current version of Application Kit Safety does not support ADAS.

6.2.5 MCU function - interconnect

6.2.5.1 System Resources Interconnect (SRI)

The AURIX™ TC3xx platform has three independent on-chip connectivity resources:

- System Resources Interconnect Fabric (SRI Fabric)
- System Peripheral Bus (SPB)
- Backbone Bus (BBB)

The SRI fabric connects the TriCore™ CPUs, DMA module, and other high-bandwidth requestor TriCore™ memories, and other resources for instruction fetches and data accesses.

An SRI error condition can be detected by both masters and slaves depending on the condition. Errors are reported via alarms to the SMU or interrupts or traps by a CPU. There are three error types:

- SRI protocol errors
- SRI transaction ID errors
- SRI EDC errors

Safety Application Kit implementation

The Safety Application Kit contains the implementation of *Safety Mechanism SRI: ERROR_HANDLING* where Safety Manual V2.0 says that if the SRI triggers an error interrupt request, the application SW must check the type of SRI error and the stored SRI diagnostic information. The application SW must evaluate the type of error and trigger the most appropriate reaction. The types of SRI errors are readable via ERRADDRx (x= 0 … 15), ERRx, PESTAT, TIDSTAT, and PECON.

In Application Kit Safety, this SM runs periodically and checks the SRI error. See the `storeSriDiagnosticInfo(void)` and `runSriErrorInjection(void)` functions in *\AppSw\SafetyKit\06_Safe_Computation\SafetyKit_Sro_Error_Handling.c.*

The user can also manually inject an error via CPU_SEGEN (used to inject the SRI error). The error injection is done via the TFT display. In the TFT display, press SRI inject error as shown in Figure 25. After pressing SRI inject error from the TFT menu, the following SMU alarm will appear.

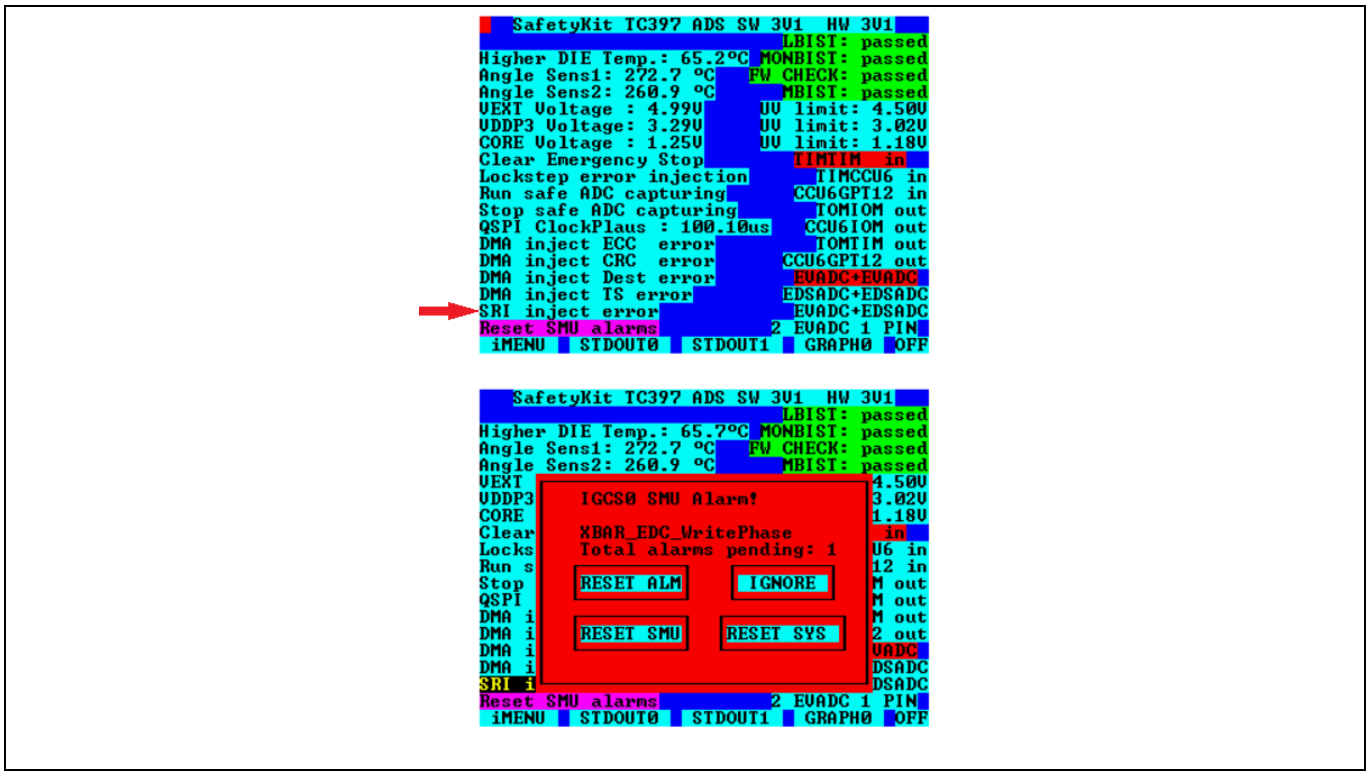


Figure 25 SRI on TFT menu and alarm after error injection

Note: Some code is commented out for this SM to inject different types of SRI errors. Uncomment the related code and then see the corresponding SMU alarm for the SRI error.

6.2.6 MCU function – Communication

6.2.7 Direct Memory Access (DMA)

The Direct Memory Access (DMA) controller moves data from source addresses to destination addresses without the intervention of the CPU or other on-chip resources. The DMA module provides numerous safety mechanisms to detect failures on its own. Some errors, such as ENDINIT protection violation, generate SMU alarms on their own. Other errors are handled by Safety *Mechanism DMA:ERROR_HANDLING:*

- Transaction Request Lost (TRL)
- Source Error (SER)
- Destination Error (DER)
- DMA RAM Error (RAMER)
- DMA Linked List Error (DLLER)
- Safe Linked List Error (SLLER)

Finally, some failures need a combination of various safety mechanisms to be detected:

- **Address CRC error:** Safety Mechanism DMA:ADDRESS_CRC
- **Data CRC error:** Safety Mechanism DMA:DATA_CRC
- **Transaction time:** Safety Mechanism DMA:TIMESTAMP
- **Supervision:** Safety Mechanism DMA:SUPERVISION

The SMs that do not report errors directly to the SMU are collected under *Safety Mechanism DMA:SUPERVISION*. The application software is responsible for triggering the appropriate reaction.

Safety Kit implementation of an DMA transaction with error injection

The Application Kit Safety DMA example is programmed to move a memory array from the source to destination. All the safety mechanisms mentioned above regarding DMA are implemented. Multiple ways of injecting an error are possible – see the user interface on Figure 26:

- ECC error injection. Handled by *Safety Mechanism DMA:SRI_TRANSACTION_INTEGRITY → ALM8[23]*

Code Listing 21 DMA ECC error injection

```
void runSriErrorInjection()
{
    /* Error injection DMA */
    if(injectErrorECC == TRUE)
    {
        /* This error will generate ALM8[23]: Alarm: DMA SRI ECC Error */
        IfxScuWdt_clearCpuEndinit(IfxScuWdt_getCpuWatchdogPassword());

        /* CPUx SRI Error Generation Register */
       __mtcr(CPU_SEGEN, 0x80000201);

        IfxScuWdt_setCpuEndinit(IfxScuWdt_getCpuWatchdogPassword());
        injectErrorECC = FALSE;
    }
[..]
}

\AppSw\SafetyKit\04_Fault_Injection\SafetyKit_Sri_ErrorHandling.c
```

- **Destination error:** Destination is an uninitialized module. Handled by *Safety Mechanism DMA:SUPERVISION.*

Code Listing 22 DMA destination error injection

```
void initAndRunDmaTransaction(void)
{
[..]
    /* SM:DMA:ADDRESS_CRC : compute the expected address CRC stored sdcrc */
    if(injectDestinationError)
    {
        /* Destination is an uninitialized module */
        cfg.destinationAddress = (uint32)wrongDestination;
        injectDestinationError = FALSE;
    }
    else
    {
        cfg.destinationAddress = (uint32)destination;
    }
[..]
}

\AppSw\SafetyKit\04_Fault_Injection\SafetyKit_Dma.c
```

- **CRC error injection:** CRC calculated by the SM[SW] is corrupted. Handled by *Safety Mechanism DMA:SUPERVISION.*

Code Listing 23 DMA CRC error injection

```
void initAndRunDmaTransaction(void)
{
[..]
    /* SM:DMA:ADDRESS_CRC : compute the expected address CRC stored sdcrc
     * SM:DMA:DATA_CRC    : compute the expected data CRC stored rdcrc */
    accumulateCRC((uint8 *)source, (uint8 *)destination, BUFFER_SIZE, 1, 4, 0);

    /* CRC error injection
     * SM:DMA:DATA_CRC    : compute the expected data CRC stored rdcrc */
    if(injectCrcError == TRUE)
    {
        /* Corrupt CRC */
        rdcrc = rdcrc -1;
        injectCrcError = FALSE;
    }
[..]
}
```

\AppSw\SafetyKit\04_Fault_Injection\SafetyKit_Dma.c

- **Transaction time:** Handled by *Safety Mechanism DMA:TIMESTAMP*

Code Listing 24 DMA timestamp

```
void isrCH0DMA(void)
{  […]
    /* Implement SM:DMA:TIMESTAMP
     * */
    dmaTotalTimestampCount = destination[BUFFER_SIZE] - transactionStartTimeStamp;

    if( (dmaTotalTimestampCount >= THRESHOLD_MAX_TIMESTAMP_COUNT)  ||
        (dmaTotalTimestampCount <= THRESHOLD_MIN_TIMESTAMP_COUNT)  )
    {
        error |= DMA_TIMING_ERROR;
    }
[…] }
void initAndRunDmaTransaction(void)
{   […]
    transactionStartTimeStamp = IfxDma_getTimestamp(chn.dma);
    /* check if time stamp error flag is on */
    if ( TRUE == injectTimestampError )
    {
        transactionStartTimeStamp = transactionStartTimeStamp - 5;   /* corrupt
time stamp current value */
     /* transactionStartTimeStamp = transactionStartTimeStamp + 5; *//* corrupt
time stamp current value */
        injectTimestampError = FALSE;
    }
/* Start DMA transaction */
IfxDma_Dma_startChannelTransaction(&chn);}
```

\AppSw\SafetyKit\04_Fault_Injection\SafetyKit_Dma.c

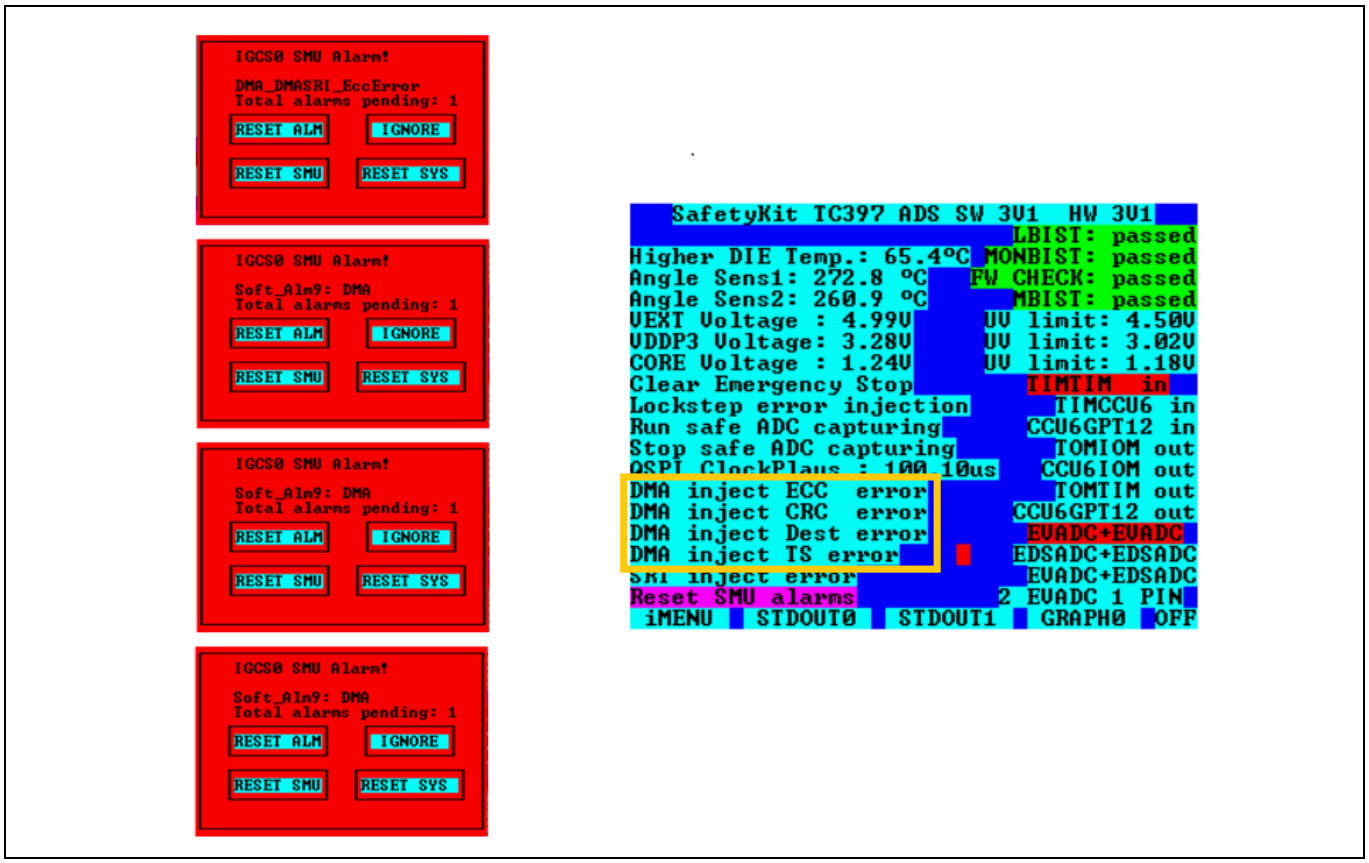


Figure 26 Application Kit Safety DMA demonstration error injection

6.2.7.1 Interrupt Router (IR)

The Interrupt Router (IR) module schedules interrupts (here called “service requests”) from external resources, internal resources, and the SW to the CPU and the DMA modules (here called “service provider”). For details of this module, see Chapter 16 of the AURIX™ TC3xx User Manual.

Safety Kit implementation

Application Kit Safety contains the implementation of the *Safety Mechanism ISR_MONITOR*, which is related to the IR module. The application SW must detect the missed or unintended service request for safety-related interrupts. When a missed or unintended safety-related interrupt is detected, the application SW must trigger the most appropriate reaction depending on the application. There are different ways to implement this SM, but the safety application contains a plausibility check of periodic interrupt: certain interrupts are expected periodically (e.g., ATOM generates interrupt of every falling or rising edge). Based on this expected rate (or for example, within each FTTI), the application must implement periodic checks for plausibility of this interrupt i.e., if an expected interrupt is missing or unintended interrupts occur.

Code Listing 25 Safety Mechanism ISR_MONITOR DMA

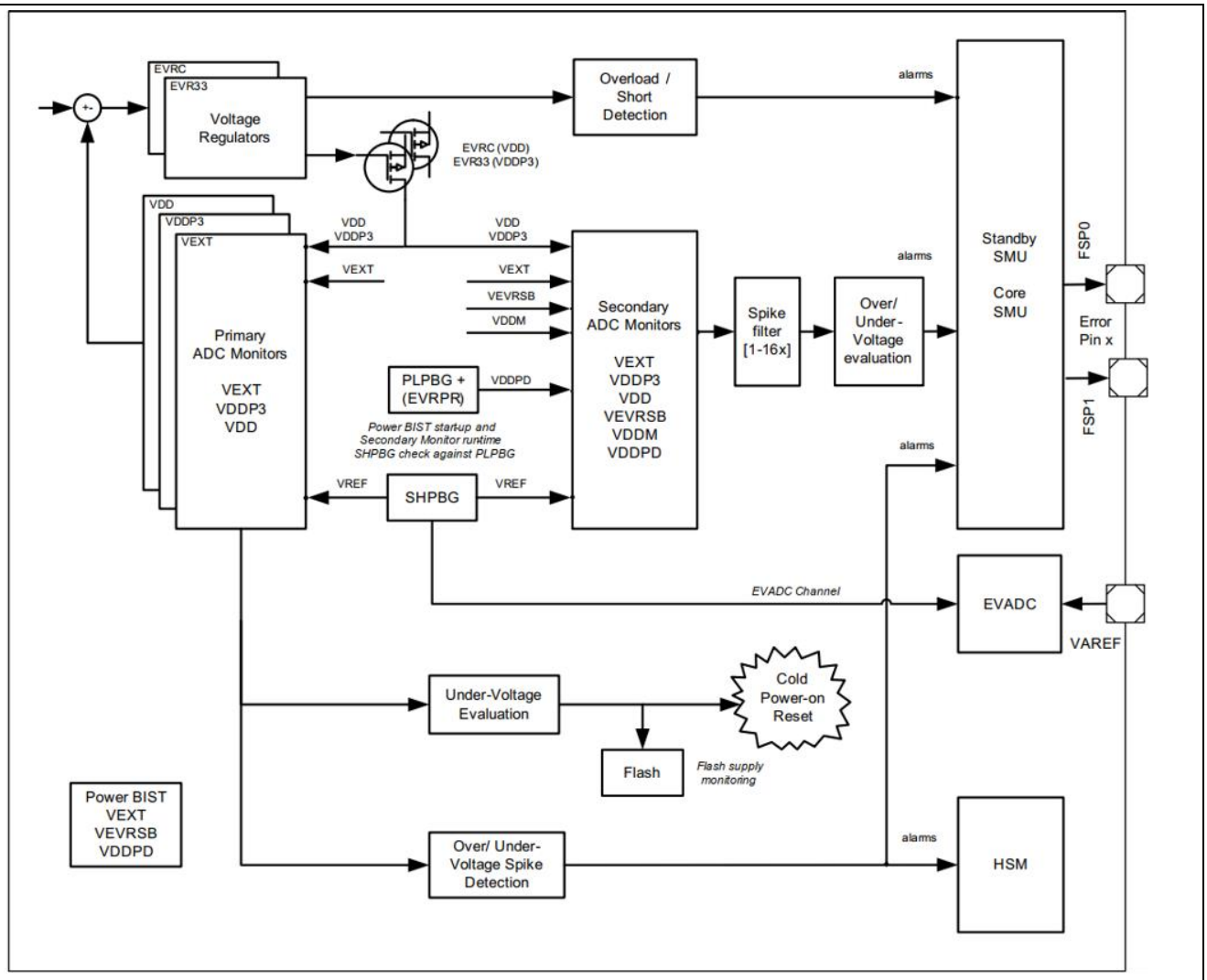
```
/*
 * Interrupt Service Routine of the ATOM
 * */
void interruptHandlerGtmAtom(void)
{
    IfxGtm_Atom_Timer_acknowledgeTimerIrq(&g_timerDriver);     /* Reset the timer event */
    currentTicksOnTimMission = STM0_TIM0.B.STM_31_0;           /* take the STM ticks value
*/
}

/*
 * SM:IR:ISR_MONITOR
 * */
void isrMonitor()
{
    uint32 diffTicksOnTimMission;
    if (currentTicksOnTimMission != previousTicksOnTimMission)
    {
        diffTicksOnTimMission = currentTicksOnTimMission - previousTicksOnTimMission;
        if (diffTicksOnTimMission > MAX_THRESHOLD_TICK || diffTicksOnTimMission <
MIN_THRESHOLD_TICK)
        {
            IfxPort_togglePin(ISR_MONITOR_LED_D8.port, ISR_MONITOR_LED_D8.pinIndex);
        }

        previousTicksOnTimMission = currentTicksOnTimMission;
    }
}

/*
 * This function initializes the ATOM
 * SM:GTM_CONFIG_FOR_ATOM
 * */
void initGTMATOM(void)
{
    IfxGtm_enable(&MODULE_GTM);            /* Enable GTM                        */

    IfxGtm_Atom_Timer_Config timerConfig;   /* Timer configuration structure    */
    IfxGtm_Atom_Timer_initConfig(&timerConfig, &MODULE_GTM); /* Initialize default parame-
ters    */

    timerConfig.atom = IfxGtm_Atom_0;
    timerConfig.timerChannel = IfxGtm_Atom_Ch_0;
    timerConfig.clock = IfxGtm_Cmu_Clk_0;
    timerConfig.base.frequency = ATOM_FREQ;
    timerConfig.base.isrPriority = ISR_PRIORITY_ATOM_ISR_MONITOR;
    timerConfig.base.isrProvider = IRQ_GET_TOS(ISR_PROVIDER_ISR_MONITOR);

    IfxGtm_Atom_Timer_init(&g_timerDriver, &timerConfig);                    /* Ini-
tialize the ATOM              */
    IfxGtm_Atom_Timer_run(&g_timerDriver);
} \AppSw\SafetyKit\06_Safe_Computation\SafetyKit_Isr_Monitor.c
```

6.2.8 MCU function – Infrastructure

6.2.8.1 Power management system (PMS)

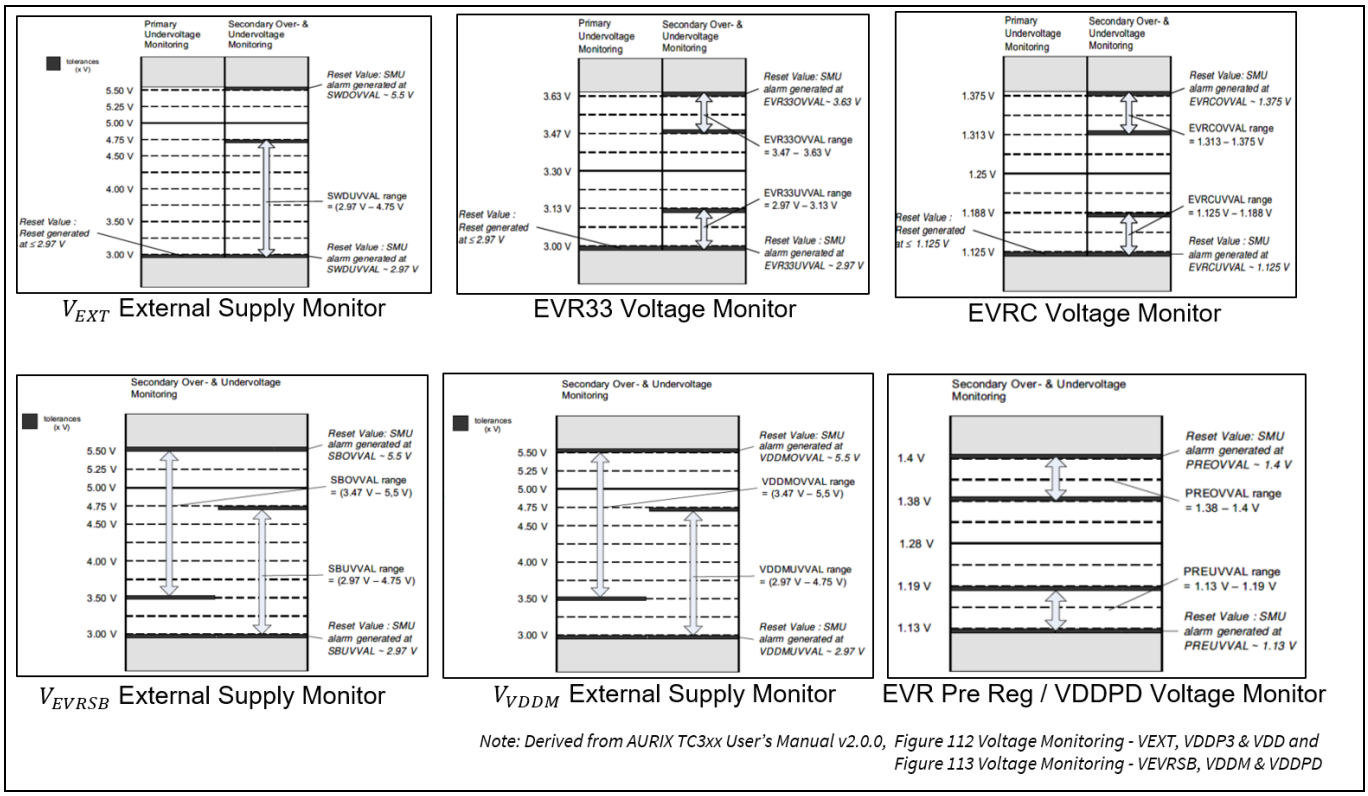
The PMS handles multiple voltage levels required to supply the different power domains of the AURIX™ TC3xx MCU. Depending on the HWCFG[1,2] pins, the PMS is responsible for distributing and monitoring the supplied voltages or to generate the voltages itself if single-supply mode is selected. The PMS is responsible for checking all the voltages (generated or supplied). Each voltage range has a primary undervoltage limit, and a user-configurable secondary upper and lower limit (*Safety Mechanism VX_MONITOR_CFG*). The primary monitor protects the device from incorrect operation at low voltages by generating a cold PORST. The secondary monitor alerts the system of a potential danger through the SMU. See Figure 27 and Figure 28 for an overview of supply monitors.

The secondary voltage monitor is configurable. As soon as the voltage exceeds or falls below a pre-configured voltage level from the secondary voltage monitor, an alarm will be triggered. Additionally, the upper and lower threshold voltage levels can be set up for single-supply voltages. These are used to trigger an SMU alarm to indicate a fault whenever the levels are crossed.



Note: Derived from AURIX TC3xx User's Manual v2.0.0, Figure 109 Supply Monitor Overview

Figure 27 Supply monitor overview



Note: Derived from AURIX TC3xx User's Manual v2.0.0, Figure 112 Voltage Monitoring - VEXT, VDDP3 & VDD and Figure 113 Voltage Monitoring - VEVRSB, VDDM & VDDPD

Figure 28 Voltage monitoring ranges

Safety Kit implementation of the PMS voltage monitoring

To demonstrate the functionality of the internal voltage monitor, the external voltage supply can be loaded with an external pull-down switch. With the encoder, the voltage can be adjusted through the DC voltage of the pull-down switch. All undervoltage limits are adjustable using the touchscreen (Figure 32). If a limit is violated, a secondary monitor SMU alarm is triggered. The lower screenshots show the messages triggered by an SMU alarm (violation of the undervoltage limit).

To demonstrate the voltage monitoring safety mechanism, the supplied voltage delivered by TLF35584 must be put in an unstable state. No self-test mechanism is present for this purpose in the AURIX™ MCU or in TLF35584 devices. For such cases, a special circuitry is designed on the Evaluation Board Safety as shown in Figure 30 and Figure 31. The idea is to overload the TLF35584 (V_UC) power line with a pull-down switch; the pull-down switch is PWM controlled by the AURIX™ TC3xx MCU. This PWM generation is controlled via the incremental encoder counter (EN1) as shown in Figure 29.

Once the critical current capability of TLF35584 is reached, the voltage will drop. Once the voltage drops below the AURIX™ secondary monitor threshold, an SMU alarm will be raised. If the voltage drop exceeds the TLF35584 critical undervoltage threshold, TLF35584 will trigger a PORST. The code for that example is provided in *\AppSw\SafetyKit\04_Fault_Injection\SafetyKit_UndervoltageSimulation.c*.

Attention: ***Buttons and switches should be operated sequentially on the Safety Board because short-circuits will appear when enabling simultaneously a strong pull-up and a strong pull-down circuitry on the same line. This could permanently damage the demo setup. In such cases, the board can also enter a safe mode due to the overtemperature condition. In that case, the board is temporarily unusable (but not damaged) for approximately one minute.***

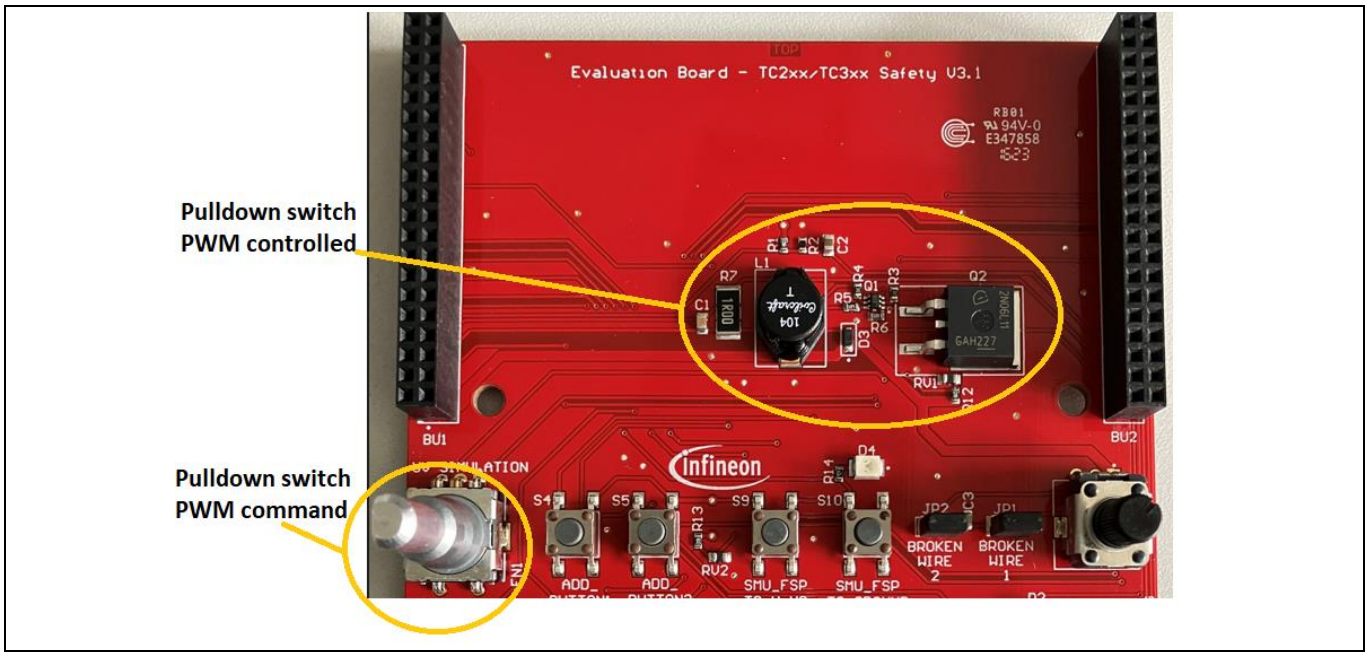


Figure 29 Overview of undervoltage simulation

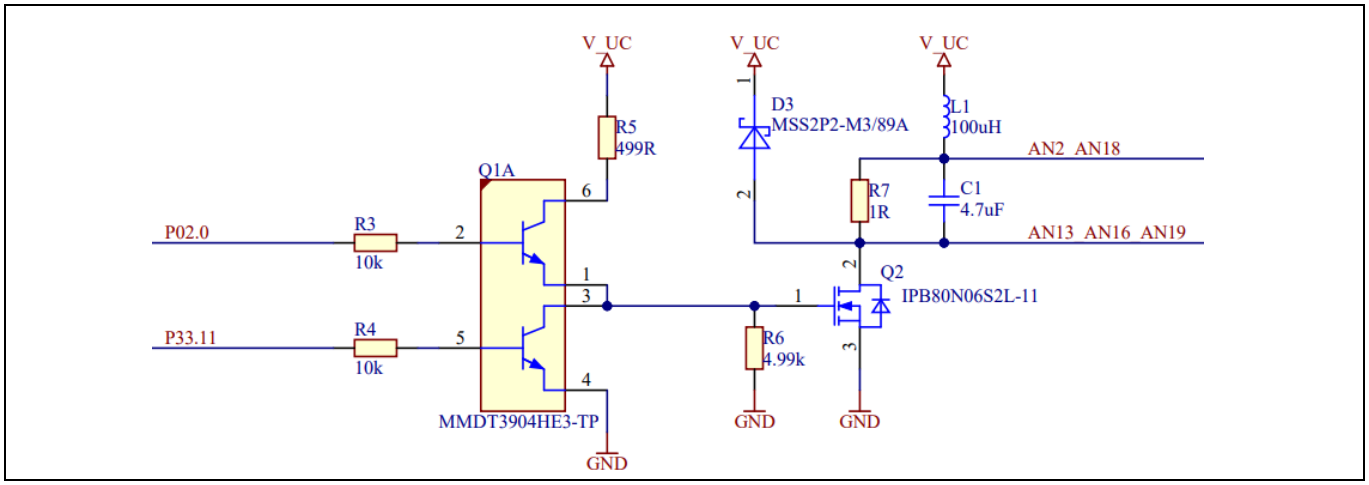


Figure 30 Undervoltage fault injection circuitry

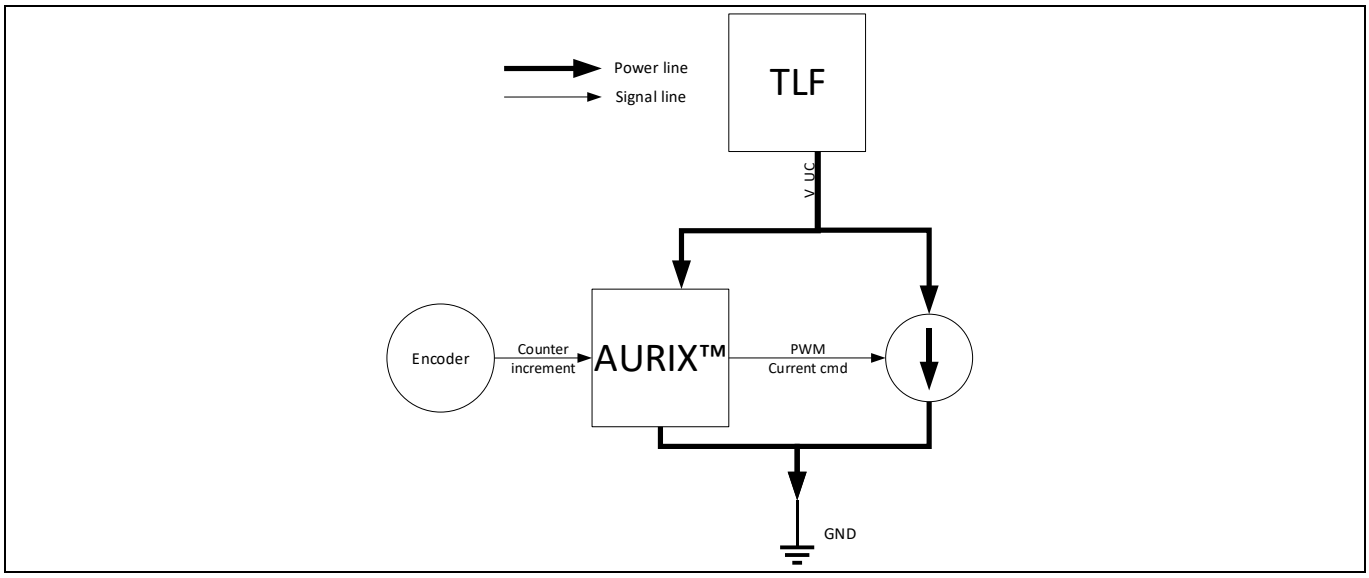


Figure 31 V_UC overloading concept

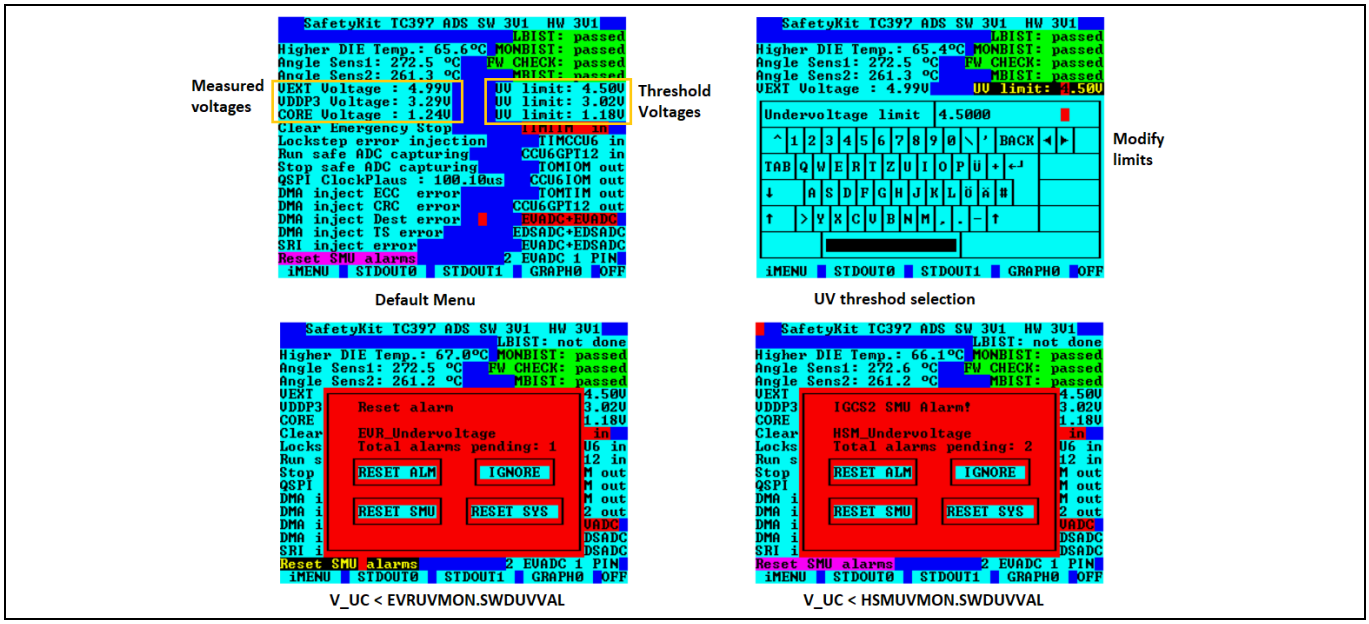


Figure 32 Triggering an undervoltage alarm

6.2.8.2 Clock

The clock system is built up as a chain composed of different building blocks, which allow different function parts for the complete chain. The building blocks are:

- Basic clock generation (clock source)
- Clock speed upscaling (PLLs)
- Clock distribution (CCU)
- Individual clock configuration (peripherals)

For more details, see Chapter 10 of the AURIX™ T3xx User's Manual [1].

Safety Kit implementation

Application Kit Safety implements the *Safety Mechanism CLOCK:PLAUSIBILITY* where the application SW must evaluate the different clock frequencies provided by PERPLLCON1.K2 and PERPLLCON1.K3 with a time measurement based on the SYSPLLCON1.K2 clock frequency. If the application SW does not match the calculated and expected clock frequency values, the application SW must trigger an alarm.

In Application Kit Safety, the QSPI5 module is used to implement this SM. The following steps are used:

1. Initialize the QSPI5 module.
   - Master mode
   - XXL mode
   - DMA enabled
   - Baud rate of 10 MHz
   - Enabled phase transition (PT) interrupt
2. Fill data into transmit buffer of size 125 bytes.
3. Start the QSPI master data transfer.
4. Capture the time via the STM timer on two PT interrupts.
   - First Interrupt will occur at the start of frame (SoF).
   - Another interrupt will occur at the end of frame (EoF).
5. Calculate the different EoF time and SoF time taken via the STM timer.
6. Compare the resulting value with the range of theoretical values limit and respond accordingly.

Error! Reference source not found. shows the implementation of these steps; the real-time calculated value can be seen on TFT menu as shown in Figure 33.

Code Listing 26 Safety Mechanism CLOCK: PLAUSIBILITY DMA

```
/* Handle QSPI5 Phase Transition interrupt */
void isrQSPI5Pt(void)
{
[…]
 } /*
 * This function initializes the QSPI5 module
 * */
void initQSPI5ForClockPlausibility(void)
{
    g_SafetyKitStatus.qspiEofandSofTimeDifference.timeMinimumThreshold = TRANS-
FER_TIME_MIN_LIMIT;
    g_SafetyKitStatus.qspiEofandSofTimeDifference.timeMaximumThreshold = TRANS-
FER_TIME_MAX_LIMIT;

    /* Initialize the Master */
    initQSPI5Master();
    initQSPI5MasterChannel();
    initQSPI5MasterBuffers();
}
/*
 * Function to check clock Plausibility on every EOF event
 * SM:CLOCK:PLAUSIBILITY
 * */
void checkClockPlausibility(void)
{
    /* check clock plausibility range */
    if ((g_SafetyKitStatus.qspiEofandSofTimeDifference.timeDifference > TRANS-
FER_TIME_MAX_LIMIT) ||
        (g_SafetyKitStatus.qspiEofandSofTimeDifference.timeDifference < TRANS-
FER_TIME_MIN_LIMIT))
    {
        /* check plausibility test failed, Trigger SMU software alarm */
        softwareCoreAlarmTriggerSMU(SOFT_SMU_ALM_CLOCK_PLAUS);
    }
    else
    {
        /* check plausibility test passed */
    }
} \AppSw\SafetyKit\05_Avoid_Detect_CCF\SafetyKit_CloclPlausibility.c
```

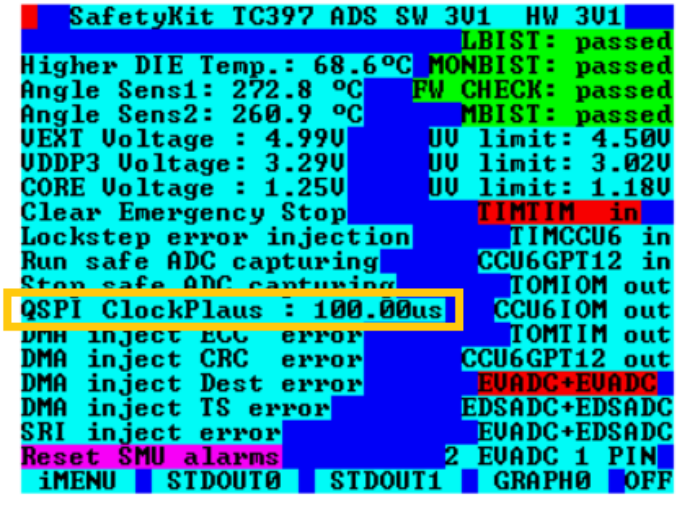


Figure 33 Clock plausibility TFT real-time value

6.2.8.3 RESET

RESET is responsible for configuring and controlling reset events based on the reset trigger source that generates the request. For more information, see the “RESET” section in the Safety Manual [3].

6.2.8.4 System Control Unit (SCU)

The SCU is responsible for controlling various system functions, including:

- Reset Control (RCU)
- External Request Handling (ERU)
- Emergency Stop (ES)
- System registers
- Watchdog Timers (WDT)
- Trap generation ([TRAP])

External request handling (ERU)

Even though there is no need to implement any additional SMs for the SCU module, Application Kit Safety demonstrates the configuration of the Emergency Stop feature and includes fault injection to trigger a safety watchdog timer overflow.

The safety code contains *Safety Mechanism SCU:ERU_CONFIG* implementation as described in the safety manual: the application software must configure the ERU parameters and start external request processing by the ERU according to the User Manual description.

Safety Kit implementation

The Safety Kit provides a button (Stall CPU): see the “Watchdog timer (WDT)” section for details. The Stall CPU button is based on ERU; therefore, this ERU is initialized according to the *SCU:ERU_CONFIG safety mechanism*. The ERU input pin is used to trigger a high-priority interrupt when the Stall CPU button is pressed.

Code Listing 27 Code snippet of ERU configuration

```
/*SM:SCU:ERU_CONFIG
 * */
void initERU(void)
{
    /* Trigger pin */
     g_configERU.reqPin = STALL_CPU_PIN; /* Select external request pin */

     /* Initialize this pin with pull-down enabled
      * This function will also configure the input multiplexers of the ERU (Register EX-
ISx)
      */
     IfxScuEru_initReqPin(g_configERU.reqPin, IfxPort_InputMode_pullDown);

    /* Determine input channel depending on input pin */
    g_configERU.inputChannel = (IfxScuEru_InputChannel) g_configERU.reqPin->channelId;

    /* Input channel configuration */
    IfxScuEru_enableRisingEdgeDetection(g_configERU.inputChannel); /* Interrupt triggers on
                                                    rising edge (Register RENx) and      */
    /* Signal destination */
    g_configERU.outputChannel = IfxScuEru_OutputChannel_1; /* OGU channel 1             */
    /* Event from input ETL1 triggers output OGU1 (signal TRx0) */
    g_configERU.triggerSelect = IfxScuEru_InputNodePointer_1;

    /* Connecting Matrix, Event Trigger Logic ETL block */
    /* Enable generation of trigger event (Register EIENx) */
    IfxScuEru_enableTriggerPulse(g_configERU.inputChannel);
    /* Determination of output channel for trigger event (Register INPx) */
    IfxScuEru_connectTrigger(g_configERU.inputChannel, g_configERU.triggerSelect);

    /* Configure Output channels, OutputGating Unit OGU (Register IGPy) */
    IfxScuEru_setInterruptGatingPattern(g_configERU.outputChannel, IfxScuEru_Inter-
ruptGatingPattern_alwaysActive);

    /* Service request configuration */
    /* Get source pointer depending on outputChannel (SRC_SCUERU0 for outputChannel0) */
    g_configERU.src = &MODULE_SRC.SCU.SCUERU[(int) g_configERU.outputChannel % 4];
    IfxSrc_init(g_configERU.src, IRQ_GET_TOS(ISR_PROVIDER_ERU_IN2), ISR_PRIORITY_ERU_IN2);
    IfxSrc_enable(g_configERU.src);
} \Appkit\SafetyKit\09_Fault_Injection\SafetyKit_Stall_Cpu.c
```

Emergency Stop (ES)

Even though there is no need to implement any additional SMs for the SCU module, Application Kit Safety demonstrates the configuration of the Emergency Stop feature and includes fault injection to trigger a safety watchdog timer overflow.

The ES feature provides a fast and software-independent response to an emergency event. As a reaction to an emergency event, selected output ports can be immediately placed into a defined state. This feature can also be used as an external failure reporting interface that enables the communication of the presence of an internal microcontroller failure to an external safe state controller (See *Safety Mechanism ES_ERROR_PIN_MONITOR* in the Safety Manual [3]). The safe state of the port in the case of an ES is part of the port configuration.

Safety Kit implementation of the ES feature

The ES feature is enabled for the port pin of LED3 in such a way that:

- LED3 is ON by default (LOW sate)
- LED3 is OFF when ES is activated (HIGH state)

See Figure 35.

As shown in Code Listing 28, all four lockstep error SMU alarms are configured to trigger ES (`IfxSmu_InternalAlarmAction` is set to “IGCS1” and the function `IfxSmu_configAlarmActionPES` is called with the parameter `PES_ON_IGCS1`).

The ES status flag bits (EMSF and SEMSF) indicate if an Emergency Stop event has occurred. The state of the ES flag is also indicated by the color of the ES display entry on the TFT screen (see Figure 36). A red color indicates that the flag has been set and that an ES event has occurred. The flag must be cleared to allow a new response if another ES request event occurs. The flags can be cleared by clicking on the entry on the display.

The importance of clearing the flags can be easily understood by triggering the ES for the first time. The LED3 port will immediately enter its safe state by turning off the LED. The ES also triggers a second SMU alarm (additionally to the alarm, which has occurred due to the initial fault injection), which leads to an alarm pop-up window on the display. Even after clearing all alarms, the port does not resume with its initial fault-free operation. Only after clearing the flags (via the TFT, see Figure 36), the LED will turn on again.

Attention: If a second ES event occurs before the flags are cleared, nothing will happen.

Code Listing 28 Code snippet to configure Port ES for IGCS1 alarms

```
const AlarmConfigStruct globalAlarmConfig[USER_ALARM_NUMBER] = {
[..]
    /*-------------------------------------------------- IGCS1 --------------------------------------------------
------*/
    {IfxSmu_Alarm_CPU0_Lockstep_ComparatorError,  IfxSmu_InternalAlarmAction_igcs1,  TRUE,     FALSE,      NULL_PTR},
    {IfxSmu_Alarm_CPU1_Lockstep_ComparatorError,  IfxSmu_InternalAlarmAction_igcs1,  TRUE,     FALSE,      NULL_PTR},
    {IfxSmu_Alarm_CPU2_Lockstep_ComparatorError,  IfxSmu_InternalAlarmAction_igcs1,  TRUE,     FALSE,      NULL_PTR},
    {IfxSmu_Alarm_CPU3_Lockstep_ComparatorError,  IfxSmu_InternalAlarmAction_igcs1,  TRUE,     FALSE,      NULL_PTR},
[..]
};
/*
 * Initial SMU module
 * */
void initSMUModule(void)
{
[..]
        /* Validation if PES configuration was successful */
    IfxScuWdt_clearSafetyEndinitInline(IfxScuWdt_getSafetyWatchdogPasswordInline());
    if(MODULE_SMU.AGC.B.PES != pesAction)
    {
        result = fail;
    }
    IfxScuWdt_setSafetyEndinitInline(IfxScuWdt_getSafetyWatchdogPasswordInline());
    g_SafetyKitStatus.smuStatus.smuCoreAlarmPESSetSts = result;
[..]
}\AppSw\SafetyKit\01_Smu\SMU\SMU.c
```

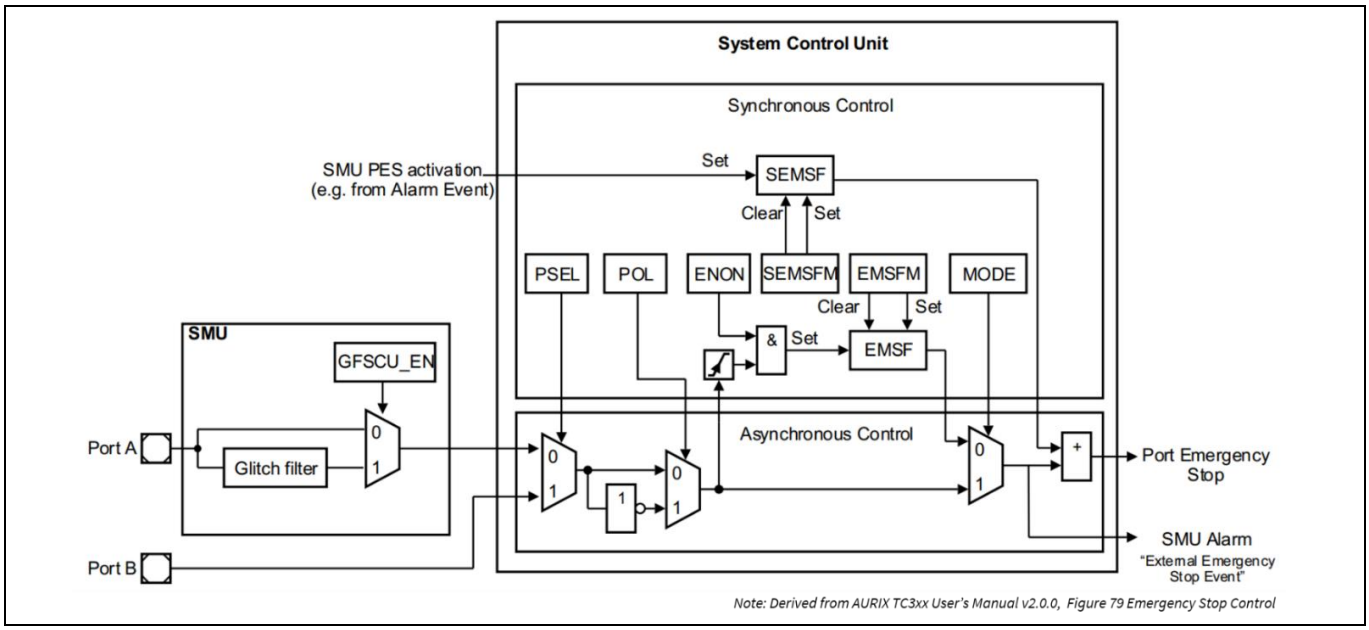


Figure 34 Emergency Stop control overview

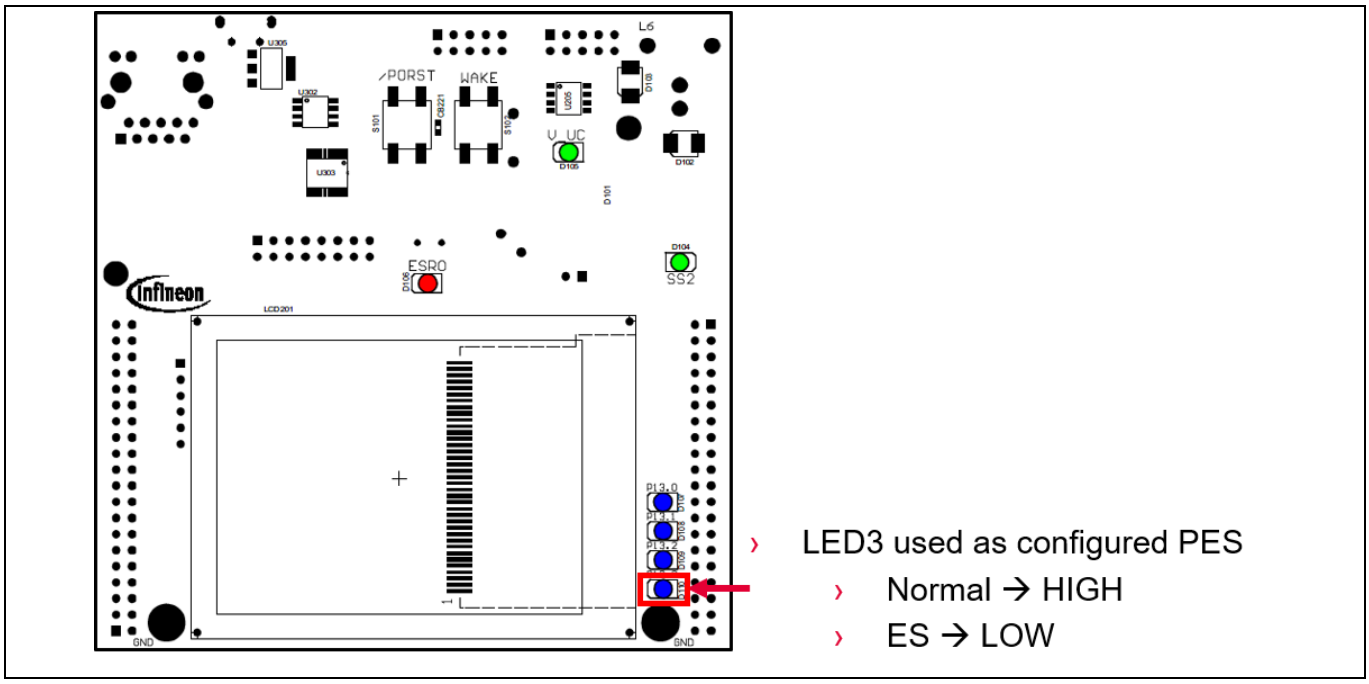


Figure 35 Emergency Stop status visualized with LED connected to Port Emergency Stop (PES)

Code Listing 29 Code snippet for ES submodule configuration

```
/*
 * Function to initialize the emergency stop
 * */
void initEmergencyStop(void)
{
    uint16 cpuWatchdogPasswd = IfxScuWdt_getCpuWatchdogPassword();

    /* PORT configuration for ES */
    IfxPort_setPinModeOutput(EMERGENCY_STOP_LED_PIN.port, EMERGENCY_STOP_LED_PIN.pinIndex, \
                             IfxPort_OutputMode_pushPull, IfxPort_OutputIdx_general);
    IfxPort_setPinLow(EMERGENCY_STOP_LED_PIN.port, EMERGENCY_STOP_LED_PIN.pinIndex);

    /* Clear CPU Endinit to enable configuration */
    IfxScuWdt_clearCpuEndinit(cpuWatchdogPasswd);

    /* Enable ES function for EMERGENCY_STOP_LED_PIN (LED3) */
    EMERGENCY_STOP_LED_PIN.port->ESR.U |= 1 << EMERGENCY_STOP_LED_PIN.pinIndex;

    /* SET CPU Endinit */
    IfxScuWdt_setCpuEndinit(cpuWatchdogPasswd);
}

/*
 * function clear the emergency stop flags
 * */
void clearEmergencyStopFlags(void)
{
    MODULE_SCU.EMSSW.B.EMSFM = 2; /* 0b10 */
    MODULE_SCU.EMSSW.B.SEMSFM = 2; /* 0b10 */
} \AppSw\SafetyKit\02_Safety_Mechanisms\SafetyKit_EmergencyStop.c
```

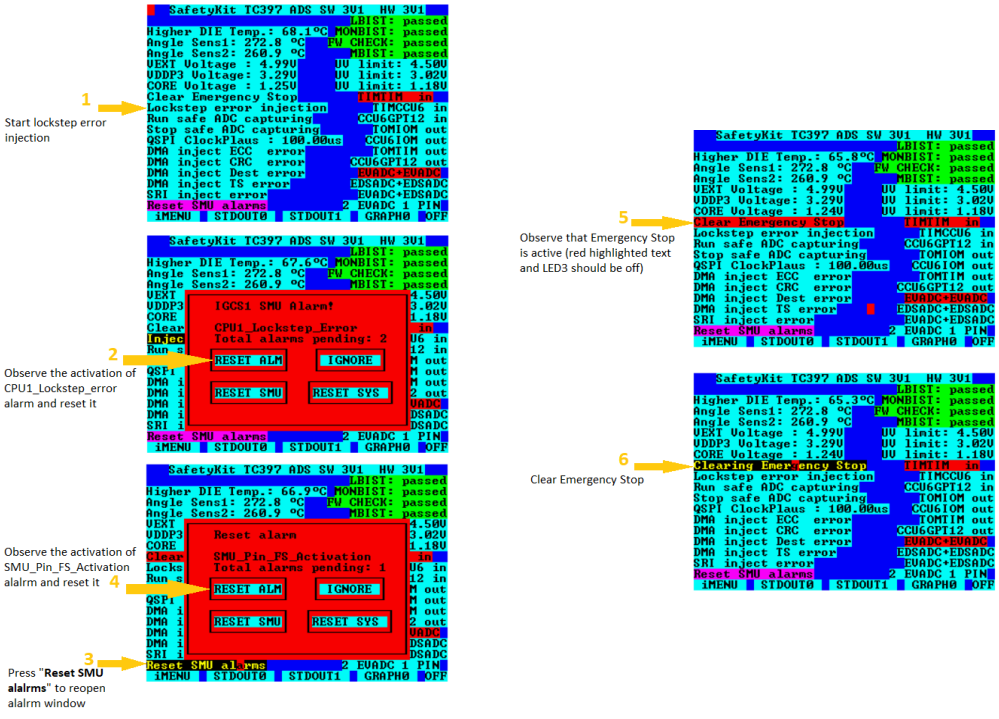


Figure 36 Lockstep fault injection to demonstrate ES activation

Safety watchdog

The safety watchdog timer has two main use cases:

- As a system-level periodic watchdog timer, which needs to be serviced to avoid an overflow and therefore, an SMU alarm request
- To protect safety-critical system registers from unintended write access with a safety ENDINIT sequence.

The safety ENDINIT must be cleared to open a small time window for configuration of these registers; after the configuration, the application software must set the ENDINIT again to avoid an SMU alarm. Write accesses to these registers without the clearing safety ENDINIT results in an SMU alarm request. The safety watchdog implements the *Safety Mechanism SAFETY_WATCHDOG*. Each set of safety ENDINIT protected registers is listed in its own safety mechanism: for example, *Safety Mechanism CLOCK:SAFETY_ENDINIT*.

Attention: Buttons and switches should be operated sequentially on the Safety Board because short-circuits will appear when enabling simultaneously a strong pull-up and a strong pull-down circuitry on the same line. This could permanently damage the demo setup. If such an event occurs, the board can also enter a safe mode due to the overtemperature condition. In such a case, the board is temporarily unusable (but not damaged) for approximately one minute.

Safety Kit implementation of the safety watchdog service with fault injection

The safety watchdog is serviced every millisecond from the CPU3 STM interrupt routine. By pressing the Stall CPU button (see Figure 37), every CPU except the CPU which handles SMU alarms, will be trapped in a `while` loop inside a high-priority interrupt routine. This prevents the triggering of the STM interrupt routine and therefore, the servicing of the safety watchdog timer.

Figure 37 Stall CPU button to trigger watchdog timeout alarm

As already explained in Section 4.2.1, the WDT timeout alarms require special processing. Therefore, a watchdog timeout alarm is configured to trigger an NMI (see Code Listing 20) and to start the recovery timer. On triggering of the NMI, the code execution will jump to the callout function `safetyKitNmiCallout` (see 0). Inside the callout function, a check is done which detects if the NMI is called because of an SMU alarm. Afterwards, it will be verified inside the `detectAlarmSourceSMU` function; if it was triggered by an alarm, which is also intended to trigger the NMI and which is also not pending. If this is the case, the recovery timer will be stopped, and code execution can continue. This means that if the STALL_CPU button is pressed the first time, the expected System Control Unit (SCU) watchdog timeout alarm will be triggered and the corresponding NMI message (see Figure 38) will be displayed.

If the **STALL_CPU** button is pressed a second time (without resetting of the alarm with the **RESET_ALM** button on the TFT), the recovery timers will not be stopped again because the alarm is still pending (refer to the `detectAlarmSourceSMU` function). The recovery timer overflow is configured to result in a reset request (see Code Listing 20).

A reset can be avoided by clearing the SCU watchdog alarm with the **RESET_ALM** button on the TFT.

Code Listing 30 Code snippet for SMU configuration of watchdog timeout alarm

```
#define SMU_ALARM_WHICH_TRIGGERS_NMI   IfxSmu_Alarm_SCU_Watchdog_TimeOut

const alarm_config_struct global_alarm_config[USER_ALARM_NUMBER] =
{
[..]
    /*------------------------------------------------------- NMI ----------------------------------
--------------------*/
    {SMU_ALARM_WHICH_TRIGGERS_NMI,    IfxSmu_InternalAlarmAction_nmi,   FALSE,    TRUE,
&SafetyKitWatchdogAlarmHandling},

    /*------------------------------------------------------- RESET --------------------------------
--------------------*/
    {IfxSmu_Alarm_SMU_Timer0_TimeOut, IfxSmu_InternalAlarmAction_reset, FALSE,    FALSE,    NULL_PTR},
    {IfxSmu_Alarm_SMU_Timer1_TimeOut, IfxSmu_InternalAlarmAction_reset, FALSE,    FALSE,    NULL_PTR},
[..]
};

void initSMUModule(void)
{
[..]
        /* Enable and configure the recovery Timer (maximum value for the duration is 0xffffff) */
    enableRecoveryTimerSMU(0xffffff);
[..]
}
```

\AppSw\SafetyKit\01_Smu\SMU\SMU.c

Code Listing 31 Code snippet for NMI callout safetyKitNmiCallout

```
void safetyKitNmiCallout(IfxCpu_Trap trapWatch)
{
    /* Check if NMI was called due to an SMU alarm */
    if ((IfxSmu_getAlarmExecutedStatus(IfxSmu_AlarmExecutionStatus_nmi)) &&
        (IfxScuCcu_getTrapStatusFlag(IfxScuCcu_Traprequest_smu)))
    {
        /* Implement appropriate reaction */
        {
            sint16 nbrRaisedAlarm;
            /* Check if an alarm, which is intended to trigger an NMI has been triggered, if yes stop re-
covery timer */
            /* Note: if the alarm is triggered a second time before it got reset via the TFT, the function
             * detectAlarmSourceSMU() will not stop the recovery timer. Therefore, when the Stall CPU but-
ton is pressed a
             * second time, the recovery timer will timeout and hence it will trigger a reset. */
            nbrRaisedAlarm = detectAlarmSourceSMU(&alarmsThatTriggerNMI[0], nbrAlarmsThatTriggerNMI);
            if (nbrRaisedAlarm > 0)
            {
#if USE_SAFETYKIT_TFT
                popMessage = "NMI ACTIVATED!";
                conio_driver.dialogmode = SHOWSMUALARM;
#endif /* USE_SAFETYKIT_TFT */
                /* ASC_SHELL print is executed from endless background task */
            }

            /* Clear alarm executed status bit*/
            IfxSmu_clearAlarmExecutedStatus(IfxSmu_AlarmExecutionStatus_nmi);

            /* Clear SMU Alarm Trap Request Flag */
            IfxScuCcu_clearTrapStatusFlag(IfxScuCcu_Traprequest_smu);
        }
    }
} \AppSw\SafetyKit\01_Smu\SMU\SMU.c
```

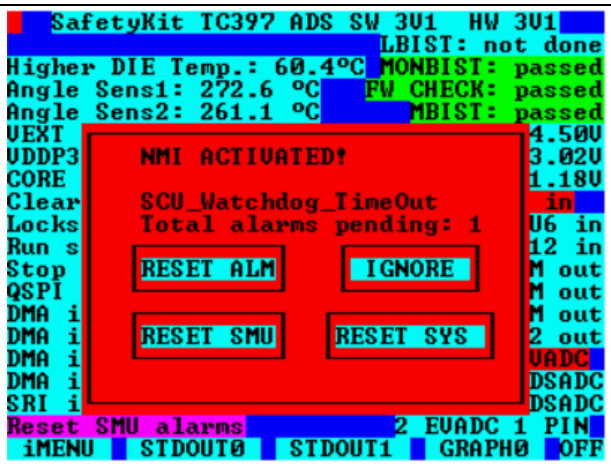


Figure 38 NMI alert because of watchdog timeout alarm

Note: For more information on watchdog timeout alarms and the recovery timer, see Section 0.

6.2.8.5 Standby Controller (SCR)

The SCR is an XC800 8-bit microcontroller that runs during the standby mode of the MCU. Its main function is to control the wakeup signals and some I/O pads (marked with SCR) during the MCU standby period.

For more information, see the “SCR” section in the AURIX™ TC3xx Safety Manual [3].

6.2.8.6 Die Temperature Sensor (DTS)

Proper AURIX™ MCU operation is guaranteed within the *Operating Conditions* listed in the datasheet, which includes limits for ambient temperature and junction temperature. There are two die temperature sensors (DTS) available on the AURIX™ microcontroller to help applications approximate the junction temperature: a Power Management System (PMS) DTS located near the regulators, and a core DTS located closer to the CPU cores. These two sensors are part of *Safety Mechanism DTS:TEMPERATURE_MONITOR*. If either DTS exceeds the limit defined in its DTSLIM register, an SMU alarm is raised as defined in the Safety Manual:

- ALM8[30] SMU alarm for die temperature underflow in the core domain
- ALM8[31] SMU alarm for die temperature overflow in the core domain
- ALM9[0] SMU alarm for die temperature overflow in the PMS domain
- ALM9[1] SMU alarm for die temperature underflow in the PMS domain

The Safety Manual specifies that the difference between the PMS and core DTS should not exceed 9°C. This check is not performed in hardware, so this periodic checking should be part of the application. An SMU software alarm should be raised if the temperature difference exceeds the value specified in the Safety Manual.

Application Kit Safety implementation of the die temperature sensor comparison

Both temperatures are compared to each other. If the difference exceeds 9°C, an SMU software alarm will be triggered.

Code Listing 32 Code snippet for the Safety Mechanism: DTS_RESULT implementation

```
void dtsMeasurementISR(void)
{
[..]
    float32 dieTemperaturePms;
    float32 dieTemperatureCore;

    /* Get the new PMS die temperature measurement */
    dieTemperaturePms = IfxDts_Dts_getTemperatureCelsius();

    /* Also get the CORE die temperature from DTSCSTAT register of SCU module */
    dieTemperatureCore = IfxDts_Dts_convertToCelsius((uint16)MODULE_SCU.DTSCSTAT.B.RESULT);

    /* SM:DTS_RESULT */
    /* Calculate the absolute value of the temperature difference, trigger SMU software alarm
if value is above limit. */
    float32 dieTempDifference;
    dieTempDifference = ( dieTemperatureCore > dieTemperaturePms) ?
                          dieTemperatureCore – dieTemperaturePms :
                          dieTemperaturePms - dieTemperatureCore;
[…]

    if(dieTempDifference > MAX_DIE_TEMP_DIFF)
    {
        SMU_SMU_core_SW_alarm_trigger(SOFT_SMU_ALM_DTS);
    }
}
\AppSw\SafetyKit\02_Safety_Mechanisms\SafetyKit_DieTemp.c
```

6.2.9 MCU function – Interfaces

The MCU interface designates all the modules that allow communication with off-chip devices:

- ERAY
- GETH
- HSSL
- MCMCAN
- QSPI
- PORT
- PSI5
- PSI5S
- SENT
- I2C
- ASCLIN
- MSC
- EBU
- SDMMC
- CIF

The AURIX™ TC3xx MCU does not implement hardware safety mechanisms for those modules, except for the hardware safety mechanisms that are common to all SRAM blocks. Multiple SMs are created to ensure safe communication. The application software must implement safe communication, which detects random hardware faults, systematic faults, and interference, which can lead to message corruption and protocol violation, including temporal and random faults of safety-related data transferred over the interface. When a failure is detected, the application software must trigger a response. The idea is similar for every module. Depending on the possibilities available, the following measures must be implemented:

- Insert redundancy on communication level (CRC and data ID)
- Frame counter
- Timeout monitoring
- Hardware redundancy (for example, two or more independent channels acquiring data from one or more sensors)
- Loopback (that is, reading back an output to detect a fault)

This application note only demonstrates the QSPI, PORT, and SENT interfaces.

6.2.9.1 Queued Synchronous Peripheral Interface (QSPI)

The main purpose of the QSPI module is to provide synchronous serial communication with external devices using Clock, Data In, Data Out, and Slave Select signals. The focus of the module is set to fast and flexible communication: either point-to-point or master-to-many slaves communication. For details of the QSPI module, see the AURIX™ TC3xx User Manual [1].

To ensure the safety and integrity of the data transmitted over QSPI, the application SW is responsible to implement safe communication SM (*Safety Mechanism QSPI:SAFE_COMMUNICATION* according to AURIX™ TC3xx Safety Manual. Thus, the SM consists of appending the payload data, data ID, a frame counter, and CRC base on

payload data of the frame on a frame. These frames should be sent regularly by the QSPI master and received independently by the QSPI slaves. Once a frame is received, safety mechanisms are activated to ensure the integrity of the data. In parallel, a timeout and a time checking mechanism are also activated.

Safety Kit implementation

In Application Kit Safety, there are two different QSPI instances connected to each other using the loopback concept. The QSPI4 instance works as a master, while the QSPI3 module is in slave mode. In the initialization function, data is filled into the master buffer. The communication is based on sending data from the master to the slave. This communication, frame length, and pin connection are shown in Figure 39.

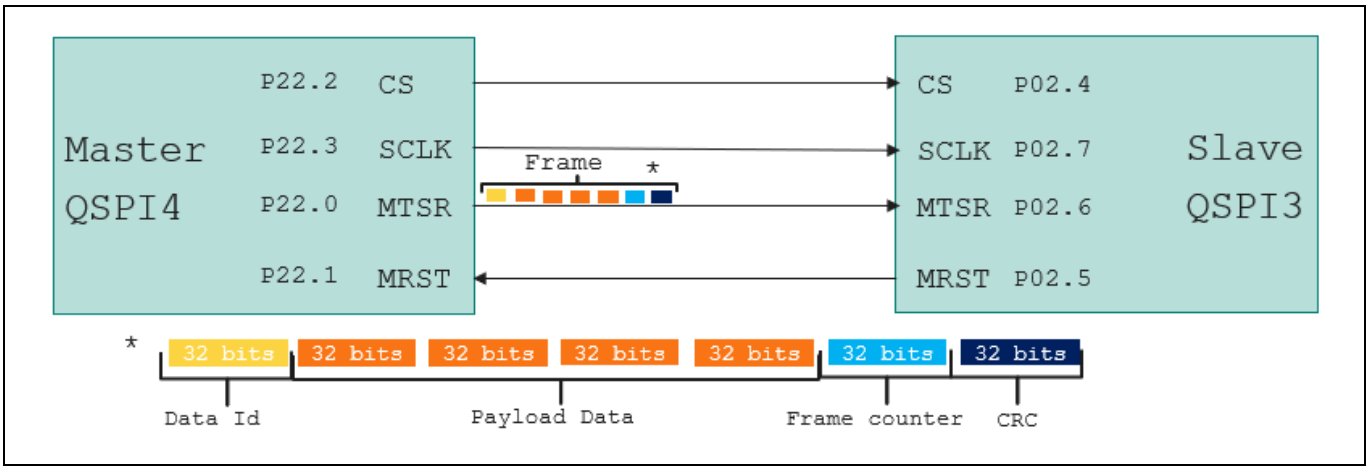


Figure 39 QSPI communication protocol

To use this QSPI configuration, the four switches must be in downward direction as shown in Figure 40.

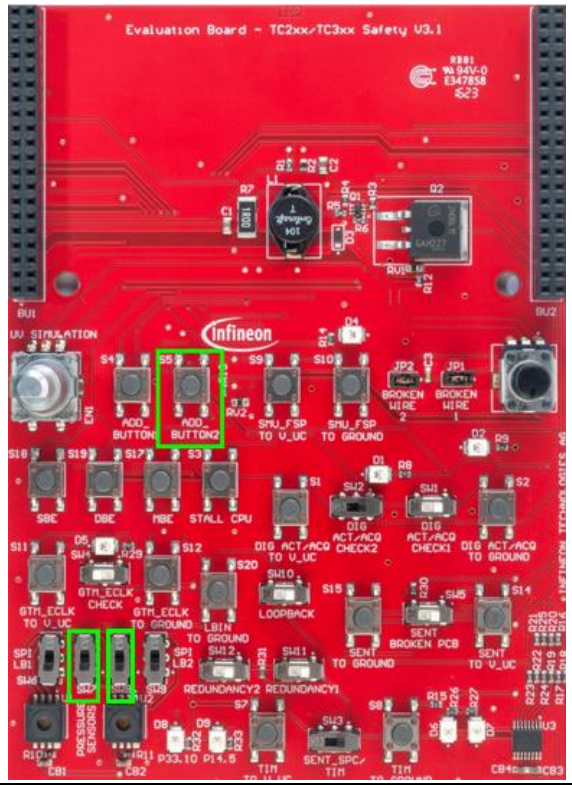


Figure 40 QSPI error injection

The data is divided into four frames, each of which contains four 32-bit elements. Four of these elements correspond to the payload data, while the remaining three are data ID, a frame counter, and a CRC element. The CRC element is calculated based on the payload data. To calculate the CRC, the TriCore™ CPU instruction CRC32 is used. Note that CRC calculation is performed on the slave side only, because the CRC and frame counter the master transmits are pre-computed and stored in corresponding buffer.

The master device runs on CPU2 and sends the frames to the slave device every 100 ms without looking at what was received on the slave side (with the `masterTransmitDataSafeCommQSPI()` function in *SafetyKit_Main.c*).

When the slave device detects in its Rx interrupt that the master has sent all the data of each frame, the frame receive flag is enabled and the communication safety measures (data validation) related to the frame are executed. In case of any errors, such as mismatching CRC or Frame Counter (FC), the system raises an alarm. For both, there is recalculation done on the slave side and compared with the data sent by the master side (see *08_Ext_Communication/SafetyKit_QSPI_Safe_Communication.c*). Then the slave is configured to receive the next frame.

The detection of incorrect timings and timeouts are managed by another independent safety measure that is triggered every 100 ms. (see the `salveTimingValidation()` function in *SafetyKit_Main.c*). The slave side running on CPU3.

Error injection

To simulate these errors, the ADD_BUTTON2 push button highlighted in Figure 40 can be used. When the button is first pressed, an early frame error is triggered; when it is pressed again, a late frame error is injected. When the button is pressed for the third time, a normal frame will start. When the button is pressed to trigger an early or late frame, frames are sent too early (every 90 ms instead of 100 ms) or too late (every 110 ms

instead of 100 ms) on the master side respectively. If an error is detected, the communication is not stopped; the user has the information that an error is detected and can choose how to manage this error on the TFT display.

To trigger a timeout error, the Chip Select need to be disconnected by switching SW7 shown in Figure 40 in the upward direction. In this case, the slave stops receiving the data sent by the master, which in turn leads to an overflow of the timer counter.

The data corruption can be triggered by disconnecting SW8; it means that the data is not sent via MOSI, even though the Slave Select and CLK are connected and the slave buffer is filled with the 0x0 value. Thus, the data does not match, resulting in the data corruption alarm appearing. For each of the errors detected, the Safety Kit can generate a different output error message using the same software alarm (ALM10).

If there is any data mismatched (CRC, Data ID, or frame counter), the corresponding software alarm (frame corrupt message) will appear. While the LED D9 should be toggling every 100 ms which indicates that the QSPI communication is working fine. To inject the error, change the direction of any switch as shown in Figure 39; LED D9 will not be toggling, which indicates that the data sent is not received via QSPI communication and therefore, an alarm will appear.

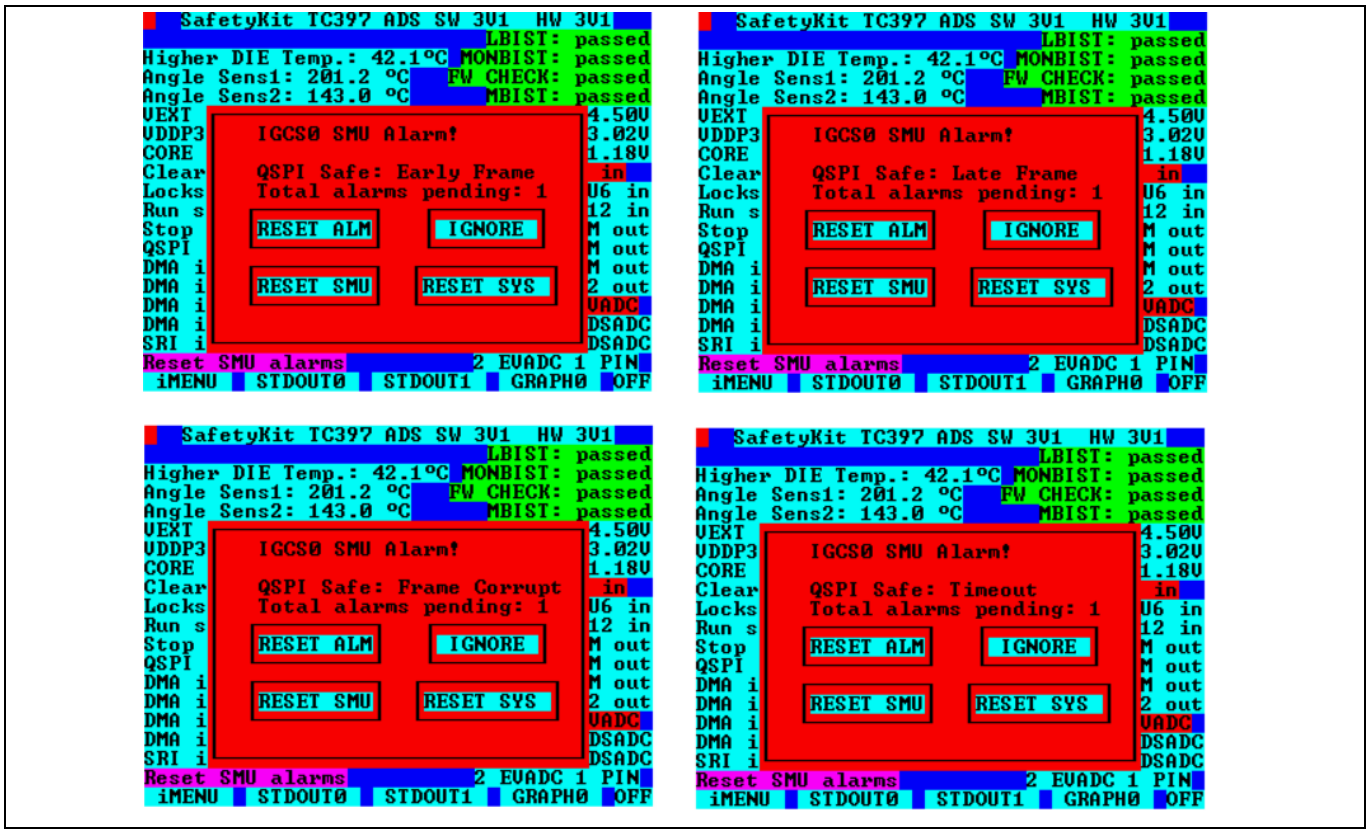


Figure 41 QSPI safe communication alarm

Note: Safety Mechanism ERAY:SAFE_COMMUNICATION, Safety Mechanism GETH:SAFE_COMMUNICATION, Safety Mechanism HSSL:SAFE_COMMUNICATION, and Safety Mechanism MCMCAN:SAFE_COMMUNICATION are not implemented but the idea is the same as of Safety Mechanism QSPI:SAFE_COMMUNICATION.

6.2.9.2 PORT

When using the port for digital or analog applications, two SMs are recommended, depending on the direction of the signal. For safe capture, it is recommended to use a redundant port (*Safety Mechanism PORT:REDUNDANCY*) with one or more signal sources for the same information. For a safe signal emission, a port loopback (*Safety Mechanism PORT:LOOPBACK*) should be used to monitor the emitted signal.

For hardware redundancy, see the application note AP32405. This application note provides the guidance to avoid common-cause failures at the port level by selecting the redundant port to use in a valid region considering the region used for the monitored port. For each package, the application note provides a map of the package region (designated as a group), and a truth table of the redundant port selection compared to the mission port selected. “True” implies that this region is safe to choose for a redundant port compared to the mission port region.

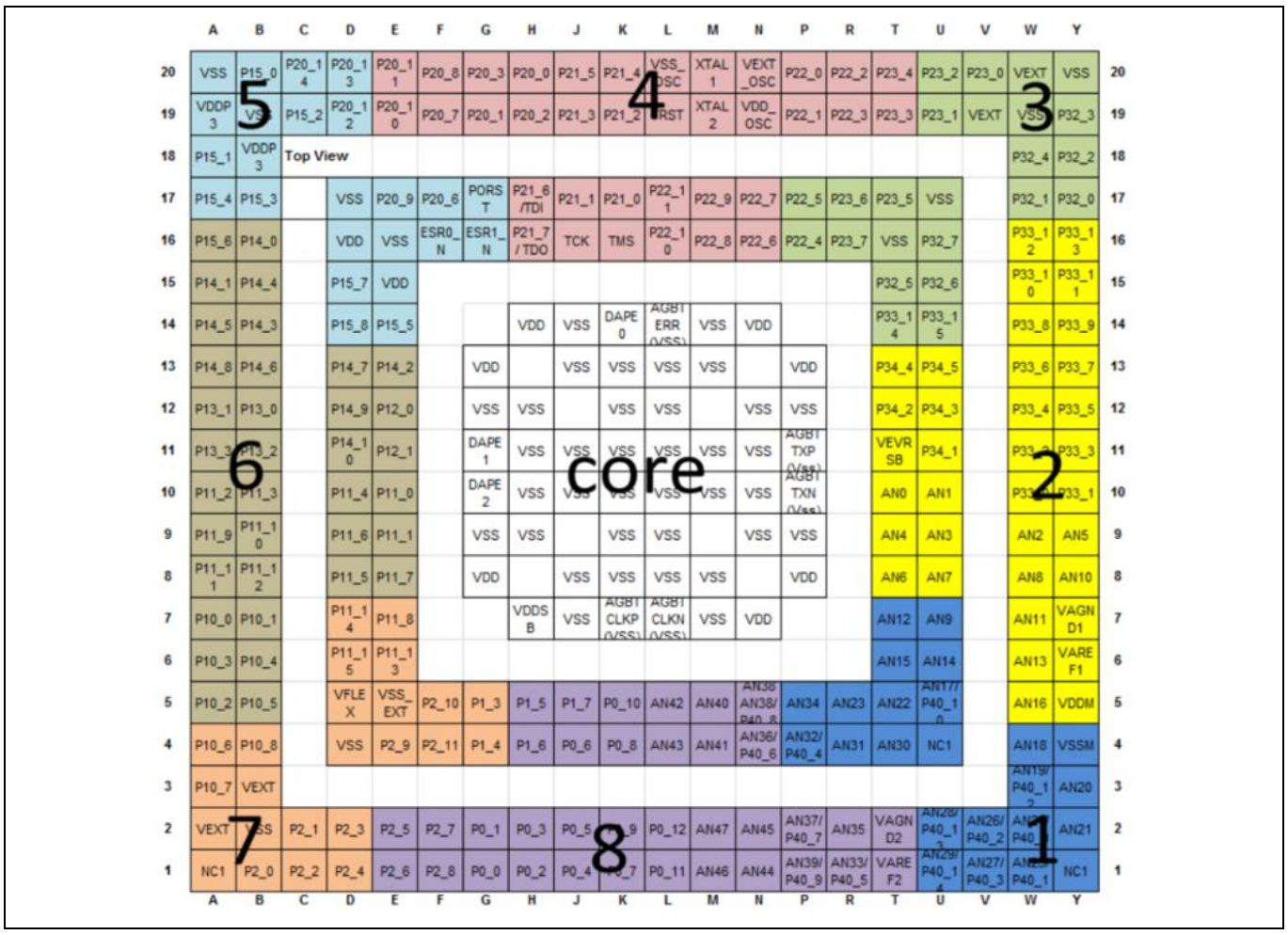


Figure 42 TC397 I/O ball groups in the LFBGA292 package

NET B \ NET A	Group 1	Group 2	Group 3	Group 4	Group 5	Group 6	Group 7	Group 8	Core
Group 1	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
Group 2	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
Group 3	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE
Group 4	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE
Group 5	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE
Group 6	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE
Group 7	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
Group 8	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
Core	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Figure 43 Truth table with the valid I/O pin groups for the mission and monitor signals corresponding to the ball-out groups in the LFBGA292 package

Safety Application Kit implementation

The Safety Application Kit contains implementation of *Safety Mechanism PORT:LOOPBACK* and *Safety Mechanism PORT:REDUNDANCY*.

For these two SMs, the Safety Manual V2.0 stipulates that that application SW must configure and use redundant GPIO communication to detect faults (when receiving information) or allow the receiver to detect faults (when transmitting information). An appropriate response for a fault is application-dependent.

The PORT:LOOPBACK safety mechanism is implemented because S1 as mission signal on Px.y (P00.4) works as the output and is given to the outer world. The same signal is looped back to the S2 monitor signal and given back to Pa.b (P33.5) pin as the input. The application SW then compares both values and respond accordingly.

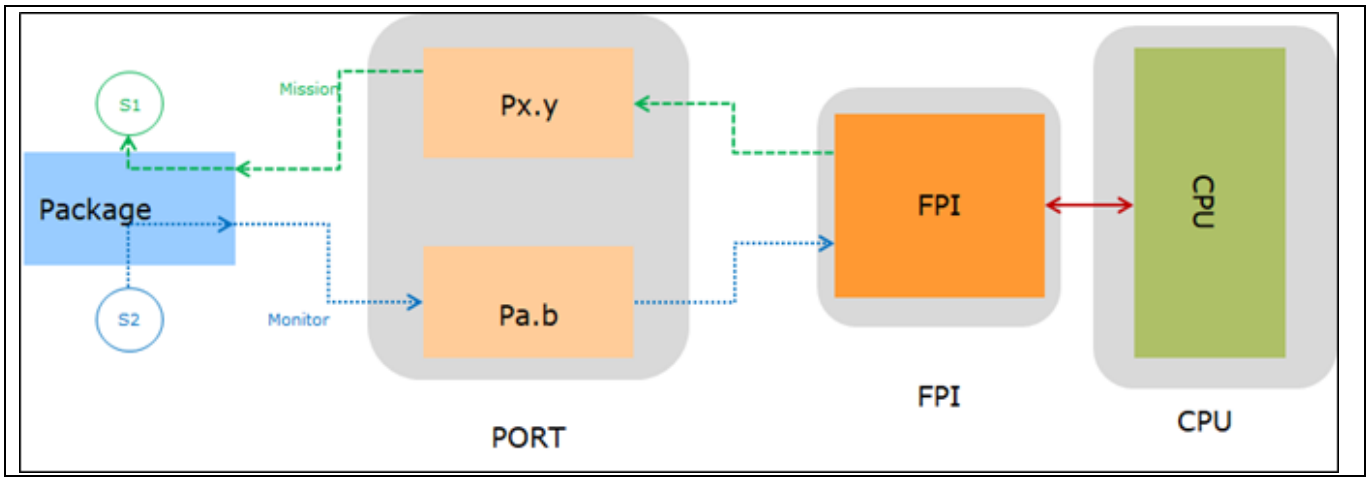


Figure 44 Safety Mechanism PORT:LOOPBACK overview

Note that the direction of SW10 as shown in Figure 45 must be toward the right, where it means that S1 is looped back to S2.

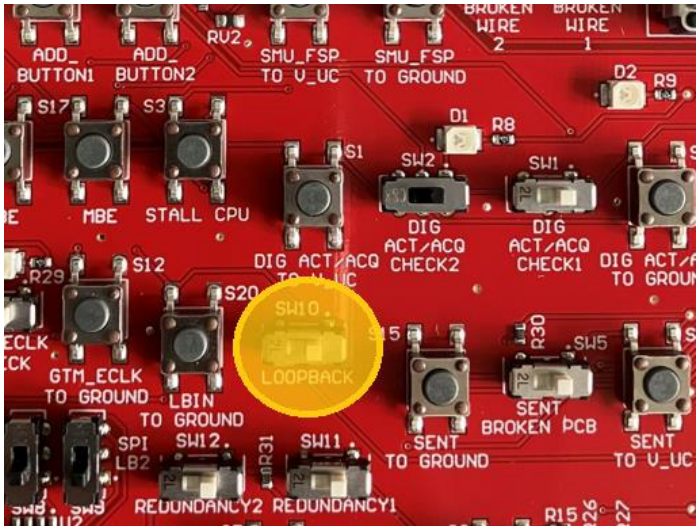


Figure 45 Port loopback switch on the Safety Evaluation Board

The fault can be injected via SW10. If SW10 is towards the right, it means that the S1 is looped back on S2, but when you change the direction of SW10 to the left, it means that S1 is no more connected to S2 (the value coming on S1 is not given on S2). Therefore, the compare result in the application SW will trigger an alarm. On the display, you will see the following alarm:

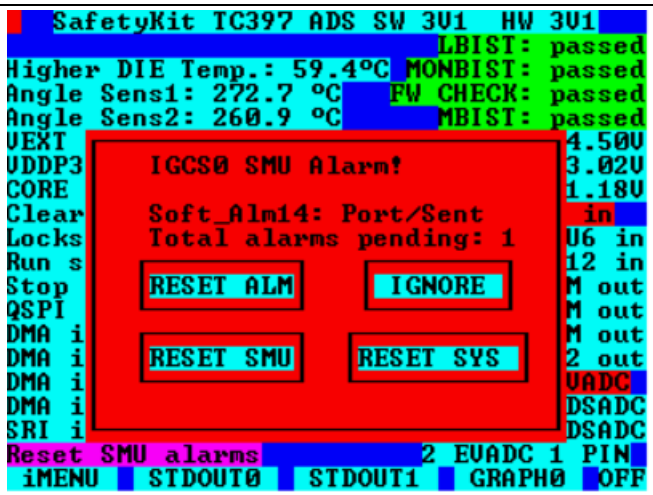


Figure 46 SMU alarm popup for port SMs

Safety Mechanism PORT:REDUNDANCY is also implemented in the Safety Application Kit. From Safety Manual V2.0, the assumed scenario (B) is implemented: the application SW receives a safety-critical input value (called signal S1 “Mission”) on pin Px.y (P02.3) from external sender (0 or 5 V in the safety case). To detect faults on S1, it receives on pin Pa.b (P23.2) the same information on signal S2 “Monitor”. The information on signals S1 and S2 is cyclically compared. The scenario is illustrated as follows:

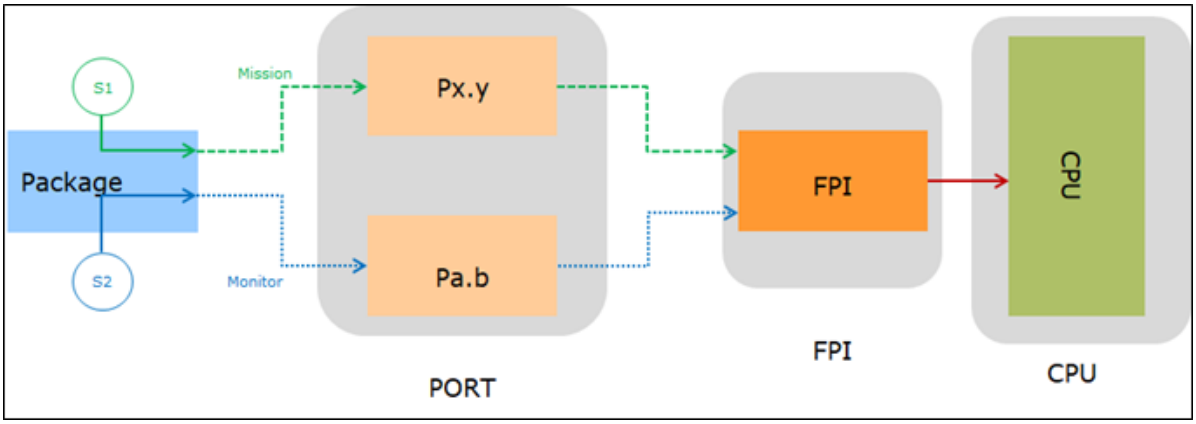


Figure 47 Safety Mechanism PORT:REDUNDANCY overview

In the safety application, SW11 (REDUNDANCY1) and SW12 (REDUNDANCY2) work as S1 and S2 respectively. If SW11 is on towards the right, it means that S1 is connected to VCC; if the direction is to the left, it means it is connected to GND. The same applies to SW12 as well. Therefore, to inject an error, either SW direction should be the opposite i.e., if SW11 is to the left, SW12 must be on the right and vice versa. This means that SW11 is connected to VCC while SW12 is connected to GND. If the application SW compares the result, an SMU alarm will pop up as shown in Figure 46.

Note: Due to limited number of software alarms, the same alarm is used for Port (loopback and redundancy) and as well for SENT SMs.

6.2.9.3 Single Edge Nibble Transmission (SENT)

The Single Edge Nibble Transmission (SENT) module communicates with the external world via one I/O line for each channel. This module supports a Short PWM Code (SPC) protocol, which enhances the standardized SENT protocol defined by SAE J2716 042016. SPC enables the usage of enhanced protocol functionality due to the ability to select between “synchronous”, “range selection”, and “ID selection” protocol modes, or even “bidirectional transmit mode”. For the Safety Application Kit use case ID selection mode, up to four sensors are selected on a bus (bus mode, 1 master with up to 4 slaves). This allows parallel connection of up to four sensors using only three lines (VDD, GND, OUT) as illustrated in the following. For more detailed description of SENT Module, see the AURIX™ User’s Manual.

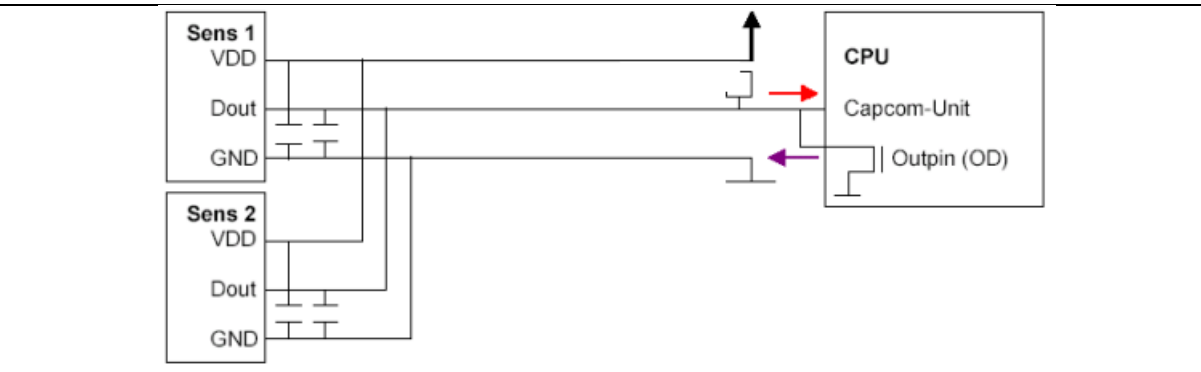


Figure 48 SPC ID selection mode

Safety Manual V2.0 includes the *Safety Mechanism SENT:CHANNEL_REDUNDANCY*, which states that the application SW must use two or more independent SENT channels for acquiring data from one or more sensors. Upon receiving a new value, the results captured by the independent channels must be compared by the application W. If the application SW detects a mismatch, the application SW must trigger an appropriate response.

Safety Kit implementation

In Application Kit Safety, a dual-die TLE5012 GMR angle sensor is used, where two sensors are built-in. As two redundant sensors are used by two redundant SENT channels, SPC triggers from the Generic Timer Module (GTM) should be redundant to avoid common-cause failures. Figure 49 illustrates the SENT module implemented redundantly to read the value from two sensors. Both redundant channels (mission “S1” and monitor “S2”) are identical; the mission signal is in green color while the monitor signal is blue colored. In the safety software, the following pins are used:

- P00.5 (SENT4B IN and SPC4 OUT) as S1
- P14_6 (TOM2_6N, TOUT86) as GTM trigger for S1
- P00.10 (SENT9B IN and SPC9 OUT) as S2
- P20.0 (TOM0_6, TOUT59) as GTM trigger for S2

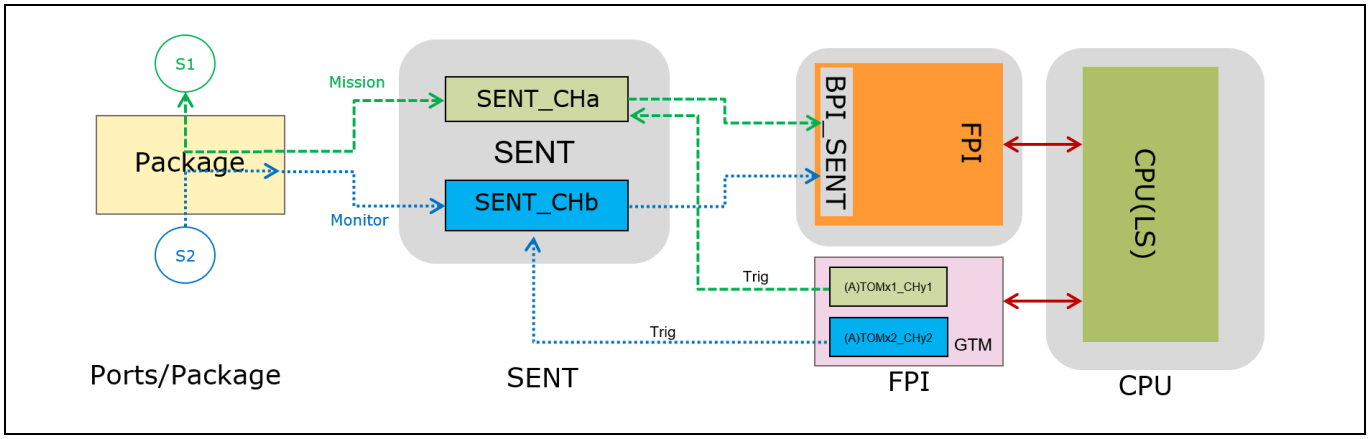


Figure 49 Safety Mechanism SENT:CHANNEL_REDUNDANCY

Application Kit Safety has some constraints to find the SENT/SPC pins to avoid common-cause failures. Figure 50 contains different pin combinations where common-cause failure can be avoided.

ok Can be used as signal pairs x DRC violation

			G2	H2	J1	J2	J4	K1	K4	K2	K5	F2
Ball	**Pin**	**Symbol**	P00.1	P00.3	P00.4	P00.5	P00.6	P00.7	P00.8	P00.9	P00.10	P02.7
G2	P00.1	SENT_SPC0										
H2	P00.3	SENT_SPC2										
J1	P00.4	SENT_SPC3										
J2	P00.5	SENT_SPC4										
J4	P00.6	SENT_SPC5										
K1	P00.7	SENT_SPC6										
K4	P00.8	SENT_SPC7										
K2	P00.9	SENT_SPC8										
K5	P00.10	SENT_SPC9										
F2	P02.7	SENT_SPC1										

Figure 50 SENT SPC pin selections

The GTM TOM used to trigger the SPC pulse runs at 100 Hz; on every PWM pulse rising edge, the SPC pulse on length according to the sensor ID (Application Kit Safety using the same ID for both sensors but with different GTM pulse and SPC pulse) is generated. The sensor then replies according to the pulse; with RDI/RSI interrupt and the sensor data available, the angle is accordingly calculated. The application SW compares the data from both sensors, checks the boundary conditions, and responds accordingly. The application SW is shown in **Error! Reference source not found.**.

Code Listing 33 Safety Mechanism SENT:CHANNEL_REDUNDANCY

```
/*
 * Init SENT and GTM module
 */
void initTLE5012Modules()
{
    /* Initial SENT module in SPC mode for TLE5012 */
    initSENTwithSPCMode();

    /* Initial GTM TOM module for cyclic triggering of SPC Pulse */
    initTriggerInputGTMTOM();
}
/*
 * Function to check SENT redundancy
 * SM:SENT:CHANNEL_REDUNDANCY
 *  */
void checkRedundancySENT()
{
[…]

        if ((g_SafetyKitStatus.sentRedundancy.angleMaximumThreshold > THRESHOLD_VALUE) ||
            (g_AppSENT.sentCrcCalculated[0] != g_AppSENT.sentCrcReceived[0])           ||
            (g_AppSENT.sentCrcCalculated[1] != g_AppSENT.sentCrcReceived[1]))
        {
            if(alarmCount > 5)
            {
                 /* It is commented because most of the time magnet is not on the
top of the sensor and hence this alarm will pop up every time. User can uncomment
and see the reaction on tft display */
                /* softwareCoreAlarmTriggerSMU(SOFT_SMU_ALM_PORT_SMs); */
                 alarmCount = 0;
            }
            alarmCount++;
        }
    }
} \AppSw\SafetyKit\07_Sensor_Acq\SafetyKit_SentChannel_Redundancy.c
```

The real-time TLE5012 shown on the TFT display is highlighted in Figure 51. Apply the magnet provided with the AURIX™ Application Kit - TC3xx Safety package on the GMR sensor to view the change in angle value. The derivation between the two angles from the two sensors is also shown in Figure 51.

On the Evaluation Board, move SW5 (SENT broken PCB) toward the left. It means that sensor 2 is no more connected to the AURIX™ MCU; thus, the value of both sensors do not match. Correspondingly, an alarm is displayed. You can also simulate the conditions of SENT to GND and SENT to VCC with this board.

Note: The software alarm is commented out; if you don't have the magnet on the sensor, this alarm will be triggering every time. You can uncomment while placing the magnet on the sensor.

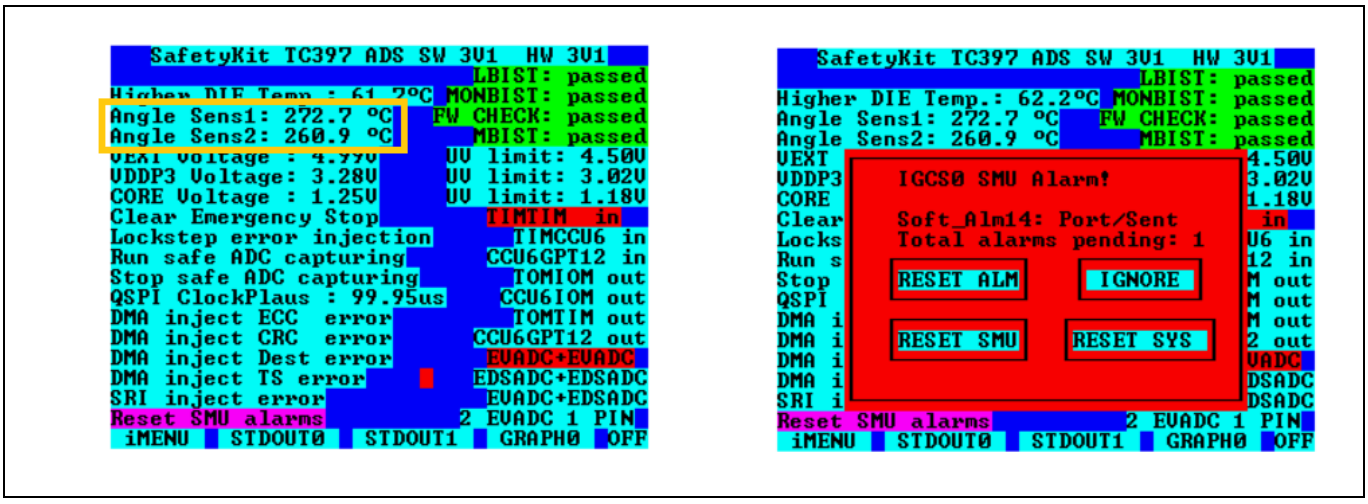


Figure 51 TLE5012 angle values (180° angle shift)

Once the angle difference deviates from the threshold value, the software alarm will pop up.

Note: The Safety Mechanism PSI5:CHANNEL_REDUNDANCY is not implemented but the idea is the same as of Safety Mechanism SENT:CHANNEL_REDUNDANCY. Due to the limited number of software alarms, the same alarm is used for Port (loopback and redundancy) and as well for SENT SMs.

6.2.10 MCU function – Analog acquisition

This section presents different use cases for the acquisition of safety-related input signals. Three different modules of the AURIX™ microcontroller are required: the converter control (CONVCTRL) module, the enhanced versatile analog-to-digital converter (EVADC), and the enhanced delta-sigma analog-to-digital converter (DSADC) module (see the AURIX™ TC3xx User Manual [1] for more information on the nominal functionality of these modules).

Using these modules, a total of five different use cases are presented in the Safety Manual; these are also covered by this application note. These use cases describe the use of the EVADC and the EDSADC for performing an acquisition of analog signals in different applications.

The following lists and Table 8 present an overview of all functional use cases mentioned in the AURIX™ TC3xx Safety Manual [3], and lists the functional blocks and SMs required for their implementation:

Analog acquisition function use cases:

- Functional Use Case **0** (FUC**0**): Analog acquisition with redundant EVADC channels
- Functional Use Case **1** (FUC**1**): Analog acquisition with redundant EDSADC channels
- Functional Use Case **2** (FUC**2**): Analog acquisition with one EVADC channel and one EDSADC channel
- Functional Use Case **3** (FUC**3**): Single analog acquisition with EVADC channels
- Functional Use Case **4** (FUC**4**): Single analog acquisition with one EDSADC and one EVADC channel

Table 8 Overview of SMs required for analog acquisition functional use cases

Functional block	SM needed	Required for FUC(s)
CONVCTRL	• *Safety Mechanism CONVCTRL:CONFIG_CHECK* (verifies whether the safety-relevant register is properly configured)	• Analog acquisition FUC**0** • Analog acquisition FUC**1** • Analog acquisition FUC**2** • Analog acquisition FUC**3** • Analog acquisition FUC**4**
EVADC	• *Safety Mechanism EVADC:PLAUSIBILITY* (compares values of mission and monitor channels + system-specific limit values check) • *Safety Mechanism EVADC:VAREF_PLAUSIBILITY* (verifies the ADC reference voltage) • *Safety Mechanism EVADC:DIVERSE_REDUNDANCY* (checks the redundancy at the module level) • *Safety Mechanism EVADC:CONFIG_CHECK* (verifies whether safety-relevant registers are properly configured)	• Analog acquisition FUC**0** • Analog acquisition FUC**2** • Analog acquisition FUC**3** • Analog acquisition FUC**4**
EDSADC	• *Safety Mechanism EDSADC:PLAUSIBILITY* (compares values of mission and monitor channels + system-specific limit values check) • *Safety Mechanism EDSADC:VAREF_PLAUSIBILITY* (verifies the ADC reference voltage) • *Safety Mechanism EDSADC:DIVERSE_REDUNDANCY* (checks the redundancy at the module level)	• Analog acquisition FUC**1** • Analog acquisition FUC**2** • Analog acquisition FUC**4**

The system integrator should also take care of the common-cause failures (CCF) when selecting the redundant port pins (see Section 6.2.9.1 for more information). See Section 6.2.10.2 for the Application Kit Safety analog acquisition use cases example.

Additional safety features

The EVADC implements test features, which may be used in a safety-related application to check the proper routing of the signal from the source to the EVADC kernel:

- **Converter Diagnostics** connects an internally generated, defined signal to the converter to test the proper operation of the converter
- **Multiplexer Diagnostics** connects a weak pull-up or pull-down device to an input channel to test the correct operation of the internal analog input multiplexer. A subsequent conversion can then confirm the expected modified signal level. Multiplexer diagnostics can be enabled for channels CH1 and CH2 of each group.
- **Pull-Down Diagnostics** connects a weak pull-down device to an input channel to test the external connection to a sensor. The strong pull-down can be used to discharge an external buffer capacitor.

- **Broken-Wire Detection** preloads the converter network with a selectable level before sampling the input channel to test the proper connection of an external analog sensor to its input pin. The result will then reflect the preloaded value whether the input signal is no longer connected. If buffer capacitors are used, a certain number of conversions may be required to reach the failure indication level. The broken-wire detection can be enabled for each channel separately: see “Single analog acquisition with one EDSADC and one EVADC channel (FUC3) and additional broken-wire detection” in Section 6.2.10.2.

6.2.10.1 Overview of analog acquisition implementation

The Safety Evaluation Add-on Shield board features multiple analog Input signals, which can be sampled with the EVADC and/or the EDSADC module. These analog inputs signals and circuitry can be seen in Figure 52 and in Table 9.

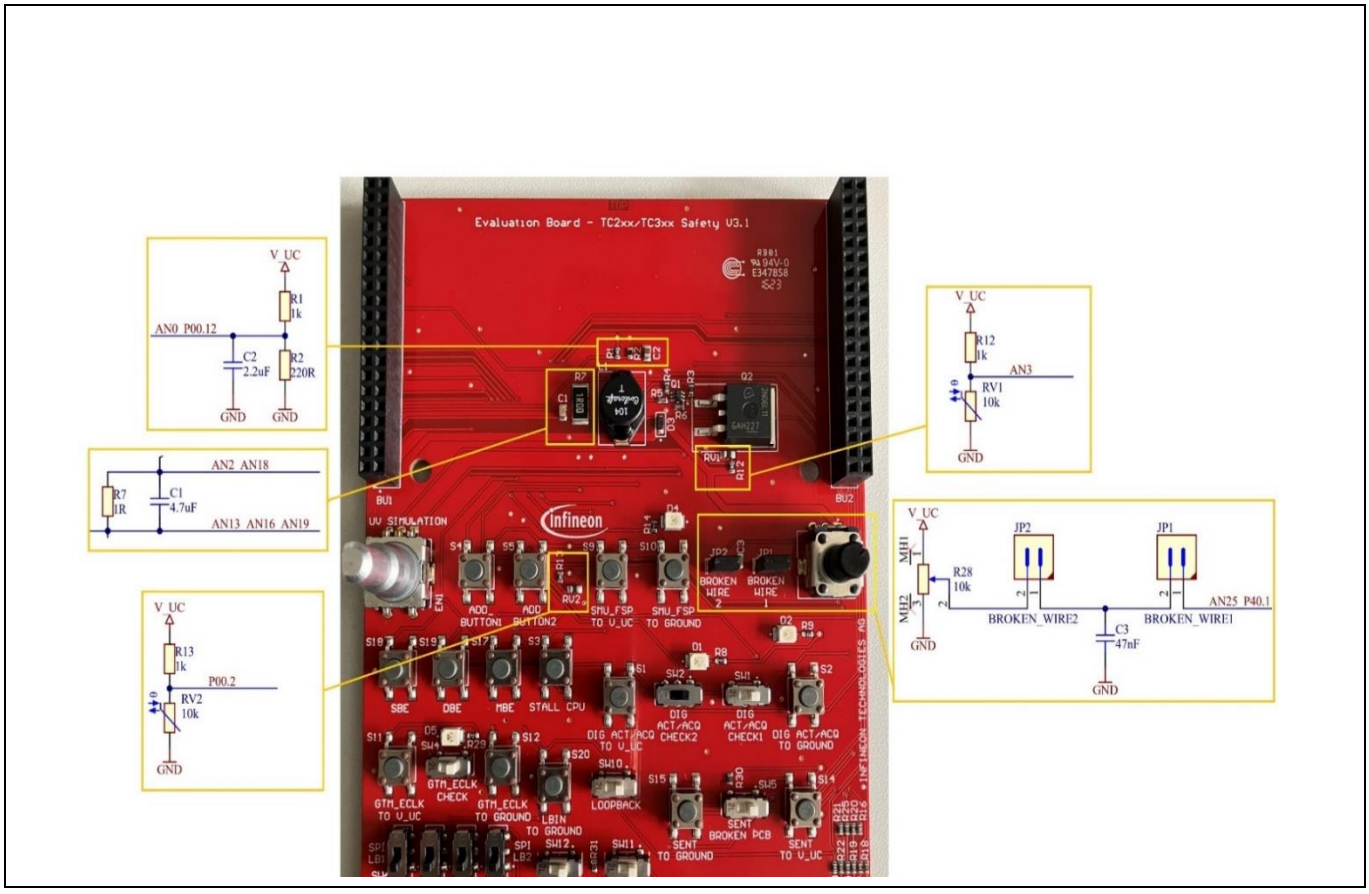


Figure 52 Visual overview of analog input signals available on the Safety Evaluation Board

Attention: ***Buttons and switches should be operated sequentially on the Safety Board because short-circuits will appear when enabling simultaneously a strong pull-up and a strong pull-down circuitry on the same line. This could permanently damage the demo setup. In case of such an event, the board can also enter a safe mode due to the overtemperature condition. In that case, the board is temporarily unusable (but not damaged) for approximately one minute.***

Table 9 Overview of analog input signals and their EVADC/EDSADC functionality

ID	Signal	Input pin / symbol	EVADC / EDSADC functionality
1	RV1 10kΩ NTC EVADC input	AN3	EVADC analog input channel 3, group 0
2	RV2 10kΩ NTC EVADC input	P00.2	EVADC analog input channel 10, group 9
3	EVADC reference voltage (VAREF)	AN0	EVADC analog input 0, channel 0, group 0
4	RV1 10kΩ NTC EDSADC input	AN3	EDSADC negative analog input channel 0, pin A
5	RV2 10kΩ NTC EDSADC input	P00.2	EDSADC positive analog input channel 5, pin A
6	EDSADC reference voltage (VAREF)	AN0	EDSADC positive analog input channel 3, pin A
7	Undervoltage simulation EVADC input 1	P40.12 (AN19)	EVADC analog input channel 3, group 2
8	Undervoltage simulation EVADC input 2	P40.12 (AN19)	EVADC analog input channel 9, group 11
9	Broken-wire potentiometer EVADC input	P40.1	EVADC analog input channel 1, group 3
10	Broken-wire potentiometer EDSADC input	P40.1	EDSADC negative analog input channel 2, pin B

Overview of the user interface

See Figure 6 to note the buttons for the analog acquisition example selection and the buttons to start/stop the potentiometer ADC measurement.

Note: All FUCs except the analog acquisition FUC3 with additional broken-wire detection can be selected via the four buttons on the right. In the current version of Application Kit Safety, the ADC results of these FUCs can only be observed via the debugger (g_analogAcquisitionStatus). As the hardware jumper J1 and J2 are used for the broken-wire simulation, you should connect the jumper when starting and stopping the analog acquisition FUC3 with the two TFT buttons "Run safe ADC capturing" and "Stop safe ADC capturing". The mark on the potentiometer should point to the bottom of the Application Kit Safety Extension Board to avoid an alarm while starting the measurement.

Overview of the software organization

All five analog acquisition functional use cases including all required SMs (depends on the module used in each FUC) are fully implemented as part of this application note. As shown in Figure 53, a global source and header file and additional files for every individual use case are provided. The global file includes functions, variables, and data types used by every FUC; these functions, variables, and data types are called and configured individually depending on the specific configuration requirements of each FUC. See the following sections and the corresponding source and header files for more information.

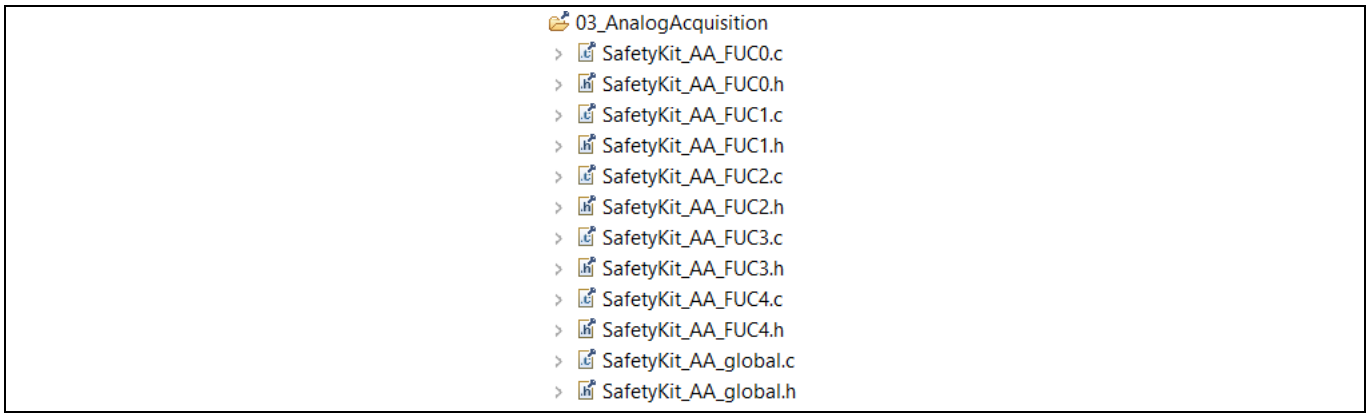


Figure 53 Folder structure of the software for analog acquisition functional use case

Code Listing 34 Example code for selection and initialization of analog acquisition FUCs

```
void initSafetyKit(void)
{
[..]
    /* Initialize */
    /* Configure mode variables for FUCs, initialization is done in background endless loop
*/
   g_SafetyKitStatus.analogAcquisitionMode = initAAcqFuc0Mode;
[..]
}
void runSafetyKitEndlessLoopCpu0(void)
{
[..]
   /* Initialize / Reinitialize Analog Acquisition functional use case examples */
        switch(g_SafetyKitStatus.analogAcquisitionMode)
        {
            case initAAcqFuc0Mode:
                initAAcqFuc0();
                g_SafetyKitStatus.analogAcquisitionMode = runAAcqFuc0Mode;
                break;
            case initAAcqFuc1Mode:
                initAAcqFuc1();
                g_SafetyKitStatus.analogAcquisitionMode = runAAcqFuc1Mode;
                break;
            case initAAcqFuc2Mode:
                initAAcqFuc2();
                g_SafetyKitStatus.analogAcquisitionMode = runAAcqFuc2Mode;
                break;
            case initAAcqFuc3Mode:
                initAAcqFuc3();
                g_SafetyKitStatus.analogAcquisitionMode = runAAcqFuc3Mode;
                break;
            case initAAcqFuc4BrokenWRMode:
                initAAcqFuc4BrokenWR();
                g_SafetyKitStatus.analogAcquisitionMode = runAAcqFuc4BrokenWRMode;
                break;
            default:
                break;
        } [..]
}
```

\AppSw\SafetyKit\SafetyKit_Main.c

Code Listing 35 Example code for the execution of analog acquisition FUCs

```
/*
 * This ISR function is called every 1ms by each CPU for task scheduling
 * */
void runSafetyKitStmIsr(App_Cpu *lcl_AppCpu, IfxCpu_ResourceCpu cpuIndex){
[..]
/* Run Analog Acquisition functional use case examples */
            switch(g_SafetyKitStatus.analogAcquisitionMode)
            {
                case runAAcqFuc0Mode:
                    runAAcqFuc0();
                    break;
                case runAAcqFuc1Mode:
                    runAAcqFuc1();
                    break;
                case runAAcqFuc2Mode:
                    runAAcqFuc2();
                    break;
                case runAAcqFuc3Mode:
                    runAAcqFuc3();
                    break;
                case runAAcqFuc4BrokenWRMode:
                    runAAcqFuc4BrokenWR();
                    break;
                default:
                    break;
            }
[..]
}
```

\AppSw\SafetyKit\SafetyKit_Main.c

6.2.10.2 Analog acquisition implementation

The following FUCs are considered for analog acquisition described in detail in AURIX™ TC3xx Safety Manual V2.0 [3].

Analog acquisition with redundant EVADC channels (FUC0)

For FUC0 of analog acquisition, two redundant NTC sensors are sampled with different channels of two EVADC modules (see signal 1 and 2 in Table 9). As specified in the AURIX™ TC3xx Safety Manual [3], an ADC reference voltage (VAREF) measurement is also required for this use case. A divider bridge is used for that purpose using V_UC (see signal 3 in Table 9 and Figure 54 for the circuit).

After a successful measurement of both signals with two different EVADC channels, the signal is further transported to a system volatile memory and compared by the CPU.

Note: It would also be possible to measure the secondary monitor internal bandgap signal using the V_{MTS} signal (channel GxCH29). See Section 32.12.5 "On-Chip Supervision Signals" in the User Manual [1]). Because V_{MTS} is a precision bandgap source, comparing it to the known expected ratio with VAREF can be used to verify VAREF. Measuring GxCH29 is not yet part of the Application Kit Safety application software.

This example is also referred as "EVADC +EVADC " on TFT display menu. The port pins used for this FUC0 are as follows:

- AN3 (G0CH3, IN) as S1 mission “connected to RV1 NTC sensor”
- P002 (G0CH10, IN) as S2 monitor “connected to RV2 NTC sensor”
- AN0 (G0CH0, IN) V reference

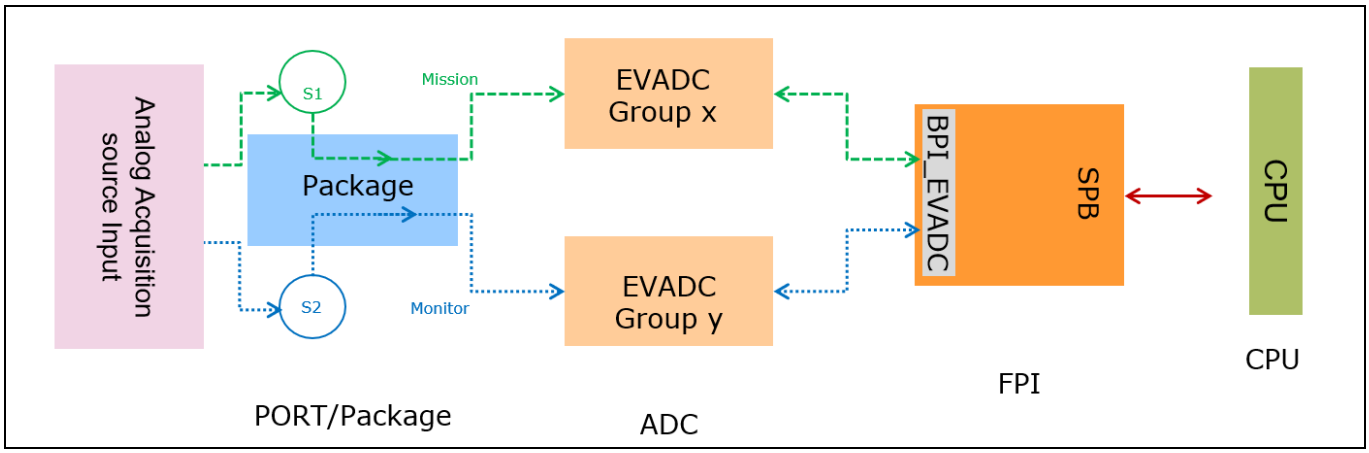


Figure 54 Analog acquisition with redundant EVADC channels (FUC0) overview

Analog acquisition with redundant EDSADC channels (FUC1)

The second use case nearly identical to FUC0. The only difference is that three EDSADC channels (see signals 4, 5, and 6 in Table 9 and Figure 55) are used instead of the three EVADC channels used in the previous use case. This is possible because many analog pins support the use both as EVADC and as EDSADC input signal.

This example is also referred as “EDSADC +EDSADC “ on the TFT display menu. The port pins used for this FUC1 are as follows:

- AN3 (DS0NA, IN) as S1 mission “connected to RV1 NTC sensor”
- P002 (DS5PA, IN) as S2 monitor “connected to RV2 NTC sensor”
- AN0 (DS3PA, IN) V reference

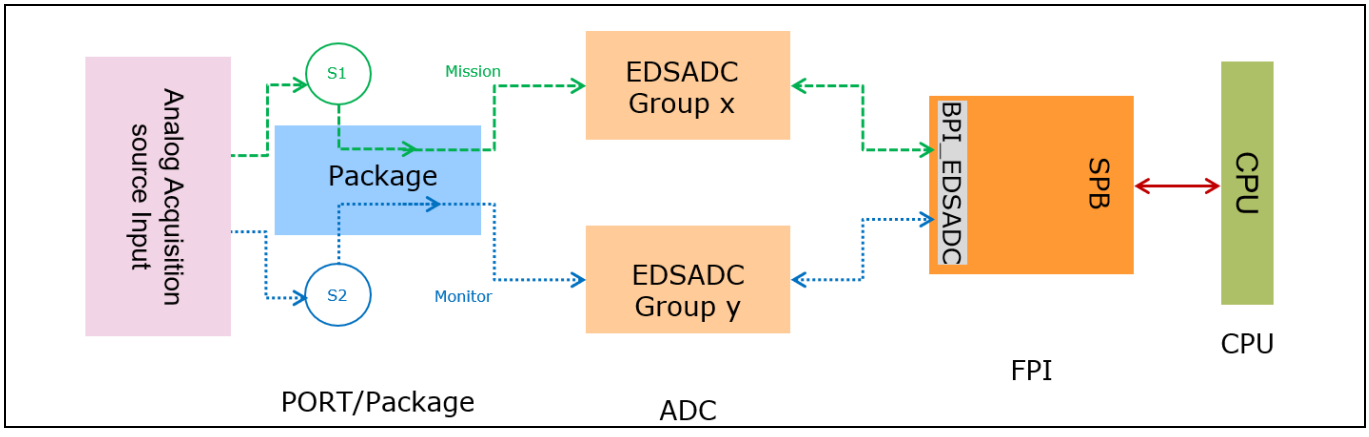


Figure 55 Analog acquisition with redundant EDSADC channels (FUC1) overview

Analog acquisition with one EVADC channel and one EDSADC channel (FUC2)

The analog acquisition usage with one EVADC and one EDSADC channel combines the first two functional use cases by using an EVADC channel for the acquisition of the RV1 sensor and an EDSADC to measure the value of the RV2 sensor (see Figure 56). Note that this could also be implemented vice versa.

This example is also referred as "EVADC +EDSADC " on TFT display menu. The port pins used for this FUC1 are as follows:

- AN3 (G0CH3, IN) as S1 mission “connected to RV1 NTC sensor”
- P002 (DS5PA, IN) as S2 monitor “connected to RV2 NTC sensor”
- AN0 (DS3PA, IN) V reference
- AN0 (G0CH0, IN) V reference

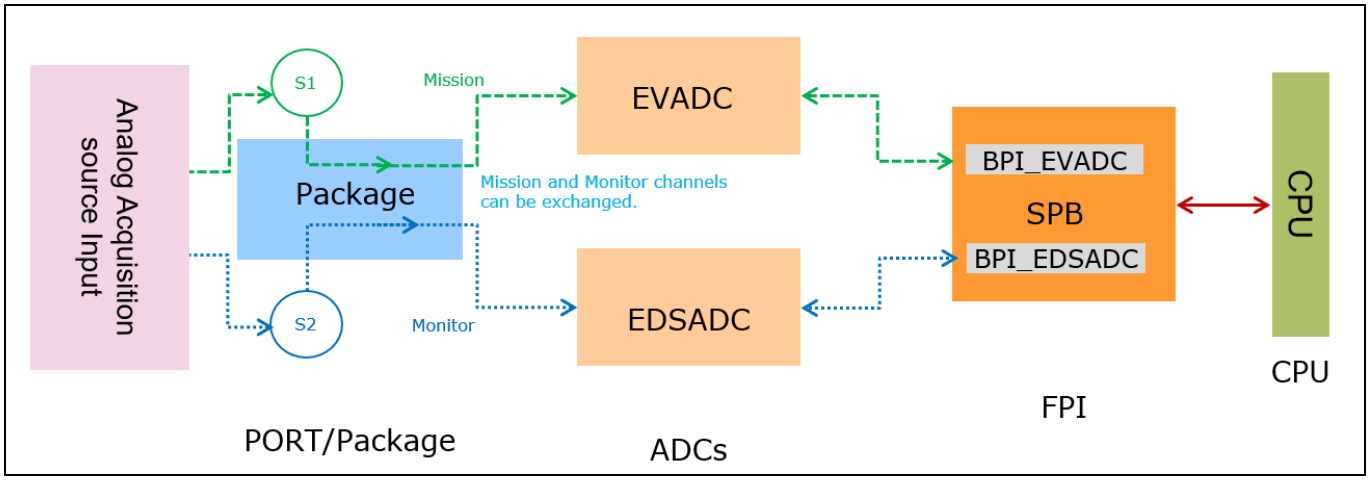


Figure 56 Analog acquisition with one EVADC channel and one EDSADC channel (FUC2)

Single analog acquisition with EVADC channels (FUC3)

In this example, a safety-related analog signal is received on a single port pin of the microcontroller (see Figure 57). This signal is later redundantly processed by two different internal resources of the EVADC (see signals 7 and 8 in Table 9). Later, the CPU compares both signals to verify fault-free acquisition.

Note: The value of the analog input signal measured in this use case depends on a PWM signal generated for the undervoltage simulation mentioned in Section 6.2.8.1. Therefore, to synchronize the sampling of both EVADC channels, the measurement is triggered by a TOM channel.

This example is also referred as "2 EVADC 1 PIN " on TFT display menu. The port pins used for this FUC0 are as follows:

- P40.12 (**G2CH3**, IN) as S1 mission
- P40.12/AN19 (**G11CH9**, IN) as S2 monitor

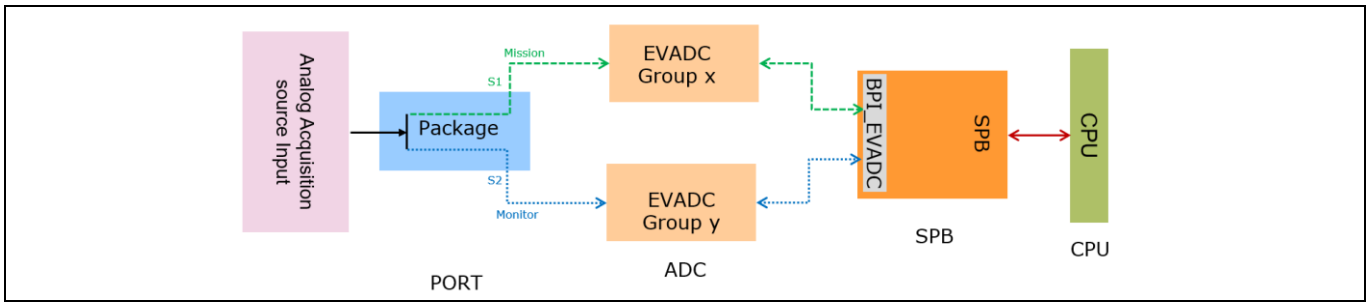


Figure 57 Single analog acquisition with EVADC channels overview

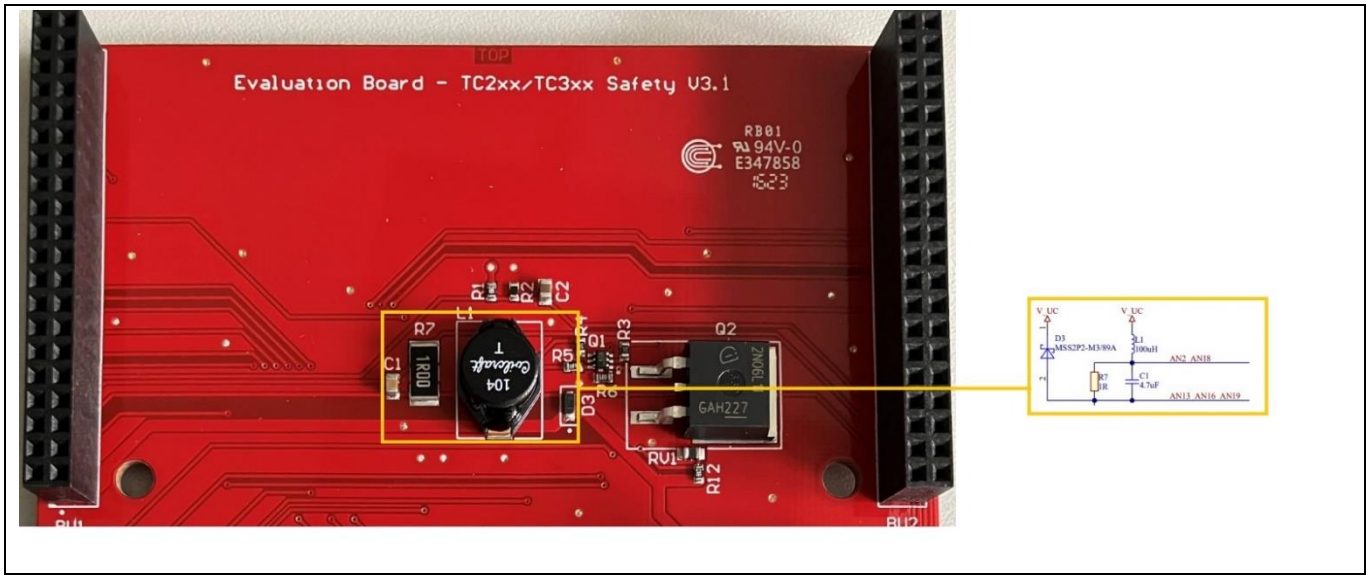


Figure 58 Single analog acquisition with EVADC channels circuitry

Single analog acquisition with one EDSADC and one EVADC channel (FUC4) and additional broken-wire detection

Another way to measure a single safety-related analog input signal via a single port pin of the microcontroller is to use both an EVADC and an EDSADC channel (see Figure 59). The signal measured by both modules is also subsequently compared by the CPU.

The port pins used for this FUC0 are as follows:

- P40.1 (**G3CH1**, IN) as S1 mission
- P40.1 (**DS2NB**, IN) as S2 monitor

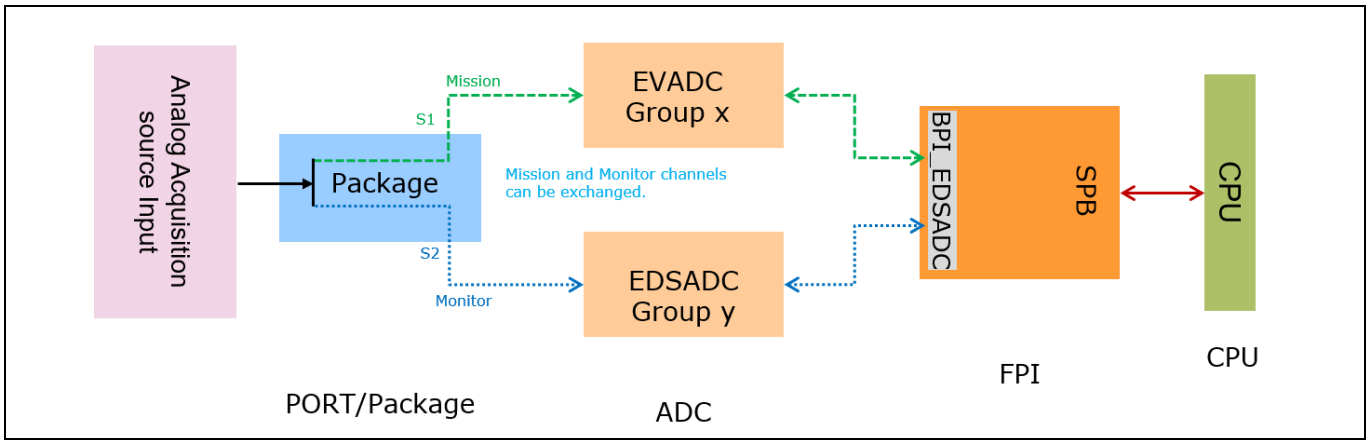


Figure 59 Single analog acquisition with one EDSADC and one EVADC channel overview

This example also includes a demonstration of the broken-wire detection hardware safety features included in the EVADC and hardware configuration of the limit check for *Safety Mechanism EVADC:PLAUSIBILITY*. This demo utilizes the black potentiometer R28 to generate the captured analog input signal. Jumpers J1 and J2 emulate a broken circuit (see Figure 60). J2 is used to break the connection to sensor (Poti), while J1 is to break the connection of the 47 nF capacitor. If J1 is connected and when J2 is disconnected, it means that the sensor is disconnected from the MCU but because the 47 nF capacitor is still charged and takes time to discharge, there will be a delay in broken-wire detection. To overcome this delay, J2 is used to disconnect the 47 nF capacitor. J1 and J2 should be connected by default and should be disconnected to see the broken-wire SMU alarm. The VAREF signal is generated with a divider bridge out of V_UC. For the list of utilized pins, see signals 3 and 6 in Table 9.

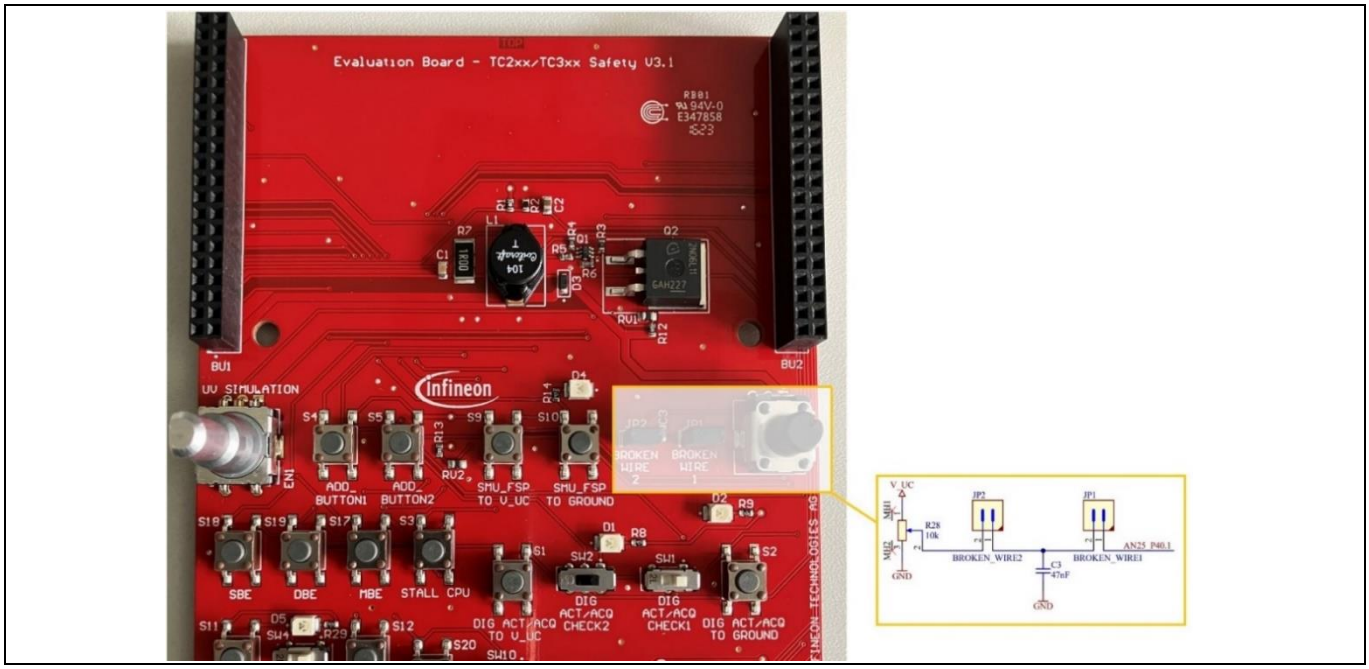


Figure 60 Single analog acquisition with one EDSADC and one EVADC channel (FUC1) and additional broken-wire detection overview

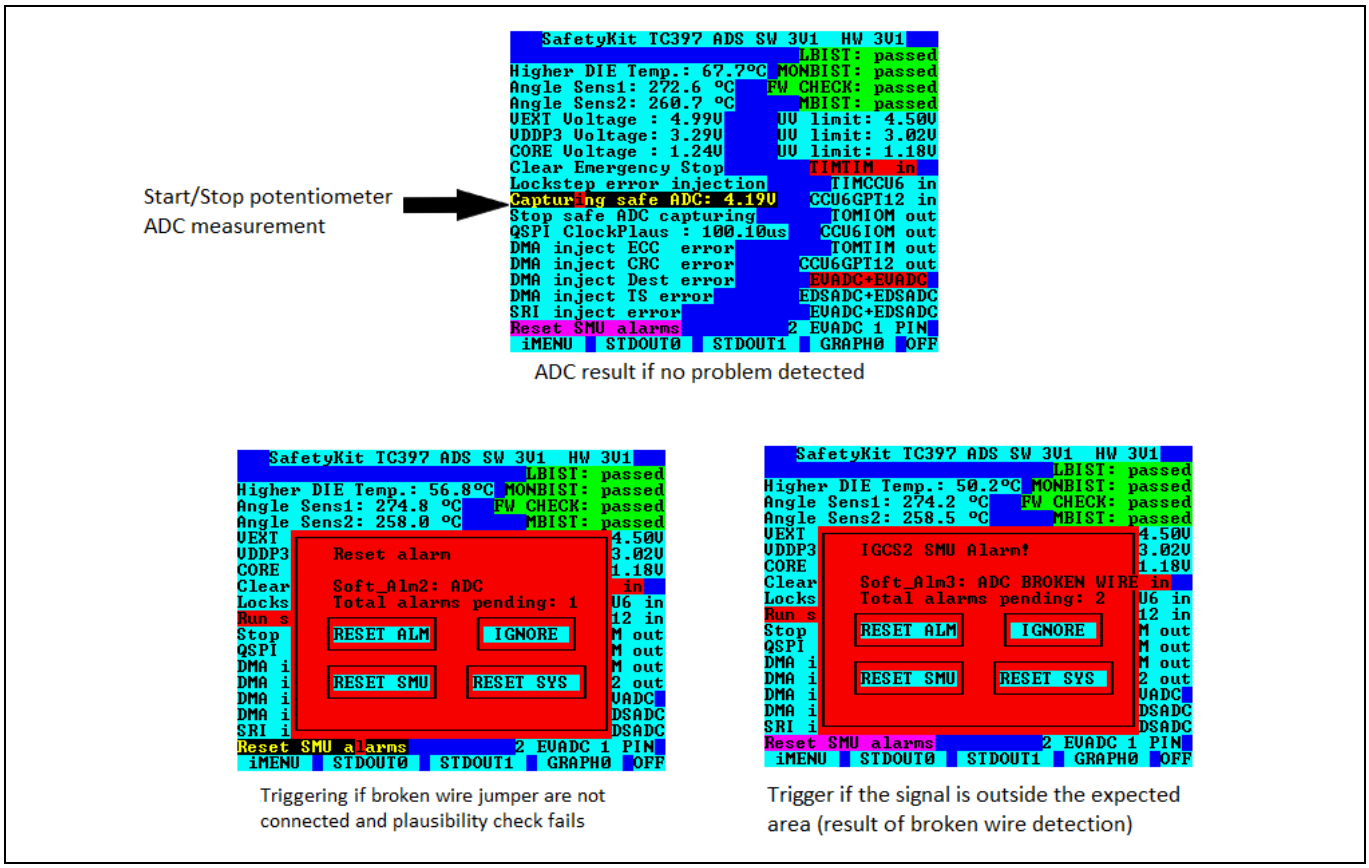


Figure 61 Analog acquisition FUC4 with broken-wire detection display overview

Note: The alarm, which was triggered last is shown on the TFT as it overwrites the previously triggered alarm. Therefore, the "ADC" alarm is shown, which needs to get reset to observe the older "ADC BROKEN WIRE" alarm, which was triggered earlier.

With broken-wire detection as an additional safety feature on the EVADC channel, the plausibility check is divided into two steps:

- Hardware boundary check on the EVADC module
- CPU comparison of the EVADC and the EDSADC conversion results

Note: See the C function `initEVADCBrokenWireDetection` *for more details on the implementation of the hardware boundary check. For more information of the ADC channel result comparison with the CPU, see the C function* `plausibilityCheck`*.*

6.2.11 MCU function – Timers

The following section covers various application examples for the acquisition and actuation of safety-related digital signals by the usage of following timer modules available in the AURIX™ TC3xx family i.e., generic timer module (GTM), capture/compare unit 6 (CCU6), and the general purpose timer unit (GPT12). See the AURIX™ TC3xx User Manual [1] for more information on the nominal functionality of these modules.

As described in the Safety Manual V2.0 [3], there are two different groups: the safety-related functions with various functional use cases (FUCs) for the usage of timer module.

As described in the Safety Manual [3], the safety-related function “digital acquisition" is composed of five different use cases for the use of these timer modules for acquisition of safety-related digital signals in different applications. In addition, four other use cases for performing actuation of safety-related digital signals are presented in the safety-related function “digital actuation”.

The following two lists and Table 10 present an overview of all functional use cases mentioned in the Safety Manual [3], and provides a list of functional blocks and SMs required for their implementation.

Digital acquisition

- FUC**0**: Digital acquisition with redundant TIM/TIM channels
- FUC**1**: Digital acquisition with redundant CCU6/TIM channels
- FUC**2**: Digital acquisition with redundant CCU6/GPT12 channels

Digital actuation

- FUC**0**: Digital actuation with redundant TOM channels and IOM comparison
- FUC**1**: Digital actuation with redundant TOM/CCU6 channels and IOM comparison
- FUC**2**: Digital actuation with redundant TOM/TIM channels and application SW comparison
- FUC**3**: Digital actuation with redundant CCU6/GPT12 channels and application SW comparison

Table 10 Overview of SMs required for digital acquisition and digital actuation functional use cases

FUC	Functional block involved	SMs required
Digital acquisition FUC**0**	• 2x GTM TIM	• *Safety Mechanism TIM_REDUNDANCY* • *Safety Mechanism TIM_CLOCK_MONITORING*
Digital acquisition FUC**1**	• GTM TIM • CCU6	• *Safety Mechanism GTM_CCU6_REDUNDANCY* • *Safety Mechanism TIM_CLOCK_MONITORING*
Digital acquisition FUC**2**	• CCU6 • GPT12	• *Safety Mechanism CCU6_CAPTURE_MON_BY_GPT12*
Digital actuation FUC**0**	• GTM TOM • IOM	• *Safety Mechanism IOM_ALARM_CHECK* • *Safety Mechanism TIM_CLOCK_MONITORING*
Digital actuation FUC**1**	• GTM • CCU6 • IOM	• *Safety Mechanism IOM_ALARM_CHECK*
Digital actuation FUC**2**	• GTM TOM • GTM TIM	• *Safety Mechanism TOM_TIM_MONITORING* • *Safety Mechanism TIM_CLOCK_MONITORING*
Digital actuationFUC**3**	• CCU6 • GPT12	• Safety Mechanism CCU6_GPT12_MONITORING

The system integrator should also take care of the common-cause failures (CCF) when selecting the redundant port pins (see Section 6.2.9.1 for more information). See Section 6.2.11.2 for examples of the digital acquisition; for digital actuation examples, see Section 6.2.11.3.

6.2.11.1 Overview of digital acquisition and digital actuation implementation

The following figure shows the components and circuitry of Application Kit Safety, which are involved in the implementation of digital acquisition and digital actuation.

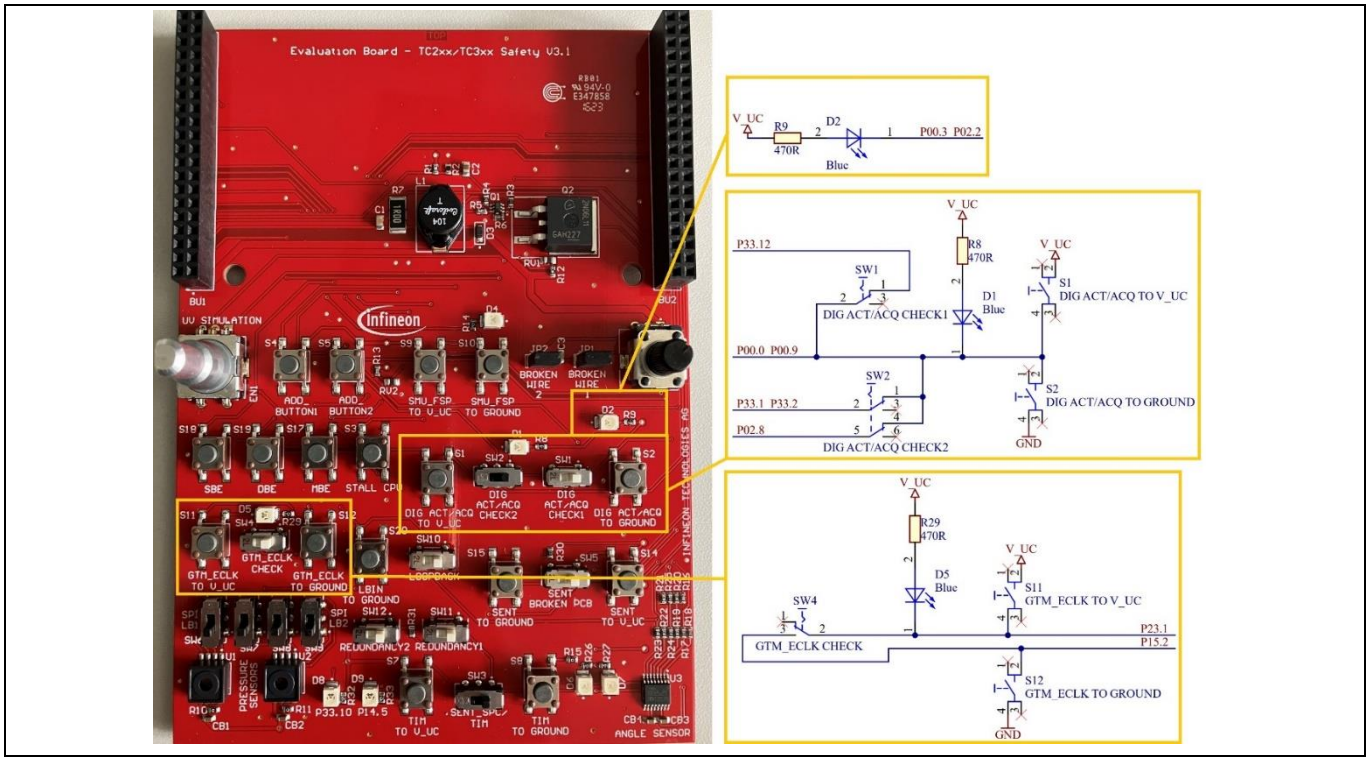


Figure 62 Overview of digital signals available on Safety Demo Board

Attention: ***Buttons and switches should be operated sequentially on the Safety Board as short-circuits will appear when enabling simultaneously a strong pull-up and a strong pull-down circuitry on the same line. This could permanently damage the demo setup. In case of such an event, the board can also enter a safe mode due to overtemperature condition. In that case, the board is temporarily unusable (but not damaged) for approximately one minute.***

Table 11 Overview of digital signals and their GTM/IOM/CCU6/GPT12 functionality

ID	Signal	Input pin/ symbol	GTM/IOM/CCU6 functionality
1	TIM mission signal	P02.8	GTM TIM3 channel 0
2	TIM monitor signal	P33.12	GTM TIM2 channel 0
3	TOM safety-related digital output signal. Either used as dummy signal to simulate a digital safe input signal for digital acquisition or as TOM mission signal in digital actuation.	P00.0	GTM TOM0 channel 4, TOUT9
4	TOM reference signal, also called “monitor signal”	P02.2	GTM TOM1 channel 10, TOUT2
5	IOM GTM monitor input pin	P33.1	IOM monitor input
6	IOM GTM reference input signal	GTM TOUT2	IOM reference input

ID	Signal	Input pin/ symbol	GTM/IOM/CCU6 functionality
7	ATOM safety-related digital output signal	P00.0	GTM ATOM0 channel0, TOUT9
8	ATOM channel to trigger start of CCU6 timer	Pin not used, only channel	GTM ATOM0 channel 1
9	CCU6 reference signal output, also called “monitor signal”	P00.3	CCU6 module 1, timer 12, channel 1 output
10	IOM CCU6 reference input signal	CCU61 Cc61	IOM reference input
11	External clock 0 (ECLK0) output signal (GTM external clock to port mapping)	P23.1	GTM_CLK0 signal (alternate output 4)
12	GTM TIM ECLK input signal	P15.2	GTM TIM2 channel 5
13	CCU6 Mission Signal	P00.9	CCU61 CC62INC input
14	CCU6 Monitor Signal	P00.9	CCU61 CC62INC input
15	CCU6 Output Mission	P00.0	CCC60 COUT63
16	GPT12 Monitor Signal	P02.8	GPT12 T4INA

Overview of the user interface

Figure 6 shows the buttons for digital acquisition and digital actuation example selection. By pressing one of the seven buttons, the specific FUC is initialized. Further information on the execution of the FUC can be observed via the debugger (*g_digitalAquisitionStatus*). Figure 65, Figure 74, Figure 77, and Figure 80 show the buttons and switches that can be used for fault injection during the execution of each FUC.

Overview of the software organization

The current implementation of Application Kit Safety includes FUC0, FUC1, and FUC2 of digital acquisition and additionally FUC0, FUC1, FUC2, and FUC3 of the safety-related function digital actuation. SMs required for each FUC are fully implemented according to the recommendation of the Safety Manual [3]. As shown in Figure 63, a global source and header file, additional files for every individual use case, and a configuration readback file are provided. The global file includes the functions, variables, and data types used by every FUC. These functions, variables, and data types are called and configured individually depending on the specific configuration requirements of each FUC. See the following sections and the corresponding source and header files for more information.

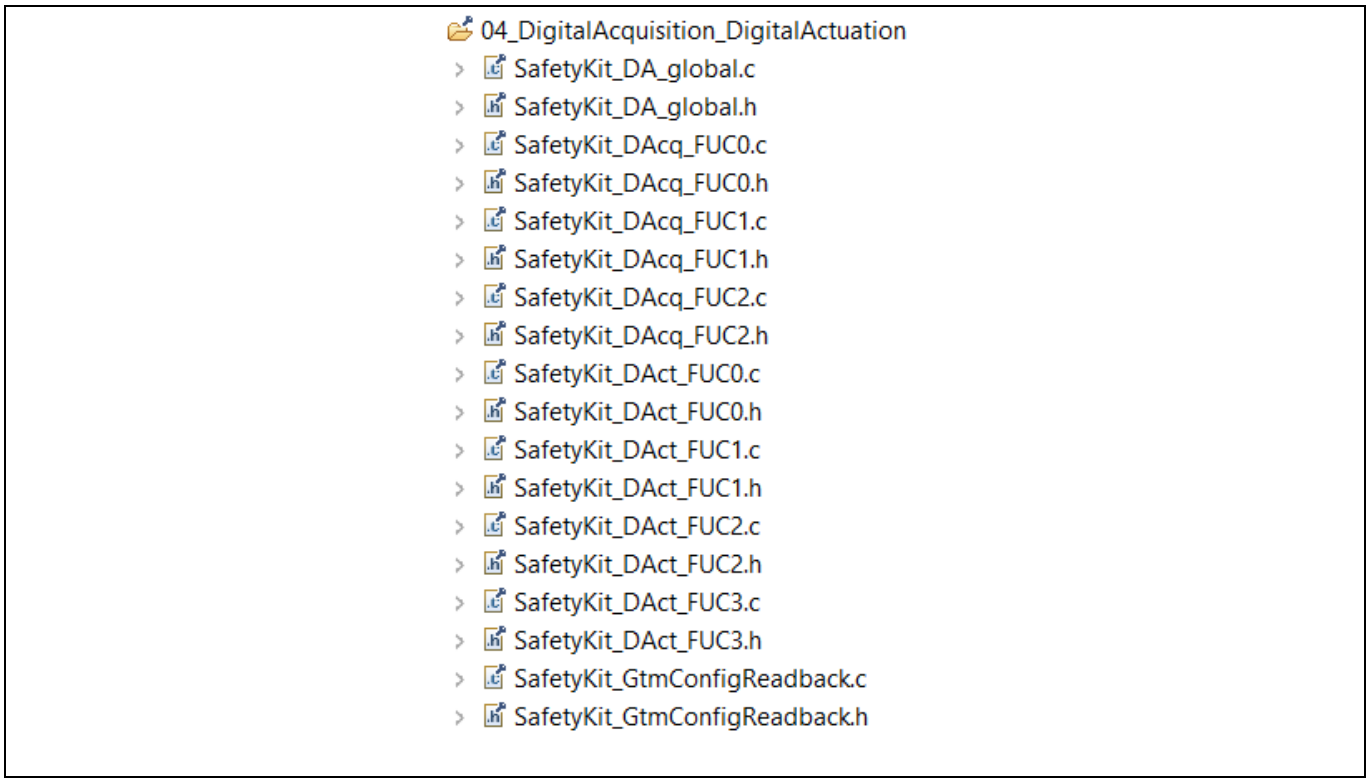


Figure 63 Folder structure of software for digital acquisition and digital actuation

Code Listing 36 Example code for selection and initialization of digital acquisition and digital actuation FUCs

```
void initSafetyKit(void)
{
[..]
    /* Initialize */
    /* Configure mode variables for FUCs, initialization is done in background endless loop
*/
    g_SafetyKitStatus.digitalAcqActMode     = initialize_DAcq_FUC0;
[..]
}
/*
 * This function is called endless loop of CPU1 main and perform digital actuation and ac-
quisition
 * */
void runSafetyKitEndlessLoopCpu1(void)
{
    if (lastDigitalAcqActMode != g_SafetyKitStatus.digitalAcqActMode)
        {
        /* Initialize / Reinitialize Digital Acquisition or Actuation functional use case
examples */
            switch(g_SafetyKitStatus.digitalAcqActMode)
            {
                case initialize_DAcq_FUC0:
                    initDAcqFuc0();
                    g_SafetyKitStatus.digitalAcqActMode = run_DAcq_FUC0;
                    break;
                case initialize_DAcq_FUC1:
                    initDAcqFuc1();
                    g_SafetyKitStatus.digitalAcqActMode = run_DAcq_FUC1;
                    break;
                case initialize_DAcq_FUC2:
                    initDAcqFuc2();
                    g_SafetyKitStatus.digitalAcqActMode = run_DAcq_FUC2;
                    break;
                case initialize_DAct_FUC0:
                    initDActFuc0();
                    g_SafetyKitStatus.digitalAcqActMode = run_DAct_FUC0;
                    break;
                case initialize_DAct_FUC1:
                    initDActFuc1();
                    g_SafetyKitStatus.digitalAcqActMode = run_DAct_FUC1;
                    break;
                case initialize_DAct_FUC2:
                    initDActFuc2();
                    g_SafetyKitStatus.digitalAcqActMode = run_DAct_FUC2;
                    break;
                case initialize_DAct_FUC3:
                    initDActFuc3();
                    g_SafetyKitStatus.digitalAcqActMode = run_DAct_FUC3;
                    break;
                default:
                    break;
            }
            lastDigitalAcqActMode = g_SafetyKitStatus.digitalAcqActMode;
        }
}
```

\AppSw\SafetyKit\SafetyKit_Main.c

6.2.11.2 Digital acquisition implementation

This section describes digital acquisition with different FUCs and their implementation.

Digital acquisition with redundant TIM/TIM channels (FUC0)

In FUC0 of the safety-related function “digital acquisition”, two redundant digital signals are acquired by redundant GTM timer input resources (TIM). The signal measurement results (e.g., period and duty cycle of the signal) must be read and compared by the CPU. The two SMs *TIM_REDUNDANCY* and *TIM_CLOCK_MONITORING* are required for the implementation of this FUC.

An illustrative example of the *Safety Mechanism TIM_REDUNDANCY* is shown in Figure 64; an overview of the Application Kit Safety implementation including the *Safety Mechanism TIM_CLOCK_MONITORING* and the direction of switches for fault injection are shown in Figure 65. This example is also referred as the "TIMTIM in” example. The port pins used for this SM are as follows:

- P02.8 (TIM3_CH0, IN) as S1 mission “TIMx_CHy”
- P33.12 (TIM2_CH0, IN) as S2 monitor “TIMa_CHb”
- P00.0 (TOM0_CH4, TOUT9, OUT) generate dummy PWM which feedback to S1 and S2

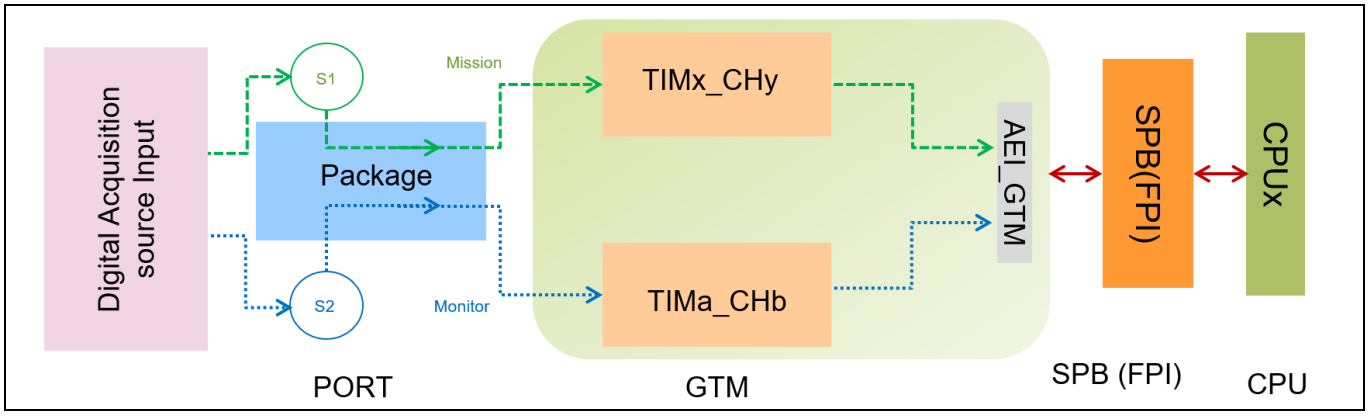


Figure 64 Illustrative example of safety mechanism TIM_REDUNDANCY

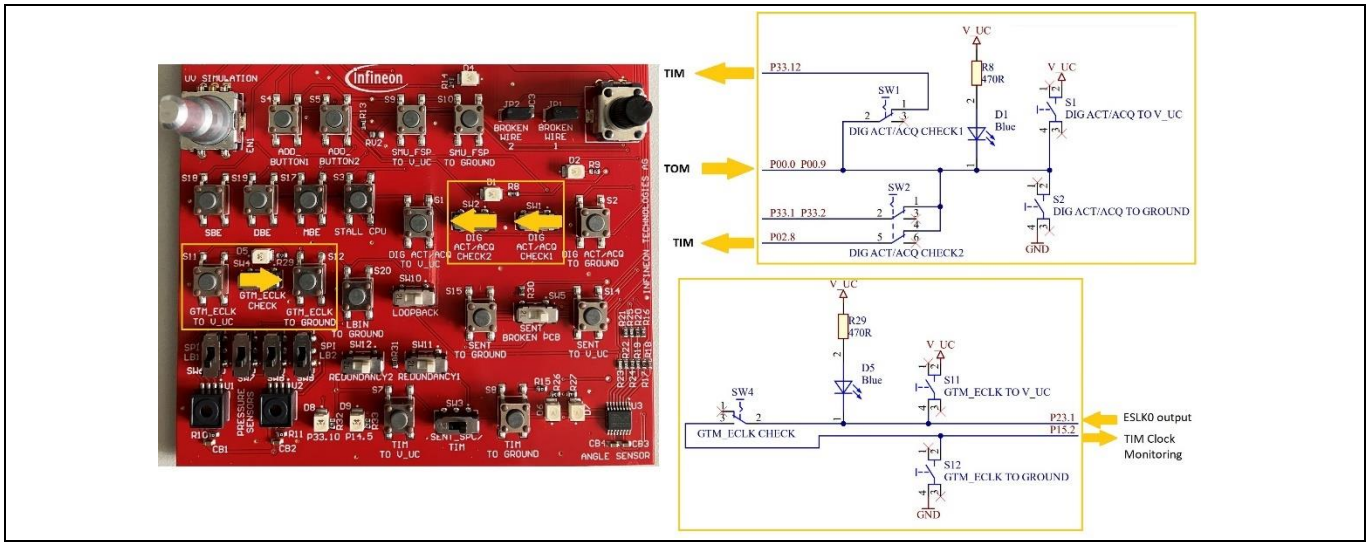


Figure 65 Digital acquisition with redundant TIM/TIM channels Application Kit Safety implementation overview including highlighted buttons and switches for fault injection

Note: See Figure 3 for more information on default states of the switches.

Figure 66 shows how to trigger the configuration of all modules involved in this FUC and shows the SMU alarm response if fault injection is triggered via the buttons or switches highlighted in Figure 65.

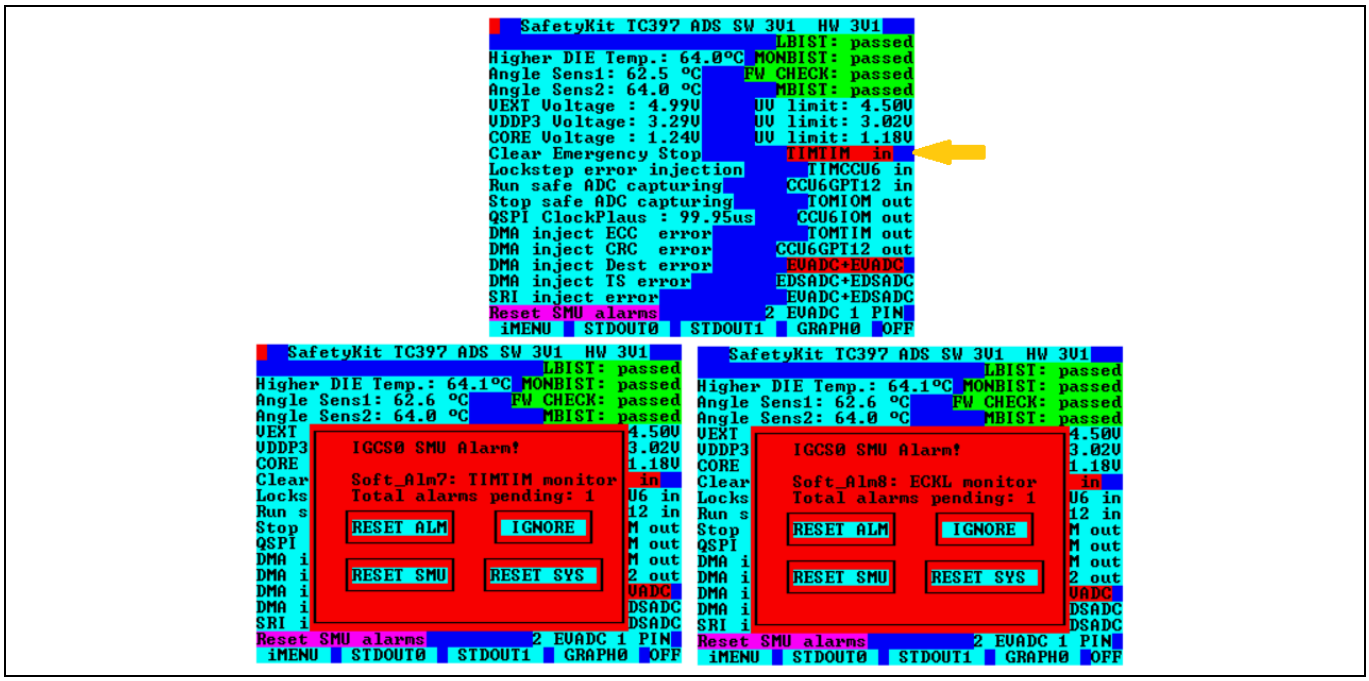


Figure 66 Digital acquisition with redundant TIM/TIM channels example selection at the top and alarm reaction to fault injection via switches or buttons

Digital acquisition with redundant CCU6/TIM channels (FUC1)

In FUC1 of the safety-related function “digital acquisition”, two redundant digital signals are acquired by redundant GTM timer input resources (TIM) and CCU6 respectively. The signal measurement results (e.g., period and duty cycle of the signal) must be read and compared by the CPU. The two SMs, *GTM_CCU6_REDUNDANCY* and *TIM_CLOCK_MONITORING* are required for the implementation of this FUC.

An illustrative example of the *Safety Mechanism GTM_CCU6_REDUNDANCY* and an overview of the Application Kit Safety implementation of the FUC including the *TIM_CLOCK_MONITORING* safety mechanism are shown in Figure 67 and Figure 68. This example is also referred as the “TIMCCU6 in” example. The port pins used for this SM are as follows:

- P33.12 (TIM2_CH0, IN) as S2 monitor “TIMx_Chy”
- P00.9 (CCU61_CC62INC, IN) as S2 monitor “CCU6”
- P00.0 (TOM0_CH4, TOUT9, OUT) generate dummy PWM which feedback to S1 and S2

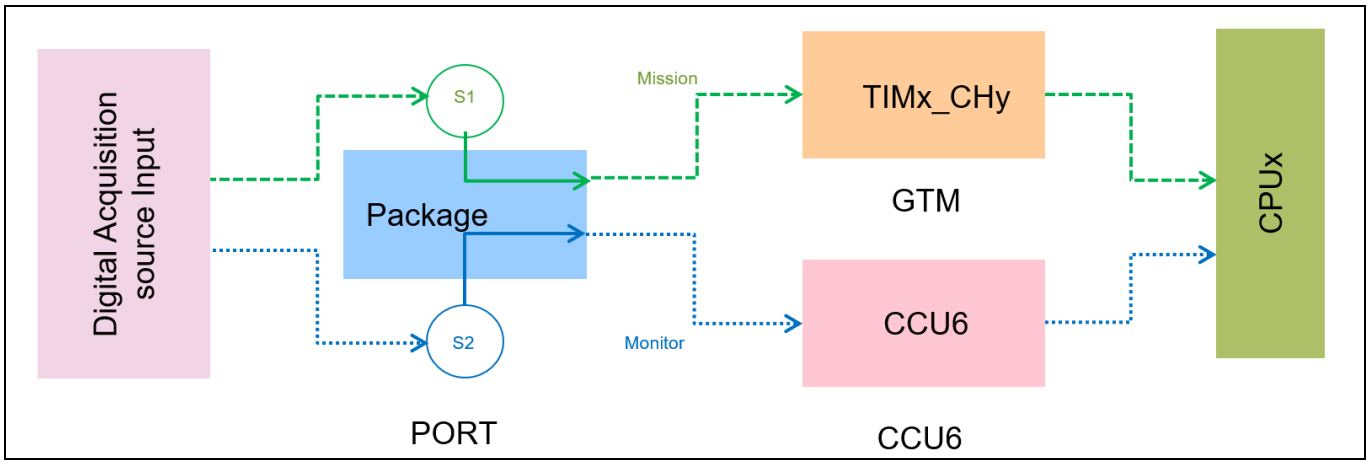


Figure 67 Safety Mechanism GTM_CCU6_REDUNDANCY

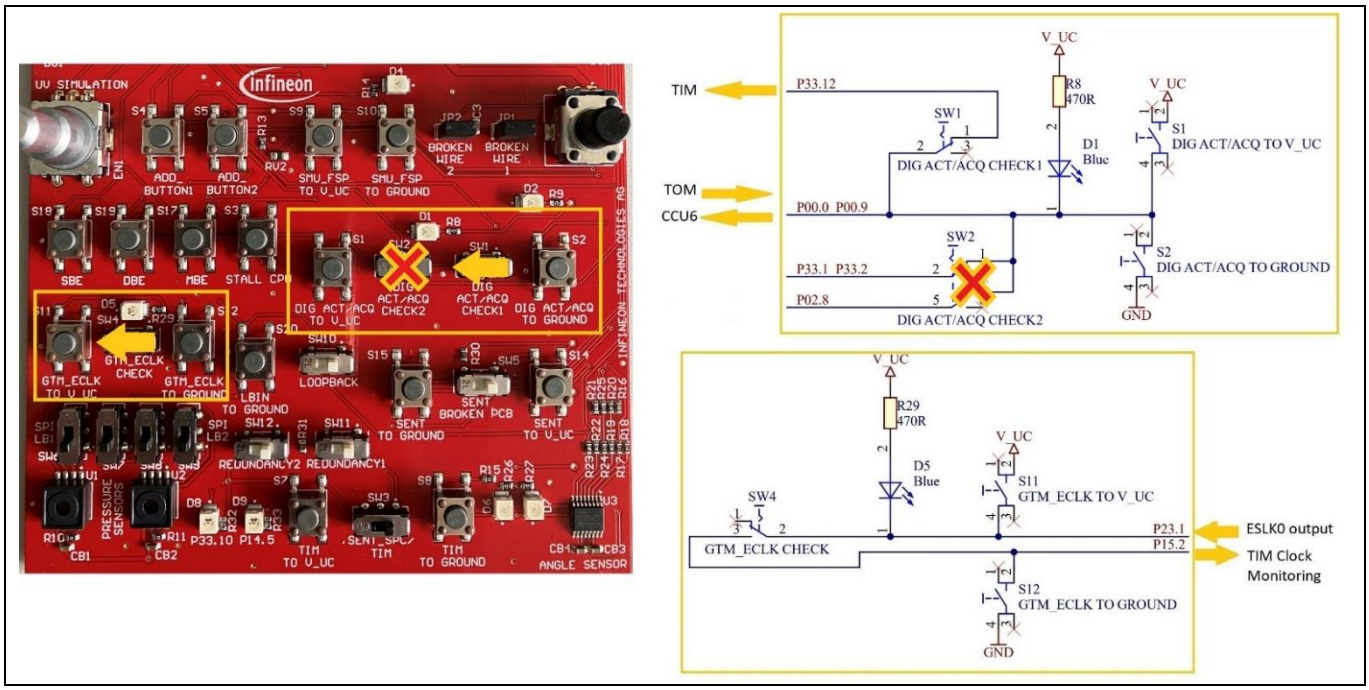


Figure 68 Digital acquisition with redundant TIM/CCu6 channels including highlighted buttons and switches for fault injection (X == not used)

Figure 69 shows how to trigger the configuration of all modules involved in this FUC and shows the SMU alarm reaction if fault injection is triggered via the buttons or switches highlighted in Figure 68.

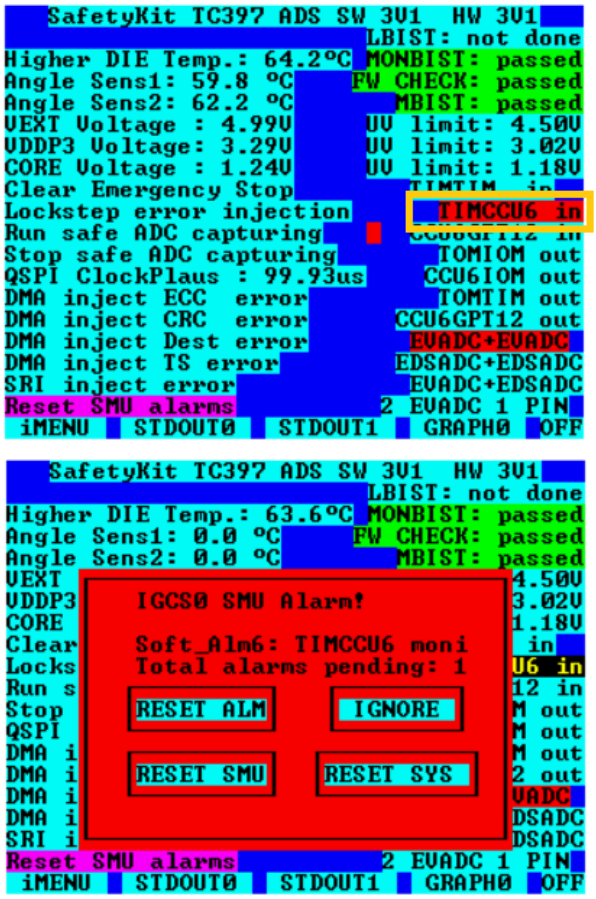


Figure 69 TIM/CCU6 alarm pop-up after error injection

Digital acquisition with redundant CCU6/GTP12 channels (FUC2)

In FUC2 of the safety-related function “digital acquisition”, the MCU receives redundant signals, and each signal is acquired by CCU6 and GPT12 respectively. The signal measurement results (e.g., period and duty cycle of the signal) must be read and compared by the CPU. The SMs *CCU6_CAPTURE_MON_BY_GPT12* are required for the implementation of this FUC.

An illustrative example of the *Safety Mechanism CCU6_CAPTURE_MON_BY_GPT12* and an overview of the Application Kit Safety implementation are shown in Figure 70 and Figure 71 . This example is also referred as the “CCU6GPT12 in” example. The port pins used for this SM are as follows:

- P00.9 (CCU61_CC62INC, IN) as S1 mission “CCU6_Tx”
- P02.8 (GPT12 T4INA) as S2 monitor “GPT12_Ta”
- P00.0 (TOM0_CH4, TOUT9, OUT) generate dummy PWM which feedback to S1 and S2

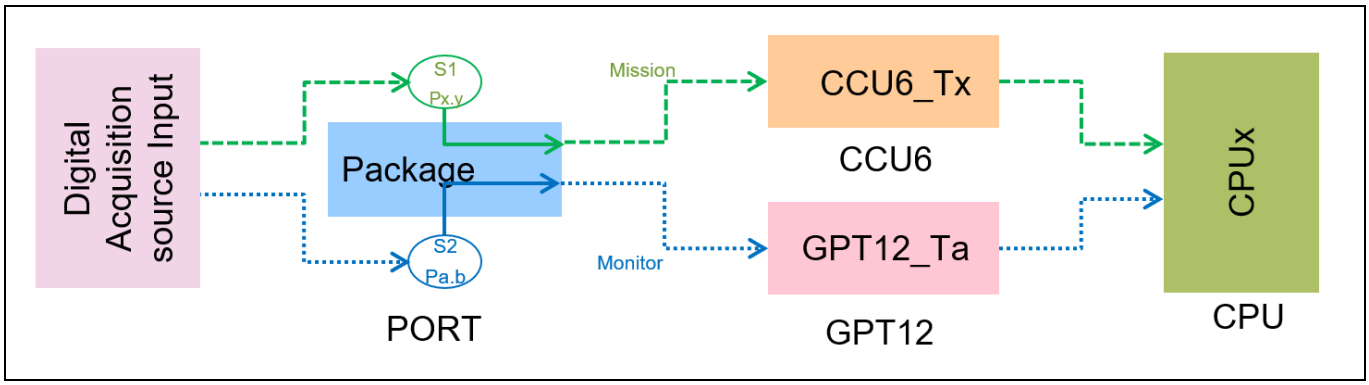


Figure 70 Safety Mechanism CCU6_CAPTURE_MON_BY_GPT12

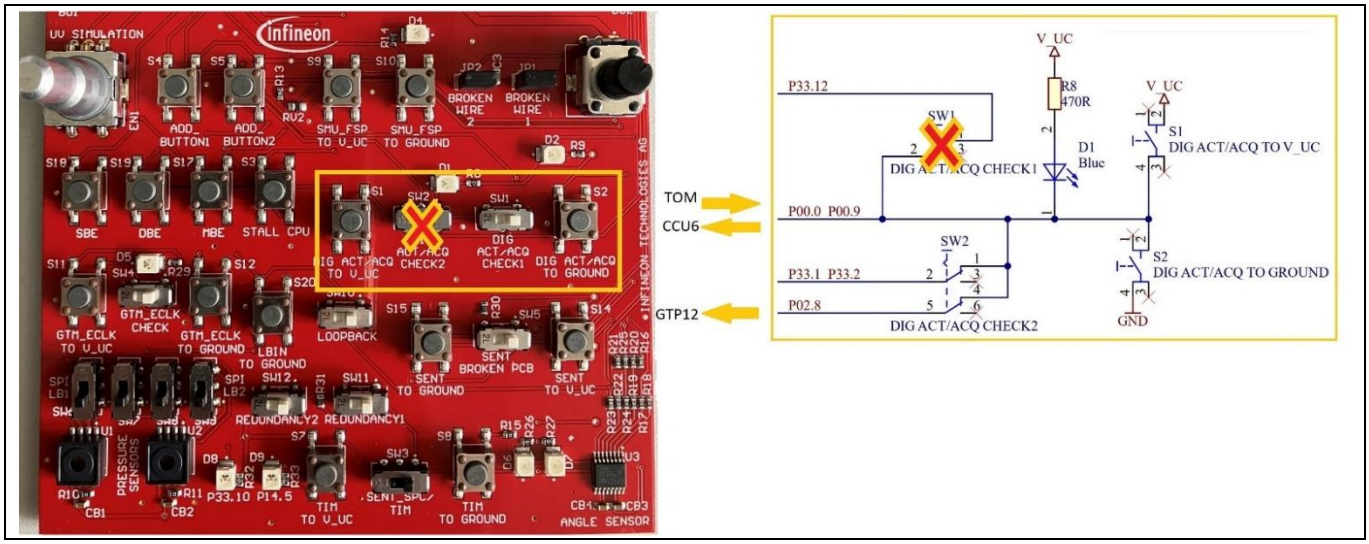


Figure 71 Digital acquisition with redundant CCU6/GTP12 channels including highlighted buttons and switches for fault injection (X == not used)

Figure 72 shows how to trigger the configuration of all modules involved in this FUC and shows the SMU alarm reaction if fault injection is triggered via the buttons or switches highlighted in Figure 71.

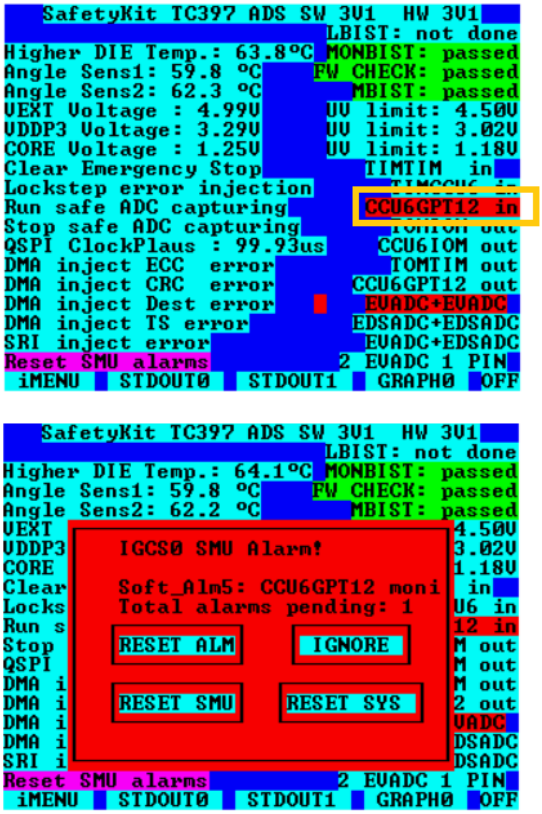


Figure 72 Digital acquisition with redundant CCU6/GTP12 Alarm pop-up after error injection

6.2.11.3 Digital actuation implementation

Digital actuation with redundant TOM channels and IOM comparison (FUC0)

In FUC0 of the safety-related function “digital actuation”, a safety-related digital output signal is generated with a GTM output resource (TOM or ATOM). This signal is connected to an external actuator and also fed back to another input port pin of the microcontroller. A second GTM output resource (TOM or ATOM) is used to generate an internal reference signal, which is compared by the input/output monitor (IOM) against the feedback signal. The *Safety Mechanism IOM_ALARM_CHECK* and *Safety Mechanism TIM_CLOCK_MONITORING* are required for this FUC.

To demonstrate this FUC with Application Kit Safety, a safe PWM output signal is generated by a TOM channel, and a redundant identical PWM is generated by another TOM channel. Both outputs are captured and monitored by the IOM. If the two outputs diverge, the IOM generates an alarm to the SMU. The application software should also check if the GTM is properly clocked. The implementation of that SM is also implemented according to the recommendations in the AURIX™ TC3xx Safety Manual [3].
An illustrative example of the *IOM_ALARM_CHECK* safety mechanism and an overview of the Application Kit Safety implementation of this FUC including *TIM_CLOCK_MONITORING* are shown in Figure 73 and Figure 74. This example is also referred as the “TOMIOM out” example. The port pins used for this SM are as follows:

- P00.0 (TOM0_CH4, TOUT9, OUT) as S1 mission “TOMx_CHy”
- T02.2 (TOM1_CH10, TOUT2, OUT) as S3 redundant internal signal “TOMa_CHb” (IOM Ref Input)
- P33.1 (IOM, IN) as S2 “IOM”

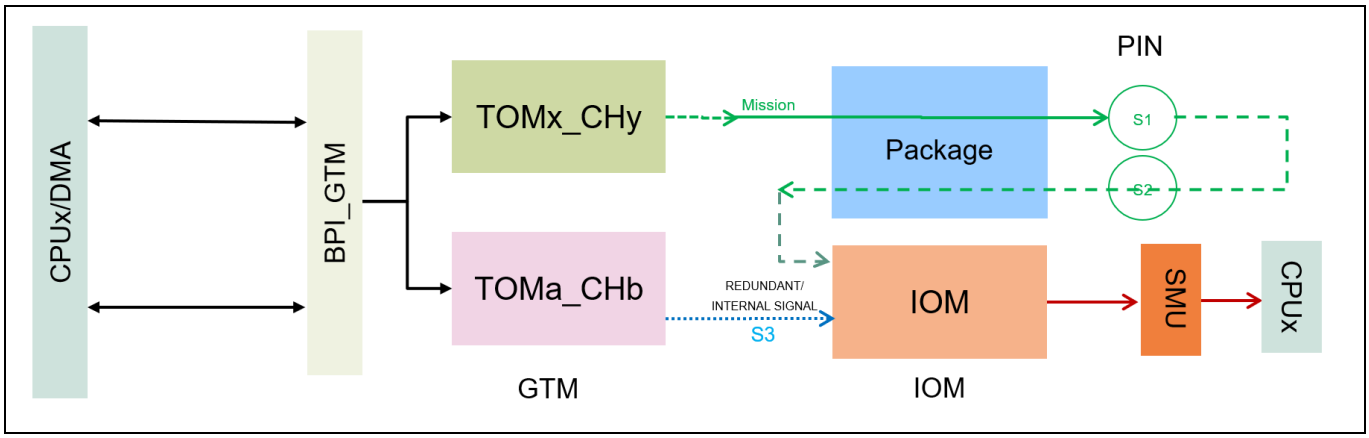


Figure 73 Illustrative example of safety mechanism IOM_ALARM_CHECK

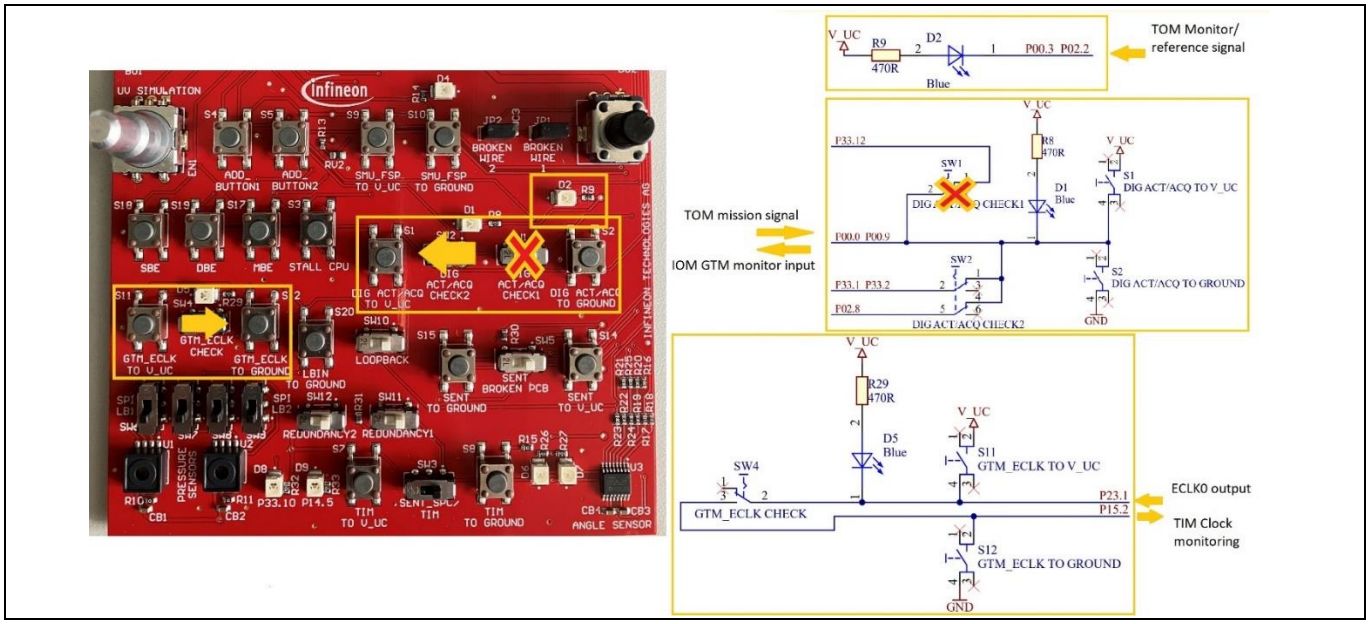


Figure 74 Digital actuation with redundant TOM channels and IOM comparison Application Kit Safety implementation overview including highlighted buttons and switches for fault injection

Note: See Figure 3 for more information on default states of the switches.

Figure 75 shows how to trigger the configuration of all modules involved in this FUC and also shows the SMU alarm response if fault injection is triggered via the buttons or switches highlighted in Figure 74.

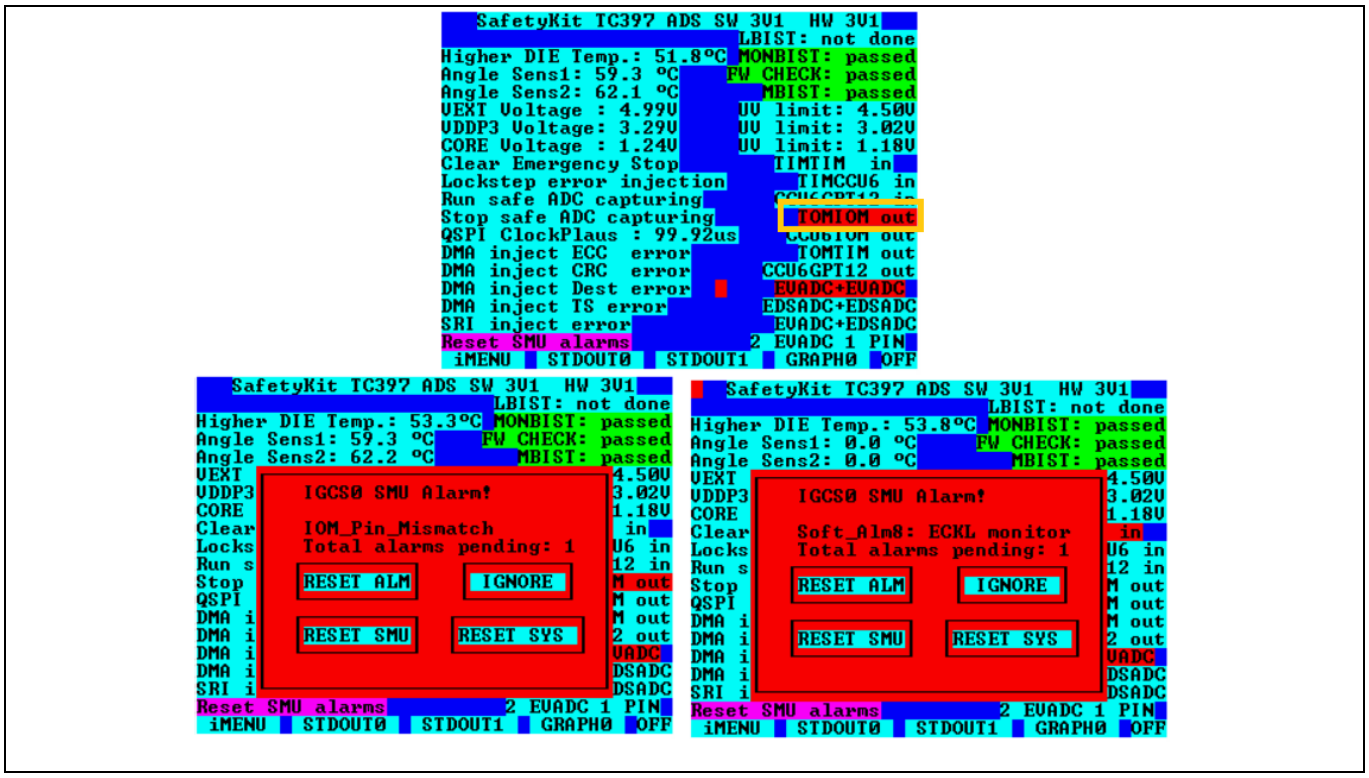


Figure 75 Digital actuation with redundant TOM channels and IOM comparison example selection at the top and alarm reaction to fault injection via switches or buttons

Digital actuation with redundant TOM/CCU6 channels and IOM comparison (FUC1)

In FUC1 of the safety-related function “digital actuation”, a safety-related digital output signal is generated with a GTM output resource (TOM or ATOM). This signal is connected to an external actuator and also fed back to another input port pin of the microcontroller. An CCU6 output resource is additionally used to generate an internal reference signal, which is compared by the input/output monitor (IOM) against the feedback signal. The *IOM_ALARM_CHECK* safety mechanism is required for this FUC.

To demonstrate this FUC with Application Kit Safety, a safe PWM output signal is generated by an ATOM channel and a redundant identical PWM is generated by a CCU6 channel. Both outputs are captured and monitored by the IOM. If the two outputs diverge, the IOM generates an alarm to the SMU.
An illustrative example of the *TOM_CCU6_MONITORING_WITH_IOM* safety mechanism and an overview of the Application Kit Safety implementation of this FUC are shown in Figure 76 and Figure 77. This example is also referred as the “CCU6IOM out” example. The port pins used for this SM are as follows:

- P00.0 (ATOM0_CH0, TOUT9, OUT) as S1 mission “TOMx_CHy”.
- P00.3 (CCU61_CC61, OUT) as S3 redundant (reference) internal signal “CCU6_Ta”
- P33.1 (IOM, IN) as S2 “IOM”

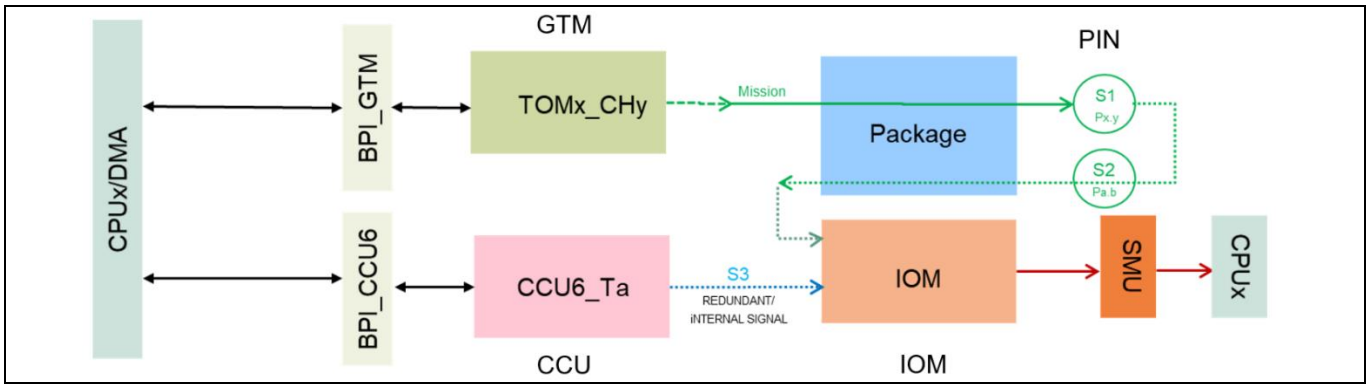


Figure 76 Illustrative example of Safety Mechanism TOM_CCU6_MONITORING_WITH_IOM

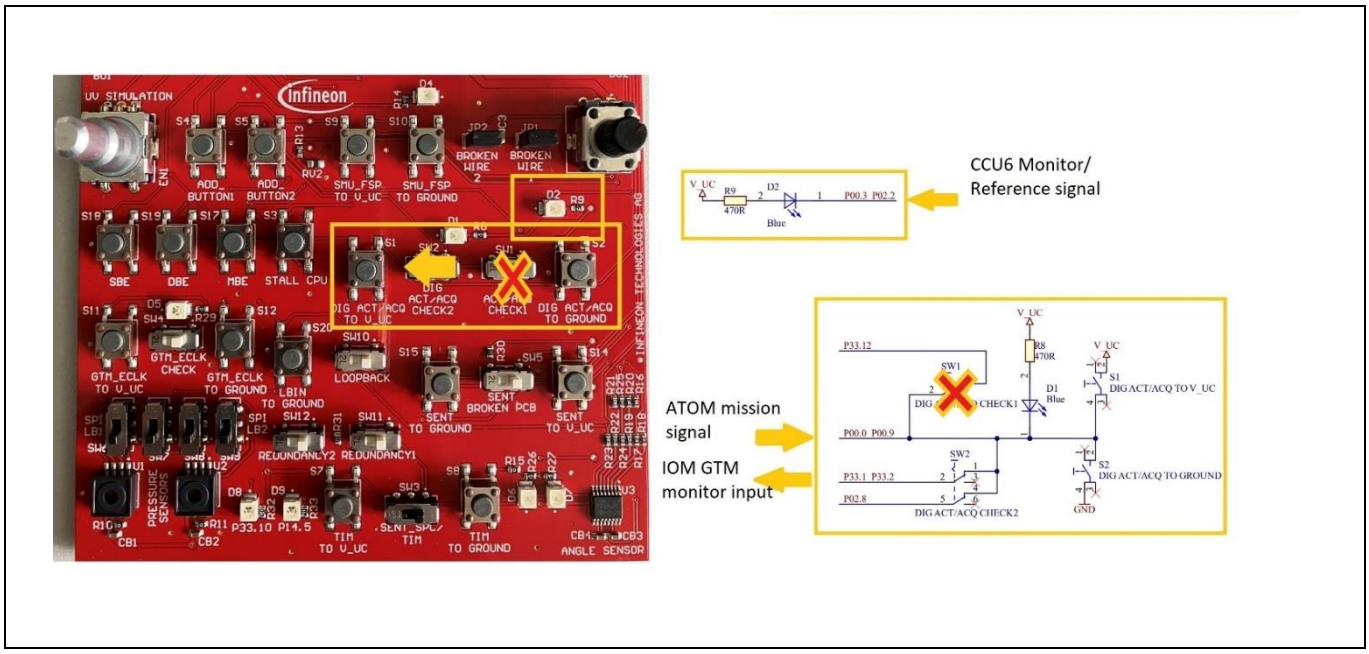


Figure 77 Digital actuation with redundant TOM/CCU6 channels and IOM comparison Application Kit Safety implementation overview including highlighted buttons and switches for fault injection

Note: See Figure 3 for more information on default states of the switches.

Figure 78 shows how to trigger the configuration of all modules involved in this FUC and also shows the SMU alarm reaction if fault injection is triggered via the buttons or switches highlighted in Figure 77.

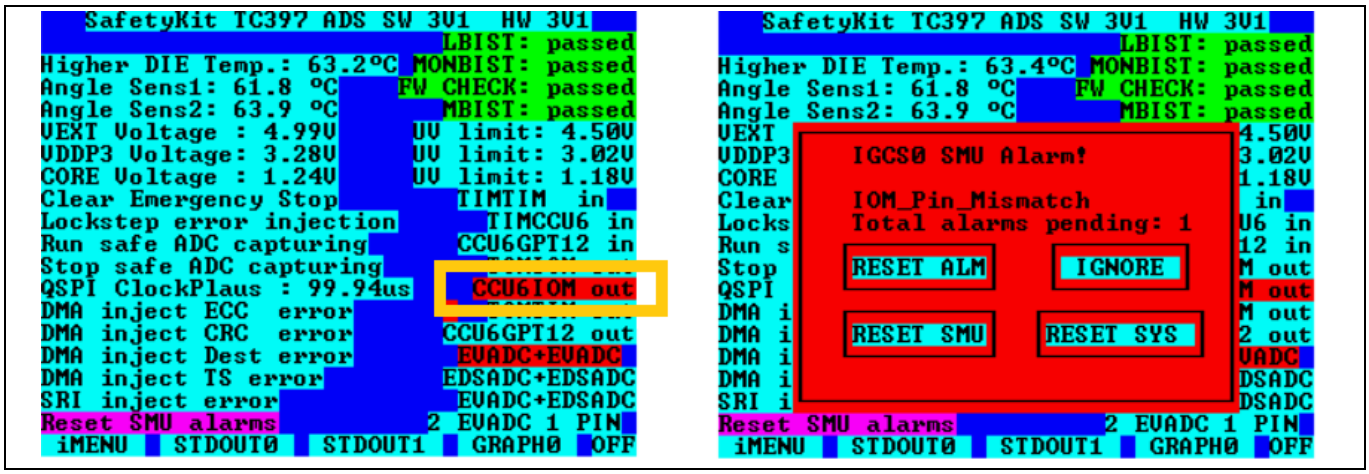


Figure 78 Digital actuation with redundant TOM/CCU6 channels and IOM comparison example selection at the top and alarm reaction to fault injection via switches or buttons

Digital actuation with redundant TOM/TIM channels and Application SW comparison (FUC2)

In FUC2 of the safety-related function “digital actuation”, a safety-related digital output signal request is either generated by the CPU or by the DMA. An GTM output resource (TOM or ATOM) is used to generate this signal. The signal is connected to an external actuator and also fed back to a GTM input resource (TIM). Finally, the application software performs a comparison of the requested PWM output signal to the digital feedback signal. The *TOM_TIM_MONITORING* and *TIM_CLOCK_MONITORING* safety mechanisms are required for this FUC.

To demonstrate this FUC with Application Kit Safety, a safe PWM output signal is generated by a TOM channel, which is fed back, captured, and monitored by a TIM channel. If the period or the duty cycle of the incoming PWM signal is not as expected, the CPU generates an alarm to the SMU. The application software should also check if the GTM is properly clocked. The implementation of that SM is also implemented according to the recommendations in the Safety Manual [3].
An illustrative example of the *Safety Mechanism TOM_TIM_MONITORING* and an overview of the Application Kit Safety implementation of this FUC including the *TIM_CLOCK_MONITORING* safety mechanism are shown in Figure 79 and Figure 80.

This example is also referred as the “TOMTIM out” example. The port pins used for this SM are as follows:

- P00.0 (TOM0_CH4, TOUT9, OUT) as S1 mission “(A)TOMa_CHb”
- P02.8 (TIM3_CH0, IN) as S2 monitor “TIMxCHy”

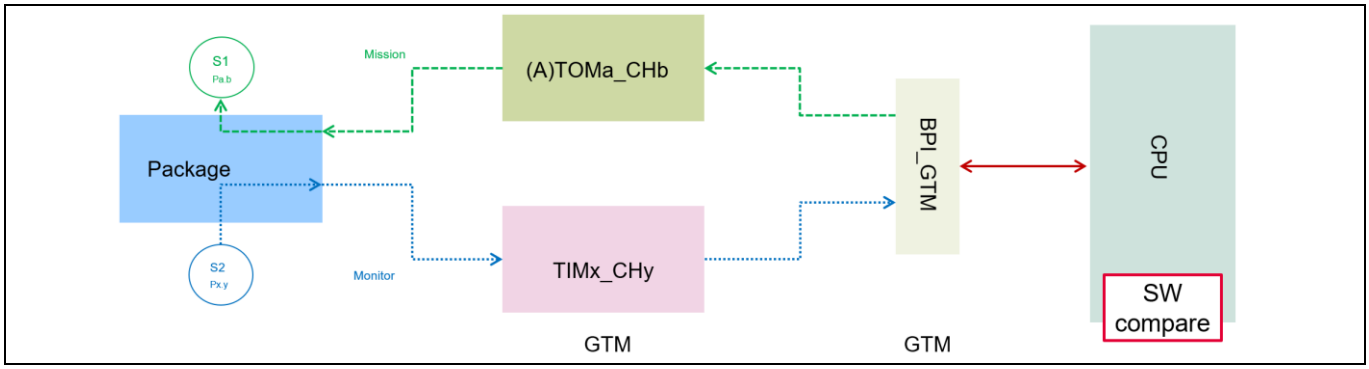


Figure 79 Illustrative example of Safety Mechanism TOM_TIM_MONITORING

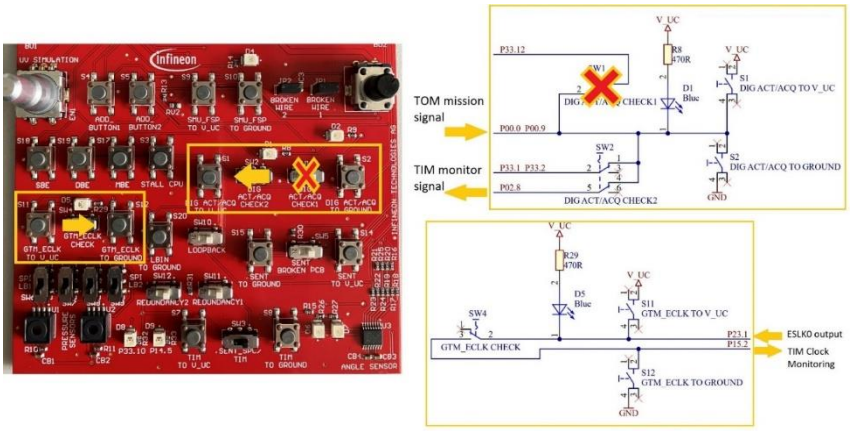


Figure 80 **Digital actuation with redundant TOM/TIM channels and application SW comparison Application Kit Safety implementation overview including highlighted buttons and switches for fault injection**

Note: *See Figure 3 for more information on default states of the switches.*

Figure 81 shows how to trigger the configuration of all modules involved in this FUC and also shows the SMU alarm reaction if fault injection is triggered via the buttons or switches highlighted in Figure 80.

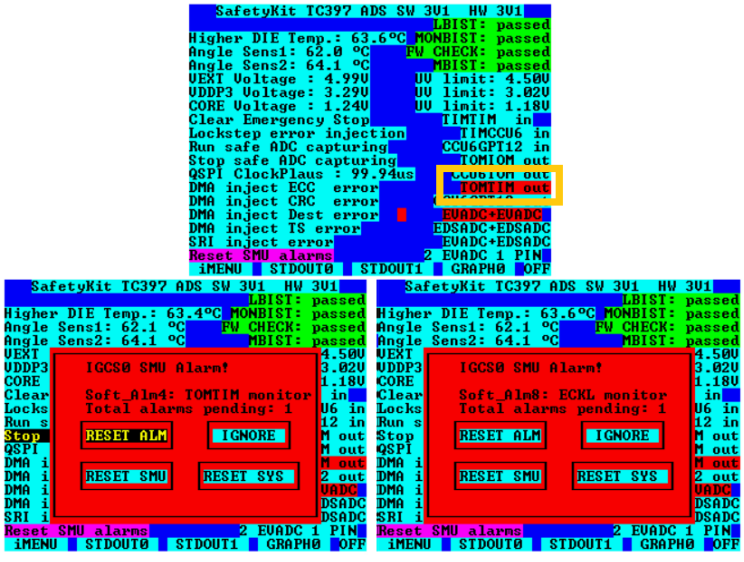


Figure 81 **Digital actuation with redundant TOM/TIM channels and application SW comparison example selection at the top and alarm reaction to fault injection via switches or buttons**

Digital actuation with redundant CCU6/GPT12 channels and application SW comparison (FUC3)

In FUC3 of the safety-related function “digital actuation”, a safety-related digital output signal request is either generated by the CPU or by the DMA. An CCU6 output resource is used to generate this signal. The signal is connected to an external actuator and fed back to a GTP12. Finally, the application software performs a comparison of the requested PWM output signal to the digital feedback signal. The *Safety Mechanism CCU6_GPT12_MONITORING* is required for this FUC.

To demonstrate this FUC with Application Kit Safety, a safe PWM output signal is generated by a CCU6 channel, which is fed back, captured, and monitored by a GPT12 channel. If the period or the duty cycle of the incoming PWM signal is not as expected, the CPU generates an alarm to the SMU. The implementation of that SM is also implemented according to the recommendations in the Safety Manual [3].

An illustrative example of the *Safety Mechanism CCU6_GPT12_MONITORING* and an overview of the Application Kit Safety implementation of this FUC are shown in Figure 82 and Figure 83. This example is also referred as the "CCU6GPT12 out" example. The port pins used for this SM are as follows:

- P00.0 (CCU60_COUT63, OUT) as S1 mission CCU6_Tx"
- P02.8 (GPT12 T4INA) as S2 monitor "GPT12_Ta"

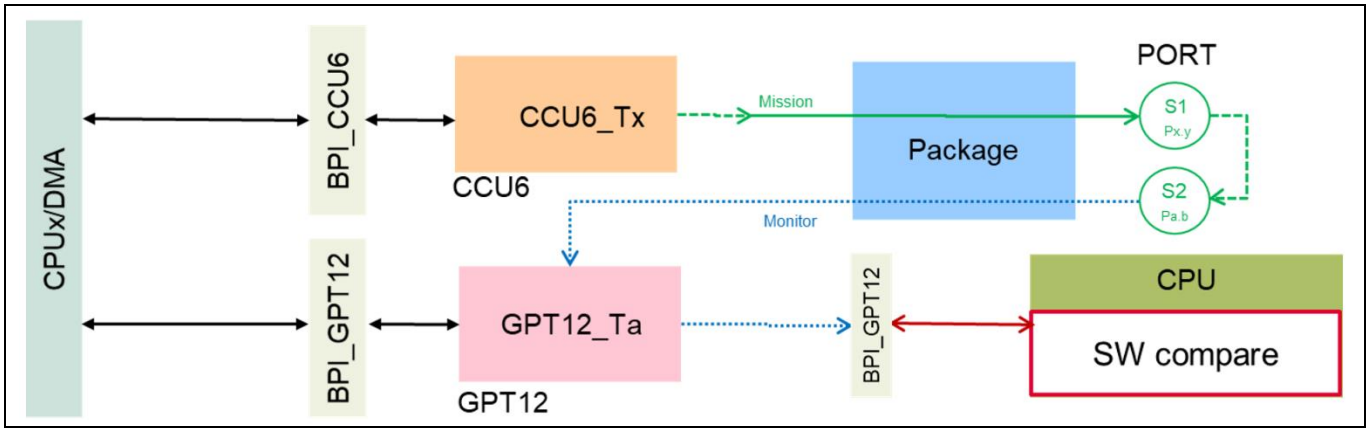


Figure 82 Safety Mechanism CCU6_GPT12_MONITORING Overview

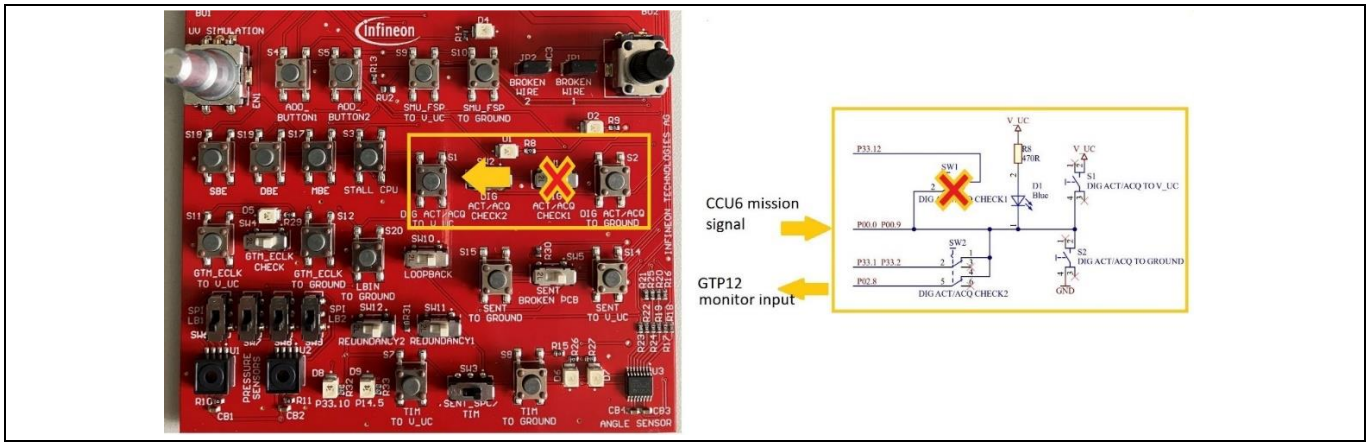


Figure 83 Digital actuation with redundant CCU6/GPT12 channels and application SW comparison Application Kit Safety implementation overview including highlighted buttons and switches for fault injection

Figure 84 shows how to trigger the configuration of all modules involved in this FUC and also shows the SMU alarm reaction if fault injection is triggered via the buttons or switches highlighted in Figure 83.

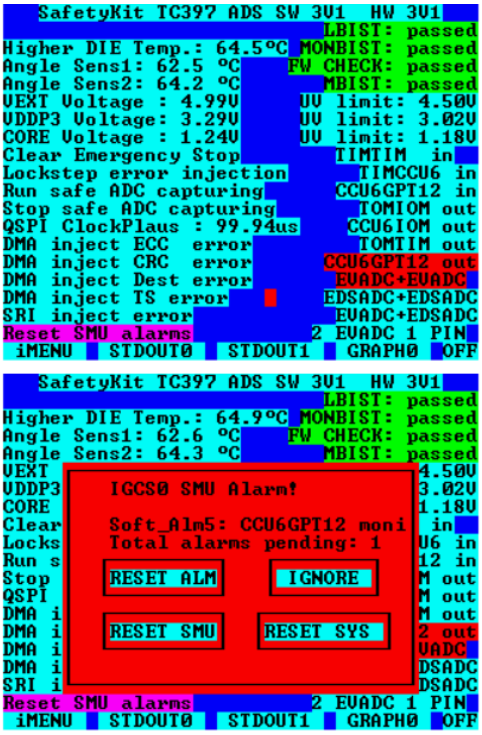


Figure 84 **Digital actuation with redundant CCU6/GPT12 channels and application SW comparison example selection at the top and alarm reaction to fault injection via switches or buttons**

6.2.12 MCU function – Signal processing powertrain

6.2.12.1 AMU.LMU_DAM

See Section 6.2.3.3.1.

6.2.13 MCU function – Safety mechanism

6.2.13.1 Safety Management Unit (SMU)

The SMU centralizes all the alarm signals related to the different hardware and software-based safety mechanisms. Each alarm can be individually configured to trigger internal actions and/or notify externally the presence of faults via a Fault Signaling Protocol (FSP). The SMU is composed of two main parts:

- **SMU core:** Located in the core domain, it responds to all alarms generated by modules supplied with the System Peripheral Bus (SPB) clock.
- **SMU stdby:** Located in the standby domain, it responds to all alarms generated by modules supplied with the BACK clock.

Multiple safety mechanisms for managing failures and to activate the required response are available:

- Safety Mechanism SMU:RT
- Safety Mechanism SMU:CCF_MONITOR
- Safety Mechanism SMU:ALIVE_MONITOR
- Safety Mechanism SMU:FSP_MONITOR

- Safety Mechanism SMU:FPI_WRITE_MONITOR

Additionally, the system developer oversees implementing initial checks on the SMU:

- Safety Mechanism MCU_FW_CHECK
- Safety Mechanism ALIVE_ALARM_TEST

For more details on SMU, see sections 3 and 4.

References

Contact Infineon Support to obtain these documents.

[1] Infineon Technologies AG: *AURIX™ TC3xx User's Manual part 1 and part 2*;

[2] Infineon Technologies AG: *AURIX™ TC39x-B User's Manual Appendix V2.0.0*

[3] Infineon Technologies AG: *AURIX™ TC3xx Safety Manual v2.0*

[4] Infineon Technologies AG: *TLF35584 Datasheet Rev. 2.0*

[5] Infineon Technologies AG : *Application Kit TC3X7 Manual*

Revision history

Document revision	Date	Description of changes
V1.0	2024-04-04	Initial version

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2024-04-04

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

AP32597

Important notice

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Please note that this product is not qualified according to the AEC Q100 or AEC Q101 documents of the Automotive Electronics Council.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.