Safe application development for AURIX™ TC375 safety lite kit

32-bit TriCore™ AURIX™ TC3xx microcontroller

About this document

Scope and purpose

This application note describes product safety mechanisms and the actions that shall 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™ TC3xx User's Manual [1] and the AURIX™ TC3xx Safety Manual [2]. The associated source code includes methods for fault injection, which is used for testing of safety mechanisms, alarm triggering, and the implementing software safety measures which are listed in the AURIX™ TC3xx Safety Manual.

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

Intended audience

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

Table of contents

About this document **1**
Table of contents **2**
1 Introduction **6**
1.1 Key features 6
2 Demonstrator **9**
2.1 Hardware overview 9
2.2 Software overview 10
2.3 Infineon's OneEye GUI 12
3 Boot and startup procedure **15**
3.1 Analog power-up 15
3.2 Boot firmware 15
3.2.1 Startup software (SSW) 15
3.2.2 Checker software (CHSW) 15
3.2.3 Bootstrap loaders (BSL) 16
3.3 Application SW startup 16
3.3.1 Implementation of application SW startup 17
3.3.2 OneEye view of application SW startup 17
3.3.3 Reset triggering 19
3.3.3.1 OneEye status flags after resets 19
3.3.4 Logic built-in self-test (LBIST) 21
3.3.4.1 Implementation 22
3.3.4.2 OneEye interface 22
3.3.5 Monitor built-in self-test (MONBIST) 22
3.3.5.1 Implementation 22
3.3.5.2 OneEye interface 23
3.3.6 Firmware check 23
3.3.6.1 Implementation 23
3.3.6.2 OneEye interface 26
3.3.7 MCU_STARTUP 26
3.3.7.1 Implementation 27
3.3.7.2 OneEye interface 27
3.3.8 SMU ALIVE_ALARM_TEST 27
3.3.8.1 Implementation 27
3.3.8.2 OneEye interface 27
3.3.9 SMU REG_MONITOR_TEST 28
3.3.9.1 Implementation 28
3.3.9.2 OneEye interface 28
3.3.10 Memory built-in self-test (MBIST) 28
3.3.10.1 Implementation 28
3.3.10.2 OneEye interface 28
3.3.11 Enable all SMU alarms 29
4 Safety management unit (SMU) **30**
4.1 Implementation 31
4.1.1 SMU driver 31
4.1.2 SMU application software alarm 31
4.2 OneEye interface 32
4.2.1 OneEye SMU script file 32

Table of contents

4.2.2 Status of SMU driver ... 33
5 Safety related function ... 34
5.1 Register monitor test ... 34
5.1.1 Implementation ... 34
5.1.2 OneEye interface ... 34
5.2 Safe computation NLS-CPU ... 35
5.2.1 CPU safe computation NLS-CPU concept ... 35
5.2.2 Access protection configuration SMs ... 37
5.2.2.1 Implementation ... 37
5.2.2.2 OneEye interface ... 38
5.2.3 Software based self-test (SBST) ... 38
5.2.3.1 Implementation ... 38
5.2.3.2 OneEye interface ... 39
5.2.4 Internal bus SMs ... 39
5.2.4.1 Implementation ... 39
5.2.4.2 OneEye interface ... 39
5.2.5 Monitor and supervision SMs ... 39
5.2.5.1 Implementation ... 39
5.2.5.2 OneEye interface ... 41
5.3 Safe computation LS-CPU ... 41
5.3.1 Safe computation LS-CPU concept ... 41
5.3.2 Program flash - NVM ... 42
5.3.2.1 Implementation ... 42
5.3.2.2 OneEye interface ... 43
5.3.3 Flexible CRC Engine (FCE) ... 44
5.3.3.1 Implementation ... 44
5.3.3.2 OneEye interface ... 44
5.3.4 Direct Memory Access (DMA) ... 44
5.3.4.1 Implementation ... 45
5.3.4.2 OneEye interface ... 45
5.3.5 System Resources Interconnect (SRI) ... 46
5.3.5.1 Implementation ... 46
5.3.5.2 OneEye interface ... 46
5.3.6 System Timer (STM) ... 47
5.3.6.1 Implementation ... 47
5.3.6.2 OneEye interface ... 47
5.4 Digital acquisition ... 48
5.4.1 Overview ... 49
5.4.1.1 User interface ... 49
5.4.1.2 Software structure ... 50
5.4.2 Redundant TIM/TIM channels (FUC0) ... 51
5.4.2.1 Implementation ... 51
5.4.2.2 OneEye interface ... 52
5.4.3 Redundant TIM/TIM channels and HW diversity (FUC1) ... 52
5.4.3.1 Implementation ... 52
5.4.3.2 OneEye interface ... 53
5.4.4 Redundant CCU6/TIM channels (FUC2) ... 53
5.4.4.1 Implementation ... 54
5.4.4.2 OneEye interface ... 54
5.4.5 Redundant CCU6/GTP12 channels (FUC3) ... 55

Table of contents

5.4.5.1 Implementation 55
5.4.5.2 OneEye interface 56
5.4.6 Single CCU6/GPT12 channels (FUC4) 56
5.4.6.1 Implementation 57
5.4.6.2 OneEye interface 57
5.5 Digital actuation 58
5.5.1 Overview 59
5.5.1.1 User interface 59
5.5.1.2 Software overview 60
5.5.2 Redundant TOM channels and IOM comparison (FUC0) 60
5.5.2.1 Implementation 60
5.5.2.2 OneEye interface 62
5.5.3 Redundant TOM/CCU6 channels and IOM comparison (FUC1) 62
5.5.3.1 Implementation 62
5.5.3.2 OneEye interface 63
5.5.4 Redundant TOM/TIM channels and application SW comparison (FUC2) 63
5.5.4.1 Implementation 64
5.5.4.2 OneEye interface 65
5.5.5 Redundant CCU6/GPT12 channels and application SW comparison (FUC3) 65
5.5.5.1 Implementation 66
5.5.5.2 OneEye interface 66
5.6 Analog acquisition 67
5.6.1 Overview of analog signals 68
5.6.1.1 User interface 69
5.6.1.2 Software structure 69
5.6.2 Redundant EVADC channels (FUC0) 70
5.6.2.1 Implementation 70
5.6.2.2 OneEye interface 71
5.6.3 Redundant EDSADC channels (FUC1) 71
5.6.3.1 Implementation 71
5.6.3.2 OneEye interface 73
5.6.4 Redundant EVADC and EDSADC channels (FUC2) 73
5.6.4.1 Implementation 73
5.6.4.2 OneEye interface 74
5.6.5 Single pin with EVADC channels (FUC3) 75
5.6.5.1 Implementation 75
5.6.5.2 OneEye interface 75
5.6.6 Single pin with EVADC and EDSADC channels and broken Wire (FUC4) 76
5.6.6.1 Implementation 76
5.6.6.2 OneEye interface 77
5.7 Avoidance or detection of CCF 77
5.7.1 PORT 78
5.7.1.1 Implementation 78
5.7.1.2 OneEye interface 79
5.7.2 Clock 80
5.7.2.1 Implementation 80
5.7.2.2 OneEye interface 80
5.7.3 Die temperature sensors (DTS) 81
5.7.3.1 Implementation 81
5.7.3.2 OneEye interface 81

Table of contents

5.8 External communication 81
5.8.1 QSPI 81
5.8.1.1 Implementation 82
5.8.1.2 OneEye interface 83
5.9 Coexistence of HW/SW elements 83
5.9.1 Interrupt Router (IR) 83
5.9.1.1 Implementation 83
5.9.1.2 OneEye interface 84
References 85
Revision history 86
Disclaimer 87

1 Introduction

The development of a safe application for a wide range of automotive and industrial applications can be a challenge due to the need to follow strict safety rules. To ease the development of such applications, Infineon provides an AURIX™ TC3xx Safety Manual [2], which defines so called 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 (SM) are classified in two main categories:

- Technical solution internal to the MCU, implemented by hardware (HW) or software (SW)
- Technical solution either in HW or SW, implemented at *System level* by the system integrator

During the design phase of AURIX™ TC3xx, most common use cases have been taken, and safety requirements were derived from them. For the implementation of these specific safety-related functions, different SMs shall be implemented according to the specification in the AURIX™ TC3xx Safety Manual. The specific SM is required depends on the modules used and the required safety level.

Hence, the intention of this application note is to provide implementation hints and demo code examples for many of these safety mechanisms. Therefore, this document is provided with example software optimized for the safety hardware. For this purpose, Infineon's AURIX™ TC375 Lite Kit V2 (KIT_A2G_TC375_LITE) is used as the safety hardware with the newly implemented safety software, combined together called the AURIX™ TC375 safety lite kit (SLK). The kit demonstrates the implementation of the SMs and other diagnostics information. Additionally, a graphical user interface (GUI) through OneEye provides the injection of faults and the status of the AURIX™ TC375 in real-time. An overview about all safety-related functions and SMs covered by the application note can be found in the tables Table 1 and Table 2.

1.1 Key features

The following lists the implemented and supported key features in AURIX™ TC375 safety lite kit:

- Boot and startup procedure including all safety mechanisms involved
- SMU driver implementation
- Implementation of various safety-related functions and required safety mechanisms
- Software-based self-test (SBST) support (dummy implementation)
- Safe computation on lockstep and non-lockstep cores
- Analog acquisition
- Digital acquisition and digital actuation
- External communication, i.e., QSPI
- Safe state support
- Avoidance or detection of common cause failure (CCF)
- Coexistence of HW/SW elements
- Register monitor test
- Fault injection for testing various safety mechanisms
- OneEye GUI for user 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 (FUCs)
Safe Computation LS-CPU	Yes	• Safety Mechanism STM:* • Safety Mechanism NVM:* • Safety Mechanism DMA.*:* • Safety Mechanism SRI.*:* • Safety Mechanism CPU.*:* • Safety Mechanism AMU.*:* • Safety Mechanism SRI.*:*
Safe Computation NLS-CPU	Yes	• Safety Mechanism CPU.*:*
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 TIM/TIM channels and HW diversity • FUC 2: Digital Acquisition with redundant CCU6/TIM channels • FUC 3: Digital Acquisition with redundant CCU6/GPT12 channels • FUC 4: Digital Acquisition with single 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
External Communication	Partly	• Safety Mechanism QSPI:SAFE_COMMUNICATION

Safety-related function	Covered (Yes / Partly / No)	Covered SMs / Functional Use Case (FUCs)
Avoidance or Detection of Common Cause Failures	Partly	• Safety Mechanism DTS:DTS_RESULT • Safety Mechanism SYS:WATCHDOG_FUNCTION • Safety Mechanism PMS:MONBIST_RESULT
Safe State Support	Partly	• Internal Failure Reporting • External Failure Reporting (FSP) • Alternate Failure Reporting • (As Safety Mechanism SYS:FSP_ERROR_PIN_MONITOR is implemented and “At least one of the Safety Mechanism SYS:* shall be implemented” is stated in AURIX™ TC3xx Safety Manual it could be treated as “Covered -Yes”)
Coexistence of HW/SW elements	yes	• Safety Mechanism IR:ISR_MONITOR

Table 2 Overview about the implemented Safety Mechanisms (Software) - Safe Startup

Safety mechanism (SM)	Application note section	C function name
LBIST_CFG, LBIST_MONITOR and LBIST_RESULT	3.3.4	`slkSswLbist`
MONBIST_CFG	3.3.5	`slkMonbist`
MONBIST_RESULT	3.3.5	`slkMonbistCheck`
MCU_FW_CHECK	3.3.6	`slkSswMcuFwCheck`
MCU_STARTUP	3.3.7	`slkSswMcuStartup`
ALIVE_ALARM_TEST	3.3.8	`slkSswAliveAlarmTest`
SMU:REG_MONITOR_TEST	3.3.9	`slkSswSmuRegMonitorTest`
MBIST	3.3.10	`slkSswMbist`

Note: For all the abbreviations, acronyms, and safety-related definitions, see the documents: [1], [2], and [3].

2 Demonstrator

2.1 Hardware overview

The existing AURIX™ TC375 lite kit hardware [5] is used as a safety Lite Kit hardware board. In some safety measure, it is required to make pin connection via connecting cables, and these are explained in each section later in this document.

AURIX™ lite kit is equipped with a 32-bit single-chip AURIX™ TriCore™ based microcontroller AURIX™ TC375 which offers 3 cores, two of which are lockstep cores, running at 300 MHz.

The whole overview specifications of the board are summarized in Table 3.

Table 3 AURIX™ TC375 lite kit specification

CPU Core AURIX™	**Manufacture order no: SAK-TC375TP-96F300W AA**
Board dimension	66.0 x 131.0 mm
Power	• On-board miniWiggler Micro-AB USB interface • External powering 5 V…40 V (recommended 7 V...14 V)
Connector	• Most AURIX™ pins are available on expansion connectors (X1, X2) • Two Infineon Shield2Go connectors • Arduino compatible connectors for 3.3 V • mikroBUS connector • Micro-USB connector • DAP Debug connector • CAN connector • RJ45 connector
Other	• CAN transceiver TLE9251VSJ from Infineon • Low Power 10/100 Mbps Ethernet Physical Layer Transceiver DP83825I from TI • 1 user push-button, 3 user LEDs • Reset push-button • Potentiometer (10 kOhm) for variable analog input

Figure 1 shows the TC375 lite kit board. For a detailed overview on this board, see the TC375 lite kit user manual [5].

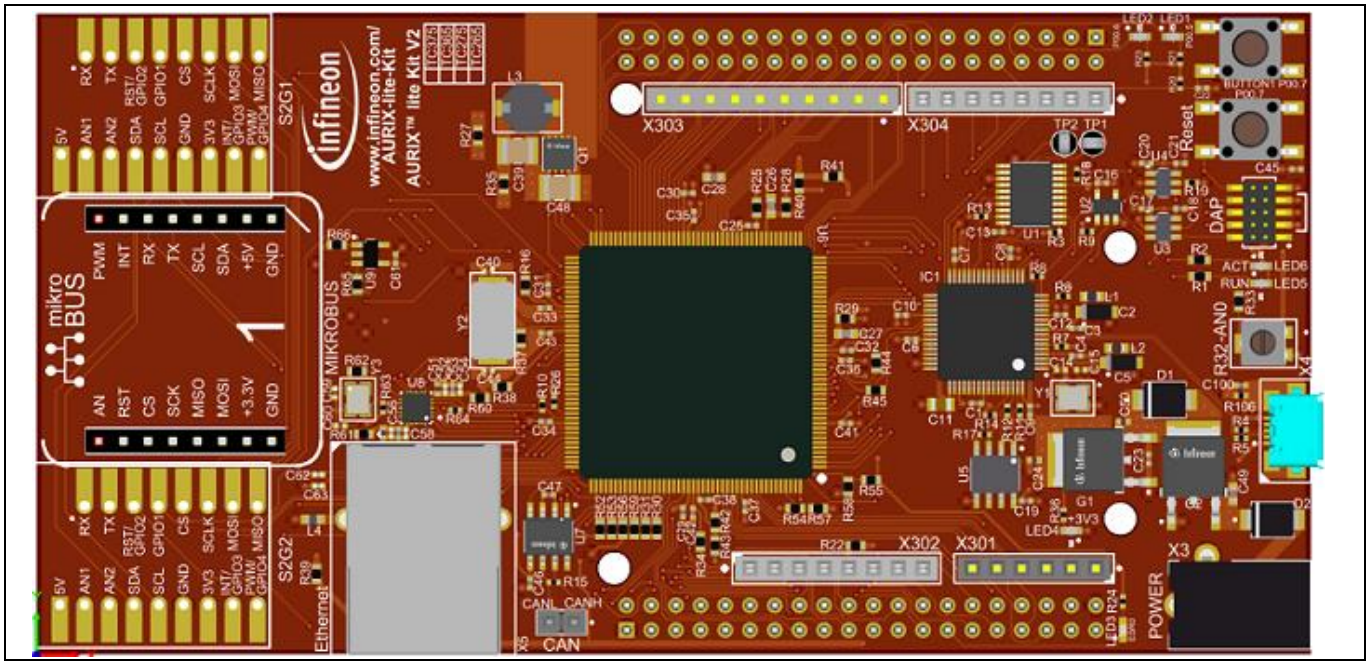

Figure 1 AURIX™ TC37x safety lite kit

2.2 Software overview

The software for AURIX™ TC37x safety lite kit is based on a free-of-charge Integrated Development Environment (IDE) from Infineon technologies, called the AURIX™ Development Studio (ADS). Note that it has not been tested on other IDE platforms. In addition, some software implementations are based on other AURIX™ - TC3xx Microcontroller Expert-Trainings, which are available on the Infineon Webpage.

The ADS software's project folder structure is shown in Figure 2. Besides the three *CpuX_Main.c (x=0...2)* files, the relevant application software is stored in the *SafetyLiteKit* folder, which contains all the code of the safety features implementation. The detailed description of the software is fully explained in each section in this application note.

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

- */Configurations/Ifx_Cfg_Ssw.c*
- */Configurations/Ifx_Cfg_Ssw.h*
- */Configurations/Ifx_Cfg_SswBmhd.c*
- */Configurations/Ifx_Cfg.h*
- *SafetyLiteKit/Slk_Cfg.h*

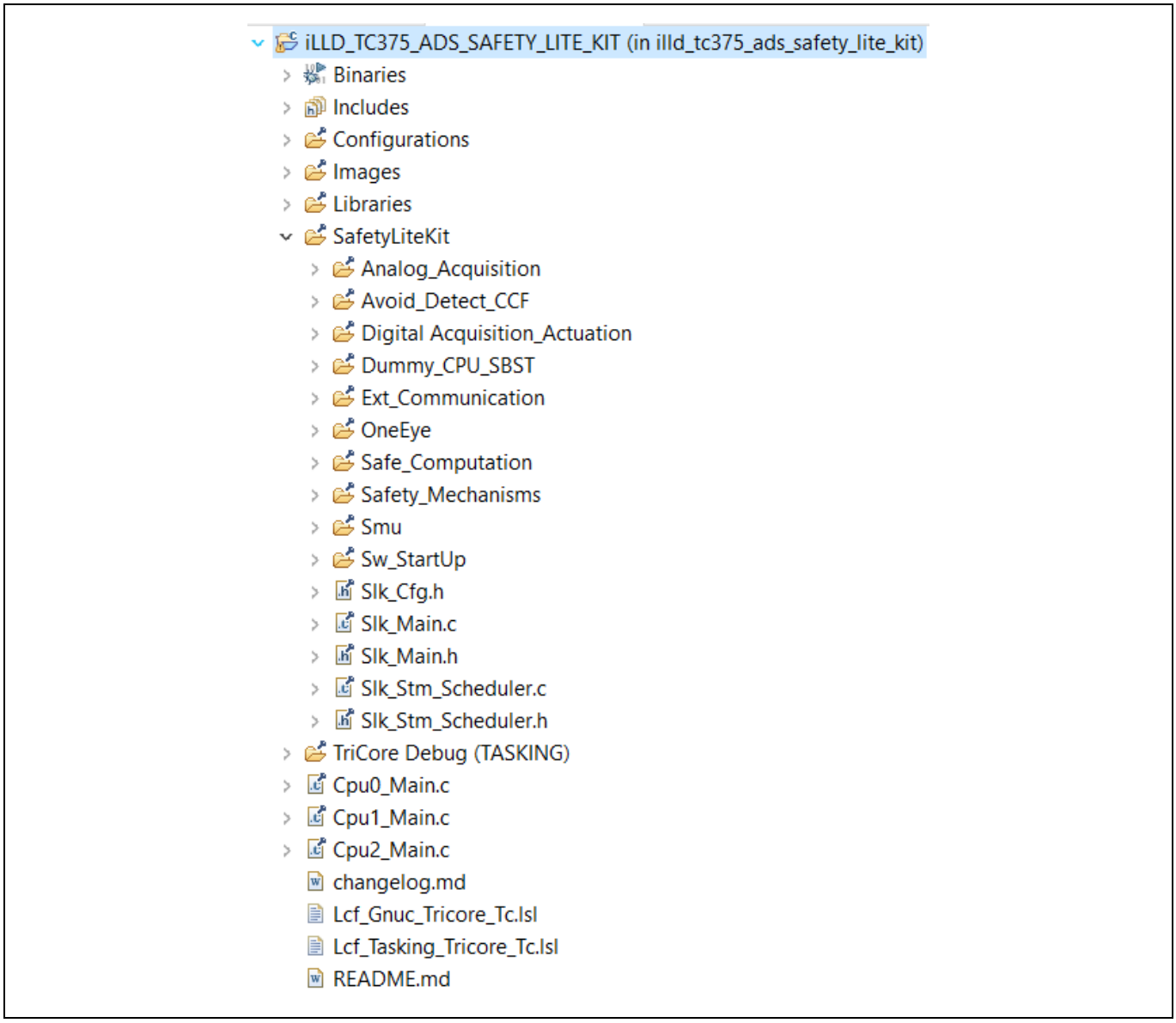

Figure 2 Project folder structure of ADS

The ADS project is available online and can be either downloaded from Infineon's GitHub repository or imported directly into ADS.

Initially, connect the AURIX™ TC37x Safety lite Kit board via an USB cable to the PC. After connecting the board, the project folder needs to be added to the ADS workspace, and it must be set as the active project. Then, to launch the OneEye GUI, it is necessary to first flash the project (by clicking button 1, as shown in the Figure 3) to the board and then launch the OneEye GUI (button 2, as shown in the Figure 3).

Alternatively, after flashing the device, it is possible to launch the OneEye application and then open the "iLLD_TC375_ADS_SAFETY_LIT_LITE.OneEye" file, which is located in the *libraries* folder of the project.

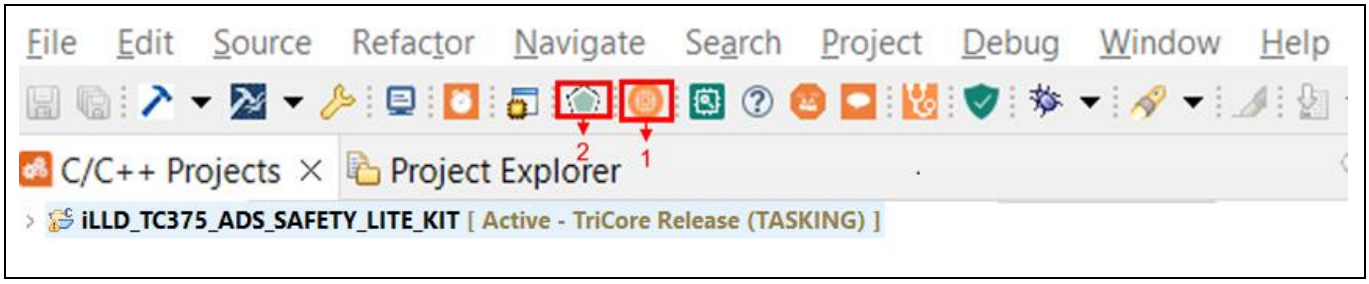

Figure 3 Launch OneEye GUI

To launch the debugger, it is necessary to first build (button 1) the project and then launch the debugger (button 2) as shown in the Figure 4. For the debugger configuration, it is recommended to use the winIDEA configuration. The main global variable to watch is *g_SlkStatus*, which is located in */SafetyLiteKit/Slk_Main.c* folder.

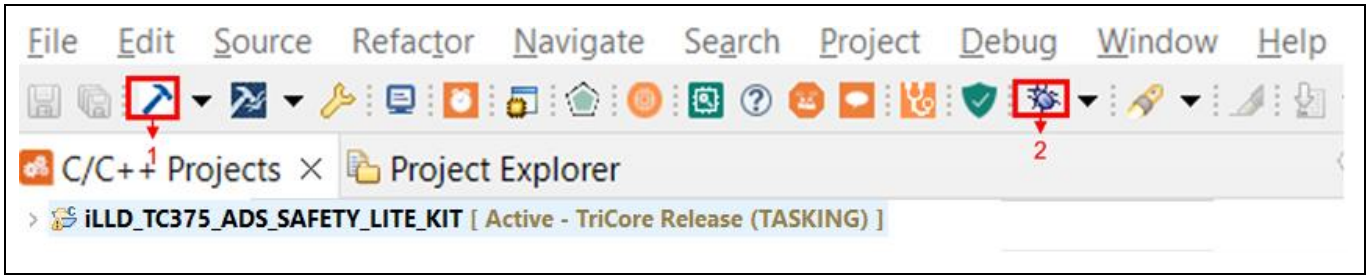

Figure 4 Launch debugger

2.3 Infineon's OneEye GUI

The GUI used for this project is called OneEye, which is a front-end software used to create a custom GUI for embedded application. This GUI interface is integrated and developed for the AURIX™ Development Studio.

The system operates in real-time and communicates through various interfaces such as UART, Ethernet, CAN, or DAS. The interface shows the output of global variables that are linked to the implemented safety measures. Additionally, errors can be injected via the GUI and observe the corresponding safety safe reactions.

The interface is composed of two menus as describe in Figure 5 and Figure 6:

Menu 1:

- Status of different safety mechanisms of the safe startup sequence
- Status of safe computation NLS-CPU (non-lockstep CPU) safety mechanisms and error injection
- Status of safe computation LS-CPU (lockstep CPU) safety mechanisms and error injection
- Status of SMU and user interaction as reset alarm, reset SMU, and reset system
- Show board status i.e., temperature, board alive LED, and alarm detection LED
- Show the status of external communication and avoidance or detection of common cause failure safety mechanisms

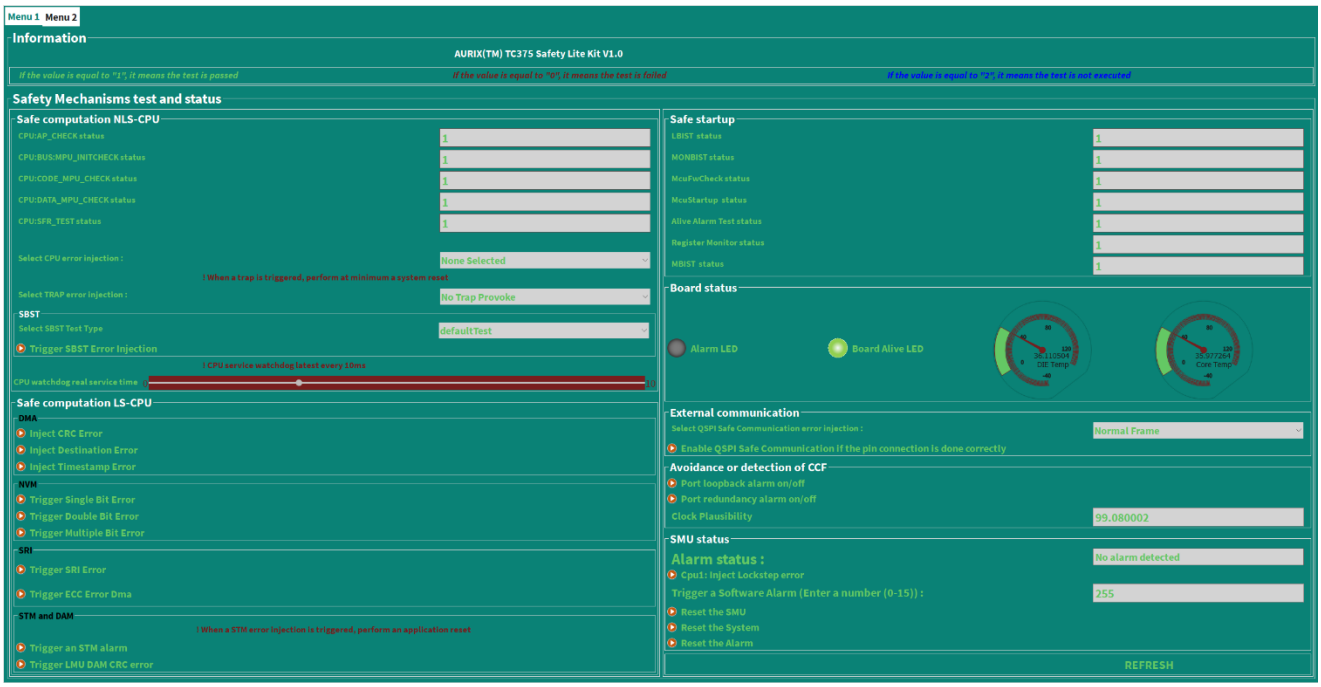

Figure 5 OneEye GUI-Menu 1

Menu 2:

- Show the status of detailed safe startup sequence, i.e., LBIST, MBIST, etc.
- Digital acquisition and actuation function use cases
- Analog acquisition function use cases
- SMU status and user interaction to reset alarm, reset SMU, and reset the complete system

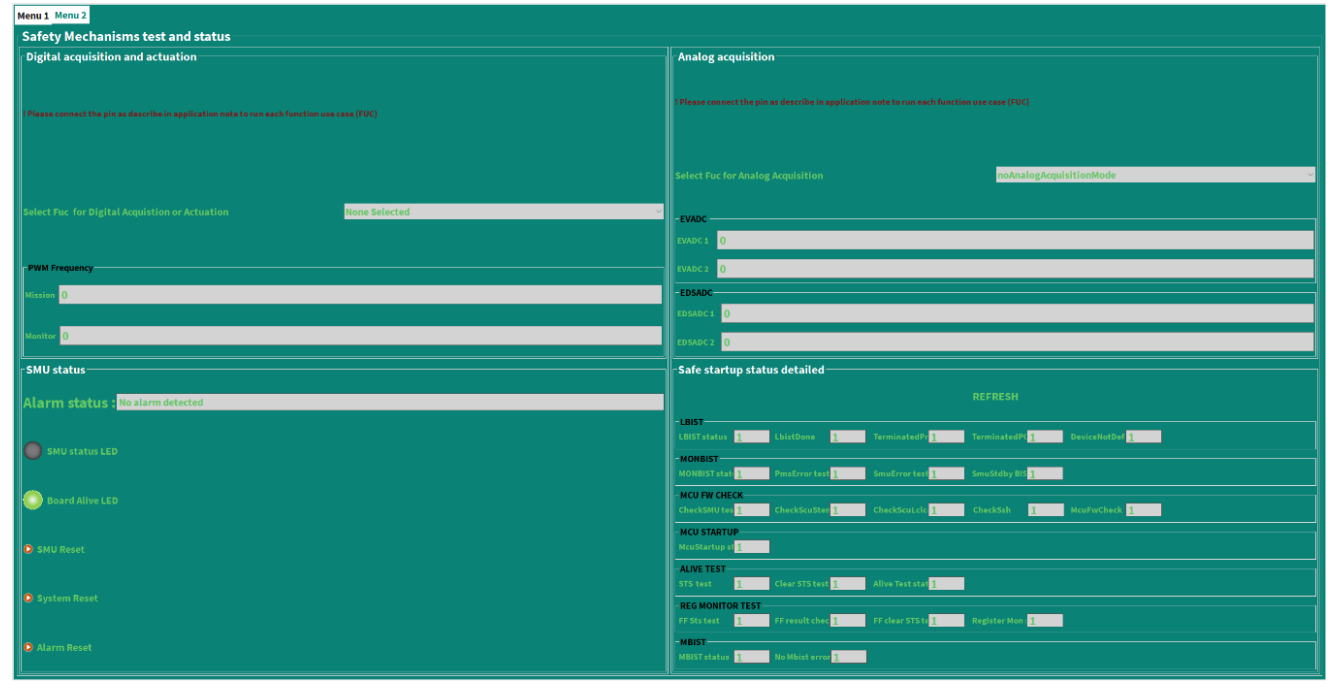

Figure 6 OneEye GUI-Menu 2

There are three color coding used for safety lite kit GUI, as shown in Figure 7.

- **Green color:** When the flag is in green, it indicates a successful test
- **Red color:** When the flag is in red, it indicates a failed test
- **Blue color:** When the flag is in blue, it indicates the test is not evaluated or performed

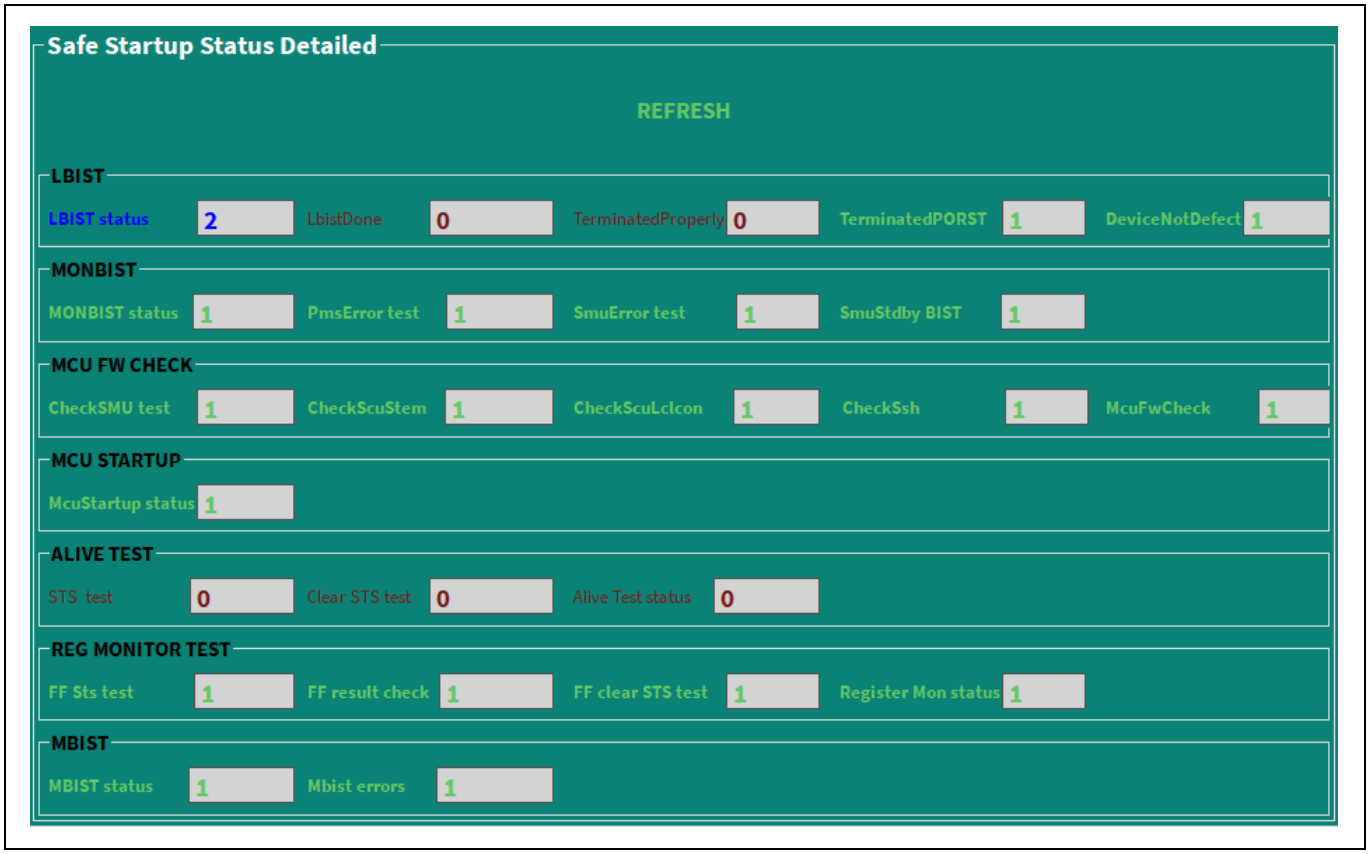

Figure 7 GUI color coding

3 Boot and startup procedure

The following section is about the boot and startup procedure which is needed to run a safe application.

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 stability of the 100 MHz backup clock source (fBACK). If the clock is stable, the PBIST (Safety Mechanism PBIST) is automatically executed. The device is released from the cold PORST reset only if the PBIST is executed successfully. The Power BIST (*SM:PBIST*) is implemented in the hardware to identify power-related faults, particularly those related to the voltage monitor mechanisms. This mechanism is automatically executed before PORST reset and only releases the AURIX™ device from reset after the test succeeds. The PBIST mechanism has no possible configuration from the user.

3.2 Boot firmware

After the PORST reset release, the system firmware (FW) execution is started by CPU0, while all other CPUs are held in a halt state. The FW is a code stored in *Boot ROM* that is automatically executed (complete or partial) by the device depending on the type of reset. The FW is configurable via UCB registers and is composed of the following parts (among others):

- Startup software
- Checker software
- Bootstrap loaders

3.2.1 Startup software (SSW)

The internal startup software (SSW) firmware contains procedures for device initialization. Depending on the type of reset of the device, the startup software will execute different tasks. Some parts may already be initialized when coming from a reset other than cold PORST. In general, the higher the reset level, the more extensive the initialization operations performed by the firmware. The start-up software is responsible for the following tasks:

- Flash ramp-up
- Device Configuration
- RAM Initialization
- Selection and execution of Startup Modes
- LBIST execution
- Lockstep configuration
- Ending the SSW 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].

3.2.2 Checker software (CHSW)

The checker software (CHSW) evaluates that all safety-critical aspects of the Start-up 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 the figure “AURIX™ TC3xx Platform Checker Software Overview” in the AURIX™ TC3xx User’s Manual [1].

3.2.3 Bootstrap loaders (BSL)

The routine of the bootstrap loaders (BSL) is to provide mechanisms to load a user program into the Program Scratchpad RAM of CPU0. This loaded code is started after exiting the *Boot ROM*. Once a Bootstrap Loader mode is entered, the selected communication protocol (CAN/ASC) must be completely executed until the user code is downloaded into the device. Therefore, after a successful execution of the firmware, the application software is started.

For more information about the bootstrap loader’s execution flow, see the Section “Bootstrap Loaders” in the AURIX™ TC3xx User’s Manual [1].

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. These operations ensure the absence of latent faults and correct initialization of the MCU before starting the runtime execution.

In particular, the application SW shall:

- 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 FW_CHECK
- Verify the correct configuration settings values installed by FW as described in Safety Mechanism 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 8 shows the sequence of safety mechanisms involved in the transition from an unpowered state to a full operational state.

Note: *In Figure 8 (bottom right), see the legends for the respective color information.*

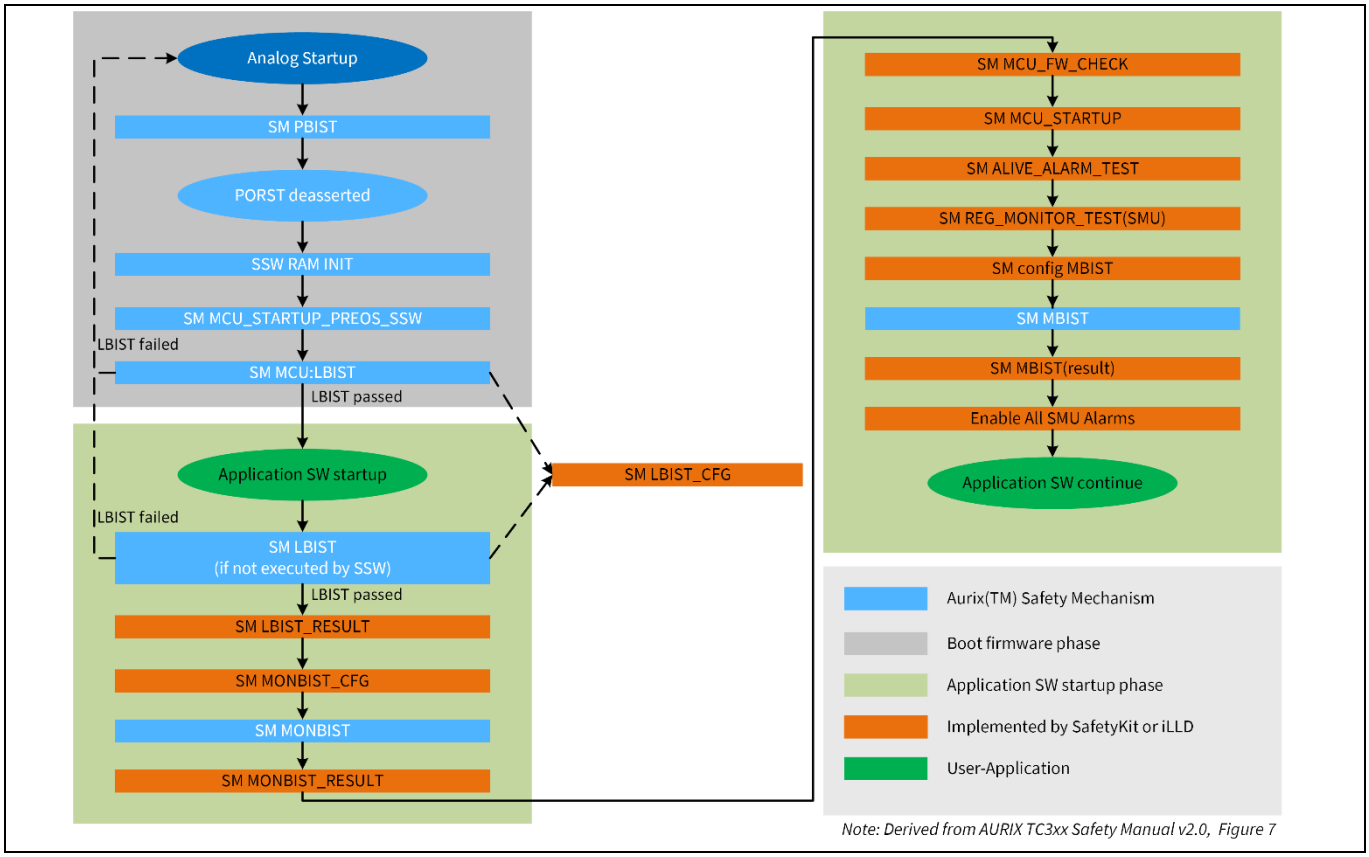

Figure 8 Safety mechanisms sequence during startup

Note: *Depending on the application, additional SMs at startup are needed. For example, if a non-Lockstep CPU is used for safety-relevant tasks, then all the SM regarding non-LS CPU must be considered.*

3.3.1 Implementation of application SW startup

The full application SW startup sequence is called in the `runAppSwStartup()` function, which is located in the */SafetyLiteKit/Sw_StartUp/Slk_Ssw.c* file. All the steps of the sequences can be activated or deactivated by the user with the macros located in the */SafetyLiteKit/Slk_Cfg.h* file. By default, all the sequences are activated. All files related to the application SW startup can be found in the */SafetyLiteKit/Sw_Startup* folder.

3.3.2 OneEye view of application SW startup

The status of complete SW startup can be found in OneEye both tabs, i.e. menu 1 as shown in Figure 9. A detailed status can be found in menu 2 as shown in Figure 10.

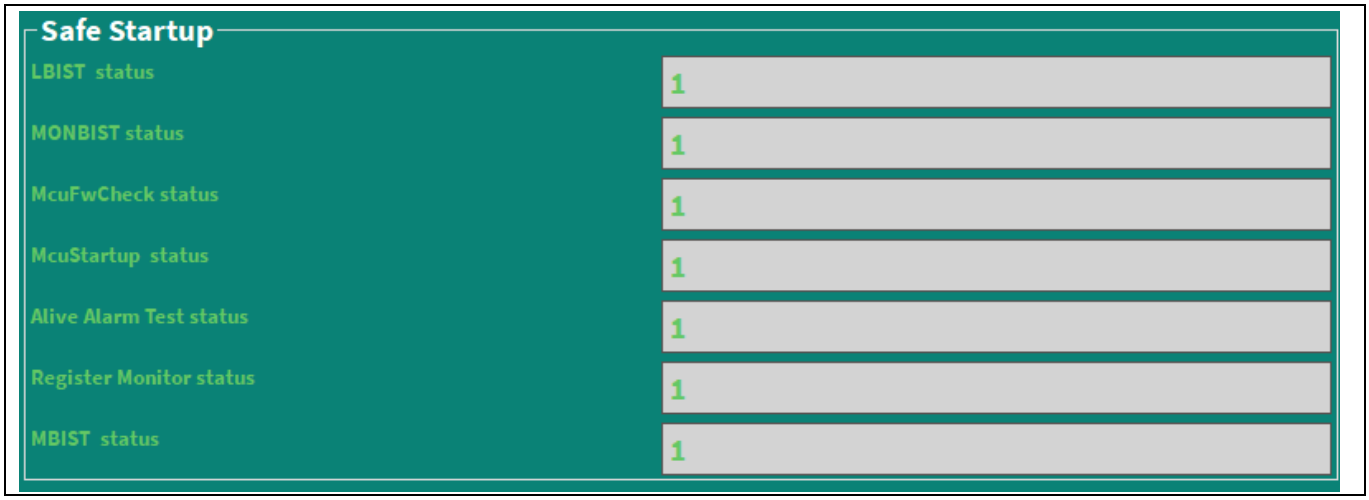

Figure 9 Application software startup (menu 1)

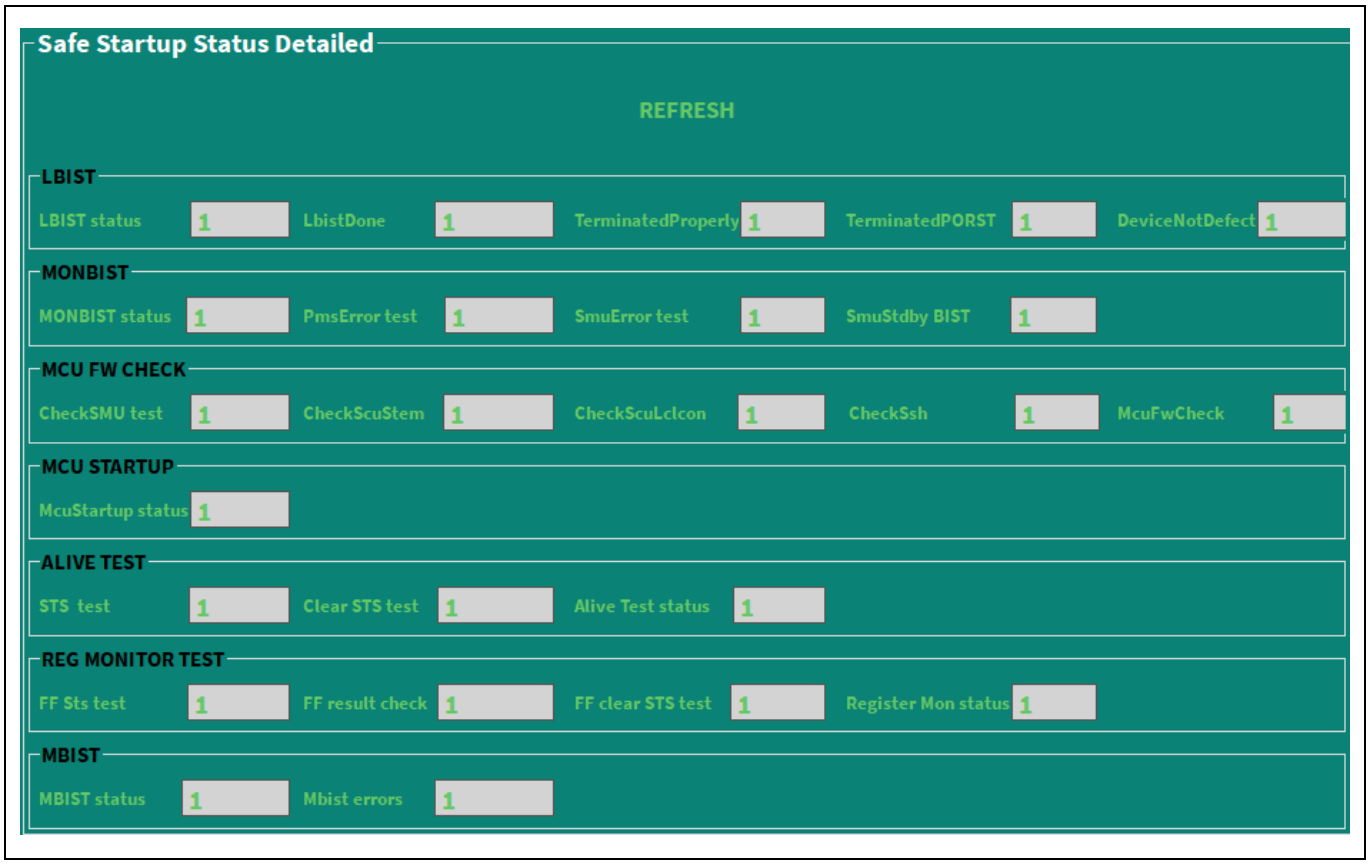

Figure 10 Application software startup (menu 2)

*Note: Regardless of the USB cable connection, the **'DAS connection status' play icon** in Figure 11 **must remain checked**.*

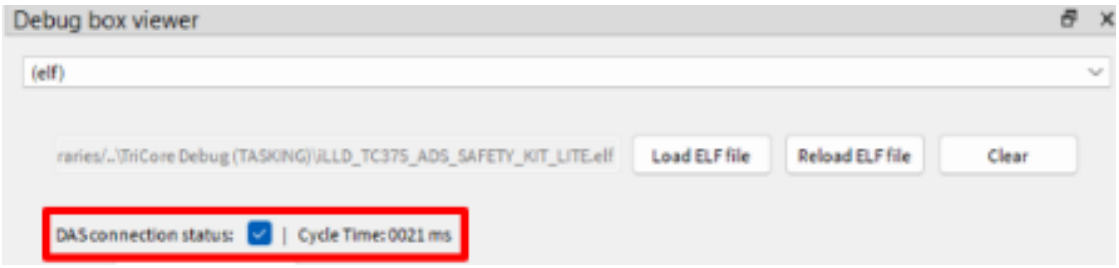

Figure 11 DAS button in OneEye interface

3.3.3 Reset triggering

In case, if any one of the tests shown in the previous sections does not pass successfully, the application must reset the microcontroller at least once. According to the AURIX™ TC3xx User Manual [1], Section “Reset Types”, there are four types 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

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 the 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 for cold PORST.

3.3.3.1 OneEye status flags after resets

Depending on the type of reset, the startup execution varies and these differences in startup behavior can result in different values of flag status. Specifically, the Logic built-in self-test (LBIST) and alive alarm test are only executed after a cold reset. These differences are detailed in Figure 12 and Figure 13.

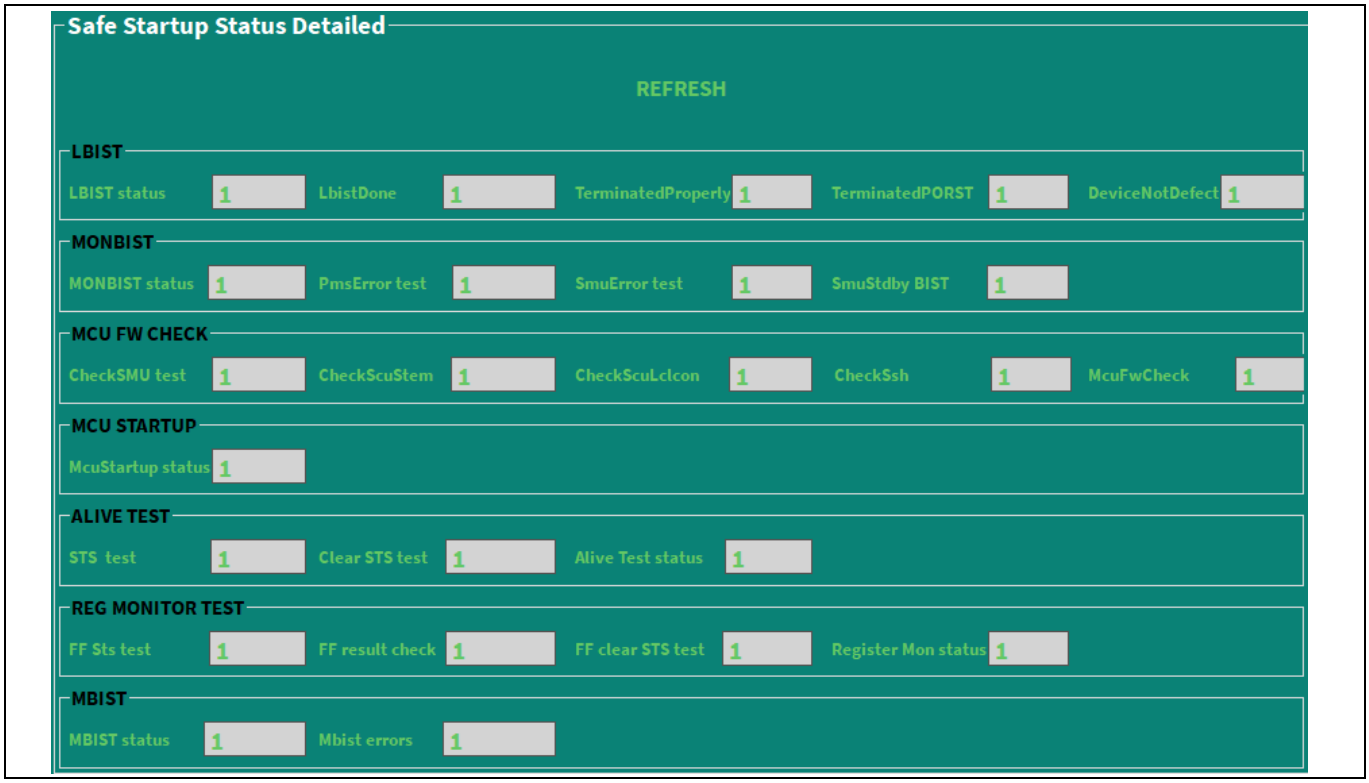

Figure 12 Status of startup software tags after a cold power on reset



Figure 13 Status of startup software tags after an application reset or a system reset or a warm reset

3.3.4 Logic built-in self-test (LBIST)

The LBIST module is responsible for testing the digital logic of the MCU. The Logic BIST executes the structural tests (the tests are using scan chains) and checks the digital logic of the MCU to detect hard errors. The LBIST is used as a safety mechanism (*SM:MCU:LBIST*) and contributes to achieve 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. This state can then be captured in a single clock cycle. The relation 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.

To detect any latent faults, 2^n possible combinations should be tested for a flip-flop chain of length 'n'. However, this would require an unreasonable time, considering the number of flip-flops. To address this issue, randomized patterns are used, and multiple patterns are 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 a test 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 AURIX™ User Manual Appendix.

As the complete circuitry of the chip is filled with a random pattern, a reset is required to restore 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 a cold power-on reset path.

All digital modules in the 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 LBIST. PMS modules, such as EVR, SCR WUT, and SMU_STDBY, are not covered by the LBIST. The PMS register interface is LBIST covered. Non-covered modules, such as Standby Controller Extension RAM (XRAM), can be used to store the number of LBIST executions in case of an error.

The LBIST covers all safety mechanisms, including SRAM, PFLASH ECC, MPUs, Bus SMs, and all SMU alarms, and the SMU_CORE. Therefore, there is no need to test the safety mechanisms again (for example by software) after a proper LBIST execution. The 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, the SRAM redundancy registers are part of the scan chain and hence corrupted. Therefore, SRAMs contents are not reliable after the LBIST and shall be initialized after the LBIST, prior to the usage. 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 LBIST should be monitored as stated by *Safety Mechanism MCU:LBIST_MONITOR*. Since the MCU's digital logic is not available during LBIST execution, an external device is responsible for ensuring the completion of LBIST execution. An external watch dog timer (WDT) can be used for this purpose, as highlighted in *Safety Mechanism SYS:WATCHDOG_FUNCTION*. After the PORST de-assertion, the external power supply waits for the 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 start-up sequence, the appropriate reaction shall be taken at system level.

As already mentioned in the Sections 3.2 and 3.3, the LBIST can be either executed by hardware during the Boot firmware or by application software during the application SW startup. Both ways are supported by the safety lite kit.

As stated in the safety mechanism, LBIST_CFG, LBIST can be executed with different configurations. The only recommended configuration that should be used to is the so-called “Configuration A”, which is given in the device-specific User’s Manual Appendix [3]. This configuration obtains the best coverage and a reasonable power consumption. In fact, with the wrong configuration, LBIST execution can drain so much current that the power monitoring will reset the device. This is not the case if configuration A is used. To avoid any kind of misconfiguration of the LBIST, the proper configuration can be found directly 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 MCU:LBIST_RESULT*).

3.3.4.1 Implementation

For the execution of LBIST during Boot firmware, a macro is activated in the */Configurations/Ifx_Cfg_SswBmhd.c* folder to change the value of the Boot Mode Index (BMI). For example, to activate LBIST, change the BMI from 0x007E to 0x017E and change the Cyclic Redundancy Check (CRC) calculation accordingly. The CRC is calculated on *bmhdid, bmi,* and *stad* (in this order). The source code with the LBIST code implementation can be found in */SafetyLiteKit/Sw_StartUp/Slk_Ssw_Lbist.c* file.

3.3.4.2 OneEye interface

All the LBIST flags are stored in the LBIST group. In the case of a cold-power-on-reset, LBIST is executed successfully. The LBIST flag reports if the test is successful. The *TerminatedPORST* flag reports if LBIST was terminated by a power-on-reset, and the *TermimatedProperly* flag reports if LBIST was terminated earlier or not and the *LbistDone* flag shows that LBIST is ready to be executed.

The safety lite kit LBIST flags can be found in both Figure 12 and Figure 13.

3.3.5 Monitor built-in self-test (MONBIST)

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

As mentioned in SMC[SW]:PMS:MONBIST_CFG in the AURIX™ TC3xx Safety Manual [2] and in the corresponding Section in the User Manual [1], the test procedure requires 16 steps. After the procedure has finished, the Application Software shall check the MONBIST results in MONBISTSTAT (Safety Mechanism PMS:MONBIST_RESULT). This procedure is fully covered by the Infineon Low Level Drivers (iLLDs), either as function as part of the Application SSW (*void* `Ifx_Ssw_Monbist(void)`) or as part of the SMU_stdby (*void* `IfxSmuStdby_startSmuStdbyMonBist(void)`).

3.3.5.1 Implementation

The source code with the MONBIST code implementation can be found in */SafetyLiteKit/Sw_StartUp/Slk_Ssw_Monbist.c.* The entire code is taken from the *iLLD* file */Libraries/Infra/Ssw/TC37A/Tricore/Ifx_Ssw_Infra.c* but with *iLLD* modification, for example, to only trigger MONBIST with the value *SLK_CFG_SSW_ENABLE_MONBIST.*

3.3.5.2 OneEye interface

All the MONBIST flags are stored in the MONBIST group. The MONBIST flag shows if the test is successful. The *TestOk* flag reports that the MONBIST check is completed. The SMU and PMS flags report an error during the execution. The safety lite kit MONBIST flags can be found in both Figure 12 and Figure 13.

3.3.6 Firmware check

The *Safety Mechanism MCU_FW_CHECK* is required for detecting failures, including both random hardware and systematic hardware/software, which may have affected the correct termination of the firmware.

In particular:

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

As highlighted in *Safety Mechanism SYS:MCU_FW_CHECK* in the AURIX™ TC3xx Safety Manual [2], the application software needs to verify that, after firmware execution, the relevant registers and SMU flags report the expected values.

Depending on the AURIX™ device and the kind of reset performed, the firmware execution varies. Therefore, the application software shall verify the correct content of registers and flags that are relevant for the specific type of reset. The complete list of registers and SMU alarms expected content can be found in the Appendix A of the AURIX™ TC3xx Safety Manual [2].

In case 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:** This can be caused by a specific UCB register combination, leading to an incorrect firmware sequence and MCU initialization
- **Hardware transient fault:** This can be caused by EM interference or other transitory events affecting some hardware part (for example CPU0) during the firmware execution
- **Hardware permanent fault:** This 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 second attempt also fails, then it should be concluded that the device is affected by a permanent hardware fault, and the device should be considered as not reliable.

3.3.6.1 Implementation

The implementation of Safety Mechanism SYS:MCU_FW_CHECK is complex and requires various steps. In this example, the main steps are as follows:

- 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. A full list of registers and their expected values can be found in Appendix A of the AURIX™ TC3xx Safety Manual **[2]**.*

In case all registers and SMU alarms report the expected values, the application SW shall:

- Clear the content of the registers mentioned in the tables in “Appendix A” of the AURIX™ TC3xx Safety Manual [2]
- 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. In this case, it is recommended to perform a second attempt by triggering a device reset (ideally of the same type).

SMU alarm check

During the firmware execution, several SMU alarms are activated. This is mainly due to access for memory initialization or SSH enabling. At the end of each reset, the application software is responsible for checking that the SMU_AG [0…11] alarm registers are reporting the expected flags. The official list of the expected values of the registers SMU_AG [0…11] are available in the AURIX™ TC3xx Safety Manual [2], Appendix A.

STMEM check

At the end of each reset, the CHSW checks the execution performed by the SSW and indicates the status of several modules in four specific registers:

- SCU_STMEM3
- SCU_STMEM4
- SCU_STMEM5
- SCU_STMEM6

For more details, read the section “Checks performed by CHSW and exit information” in the AURIX™ TC3xx User Manual. The expected values of these registers are available in the Section “Checker Software exit information for ALL CHECKS PASSED” of the device specific User’s Manual Appendix [3].

LCLCON check

During SSW execution, the lockstep cores are enabled according to the UCB_BMHD registers. After each reset (except an application reset), the application software shall 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. These values primarily depend on the reset type. The following registers are verified during the SSH register check:

- ECCD
- FAULTSTS
- ERRINFO [0]

The source code with the firmware check code implementation can be found in */SafetyLiteKit/Sw_StartUp/Slk_Ssw_Mcu_Fw_Check.c* file and all the tables used for comparison are stored in */SafetyLiteKit/Sw_StartUp/Slk_Ssw_Mcu_Fw_Check_Tables_TC375A.h* file. The status of this step in the startup sequence is considered passed only if SMU, STEM, LCLCON, and SSH registers are matching their expected values. If the status is not passed, the code called a reset, depending on the type of reset, and the registers are checked again.

If one of the checks shown in previous section did not meet, the aplication must reset the microcontroller at least once. There are four kinds of resets that can ve used in an application

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

Cold reset through LBIST

In case the external power supply is not able to generate a cold PORST by removing the supply voltage, it is still 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 the software. The LBIST controller will start and immediately stop the LBIST scan chain. At the end of the LBIST execution, the MCU internally triggers an automatic internal reset, which from a firmware execution point of view is equivalent to a cold PORST.

Before launching a new LBIST execution:

- The LBIST controller must be reset to its initial state and the “done” bit has to be set to zero again by the setting LBISTCTRL0.LBISTRES = 1
- Cold reset status bits must be cleared by setting LBISTCTRL2.CLRC = 1

At the next start-up, 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 shall 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 shall be considered faulty.

The function to called LBIST is `Slk_triggerLbist` in the */SafetyLiteKit/Sw_StartUp/Slk_Ssw_Lbist.c* file.

Warm PORST

A warm PORST is generated when the /PORST pin is asserted 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 PROCONRAM register and even to retrigger an LBIST execution, although this is not necessary. The standby domain remains functional during a warm PORST, ensuring that 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:

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

In the source code, it refers to the function `Slk_triggerSwReset` in the /SafetyLiteKit/Sw_StartUp/Slk_Ssw.c file.

`Note:` *In the OneEye GUI, the current reset code can be shown in the debug box viewer using the global function `g_SlkStatus/resetCode`. The reset code number is explained in the code section located at* the */SafetyLiteKit/Sw_StartUp/Slk_Ssw.h* file *.*

3.3.6.2 OneEye interface

All the FW CHECK flags are stored in the FWCHECK group. The MCUFWCHECK indicates whether the check was successful. The others check flags report the status of each check.

The safety lite kit Firmware check flags can be found in both Figure 12 and Figure 13.

Attention: *Debugger influence:*

Debuggers might have an (undefined) interference to the firmware behavior and alarms set during the 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 the expected values.

Attention: *“Early” execution:*

Although the firmware check (Safety Mechanism SYS:MCU_FW_CHECK) can be executed anytime during the application software start-up, it is recommended to perform its execution in an early stage of the start-up. 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 start-up timing requirements, the application shall be able to detect a fault in the firmware check and trigger a second reset (if required) as soon as possible.

3.3.7 MCU_STARTUP

Before entering the run mode, the application SW shall monitor the corruption of data stored in the safety-relevant registers.

To verify the correctness of these registers’ contents, 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.

3.3.7.1 Implementation

The source code with the MCU startup code implementation can be found in */SafetyLiteKit/Sw_StartUp/Slk_Ssw_Mcu_Startup.c*. The calculation of CRC32 is done by the TriCore™ function `__crc32()`.

3.3.7.2 OneEye interface

The MCU startup flag is stored in the MCUSTARTUP group, which indicates the outcome of the startup test by notifying whether the test is successful. The mismatch between the calculated and the expected CRC values are reported as an error. The safety lite kit MCU startup flag can be found in both Figure 12 and Figure 13.

3.3.8 SMU ALIVE_ALARM_TEST

In case of a malfunction, 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 will be generated if one of the following conditions are met:

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

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

For more information, see the Section “Interface to SMU_stdby” in the User’s Manual [1] and to the Section *Safety Mechanism SMU:ALIVE_ALARM_TEST* in the AURIX™ TC3xx Safety Manual [2].

3.3.8.1 Implementation

The source code with the SMU alive alarm test code implementation can be found in the */SafetyLiteKit/Sw_StartUp/Slk_Ssw_Alive_Alarm_Test.c* file. Only the reset type that resets the SMU is allowed to trigger the alive alarm test.

3.3.8.2 OneEye interface

All the SMU ALIVE_ALARM_TEST flags are stored in the ALIVE TEST group. The ALIVETEST flag shows if the test is successfull. The TestSts and TestClearSts flags reports an error if an alarm occurs during the execution.

The safety lite kit alive alarm test flags can be found in both Figure 12 and Figure 13.

3.3.9 SMU REG_MONITOR_TEST

According to *Safety Mechanism SMU:REG_MONITOR_TEST*, the application software shall 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 AURIX™ TC3xx Safety Manual. At the end of the test, the application software shall check the result by reading the RMEF register. Please refer to section *Safety Mechanism SMU:REG_MONITOR_TEST* in the AURIX™ TC3xx Safety Manual [2] for more information.

3.3.9.1 Implementation

The source code with the SMU reg monitor implementation can be found in the */SafetyLiteKit/Sw_StartUp/Slk_Ssw_Smu_Reg_Monitor_Test.c* file.

3.3.9.2 OneEye interface

All the SMU REG_MONITOR flags are stored in the REG MONITOR TEST group. The REGMONITOR flag shows if is the test is successful. All the flip flop reports errors during the execution. The safety lite kit SMU reg monitor flags can be found in both Figure 12 and Figure 13.

3.3.10 Memory built-in self-test (MBIST)

The Memory Built-in-Self-Test (MBIST) is a Memroy Test Unit (MTU) feature, which enables the verification of the internal SRAMs' integrity. As mentioned in the section *Safety Mechanism VMT:MBIST* in the AURIX™ TC3xx Safety Manual [2], the Memory Built-In Self-Test (MBIST) checks SRAM integrity. The SRAM areas are not covered by LBIST. In order to increase the latent fault detection coverage, this memory check is necessary.

The full Non-Destructive Test (NDT) described in the section "Non-Destructive Test (NDT)" of the AURIX™ TC3xx User's Manual [1] is fully implemented in the iLLD function `IfxMtu_runMbistAll(mbistGangConfig)`.

As stated in the Section "*Safety Mechanism VMT:MBIST"* of the AURIX™ TC3xx Safety Manual [2], the application software shall prevent any attempt of read or write operations on the memory during the execution.

Note: *During an MBIST, the memories being tested are not accessible. To prevent any unintended memory accesses, bus masters like CPU1…CPU2, etc., are kept disabled during the test. Since 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 be failed. Therefore, during this test, no RAMs of CPU0 are accessed because the function and struct etc. are stored in PFLASH and the local variable are handled by core registers. The number of local variables used should not exceed the core registers, as any exceed will be stored on the stack and cause the MBIST to fail (e.g., using also local vector or struct definitions). Therefore, it is the responsibility of user to take care of this constraint while performing MBIST.*

3.3.10.1 Implementation

The source code with the MBIST code implementation can be found in the */SafetyLiteKit/Sw_StartUp/Slk_Ssw_Mbist.c* file.

3.3.10.2 OneEye interface

All the MBIST flags are stored in the MBIST group. The MBIST flag shows if the test is successful. The MbistErrors reports errors during the execution. The safety lite kit MBIST flags can be found in both Figure 12 and Figure 13.

3.3.11 Enable all SMU alarms

The AURIX™ TC3xx Safety Manual [2] recommends enabling all SMU alarms relevant for the application. In particular, the user shall re-enable ALM8[0] and ALM8[3] that shall be disabled during the oscillator and PLL configuration procedure. For more information, see the function *void* `SlkEnableAllSMUAlarms(void)` in the file */SafetyLiteKit/Smu/SMU.c*.

4 Safety management unit (SMU)

The cornerstone of AURIX™ MCU 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 the different hardware and software-based safety mechanisms. The purpose of the SMU is to configure the behavior of each alarm. These alarms can be individually configured to trigger internal actions and/or notify externally the presence of faults through a fault signaling protocol (FSP). The severity of each alarm shall be configured according to the needs of the safety application(s). By default, every alarm reaction is disabled, apart from the watchdog time-out alarms.

The SMU is split in two parts (see Figure 14 to understand the SMU simplified version):

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

Note: For more information, see the AURIX™ TC3xx Safety Manual [2], Chapter 6 “Safety Mechanisms”, and AURIX™ TC3xx User’s Manual [1], Chapter “Safety Management Unit (SMU)”.

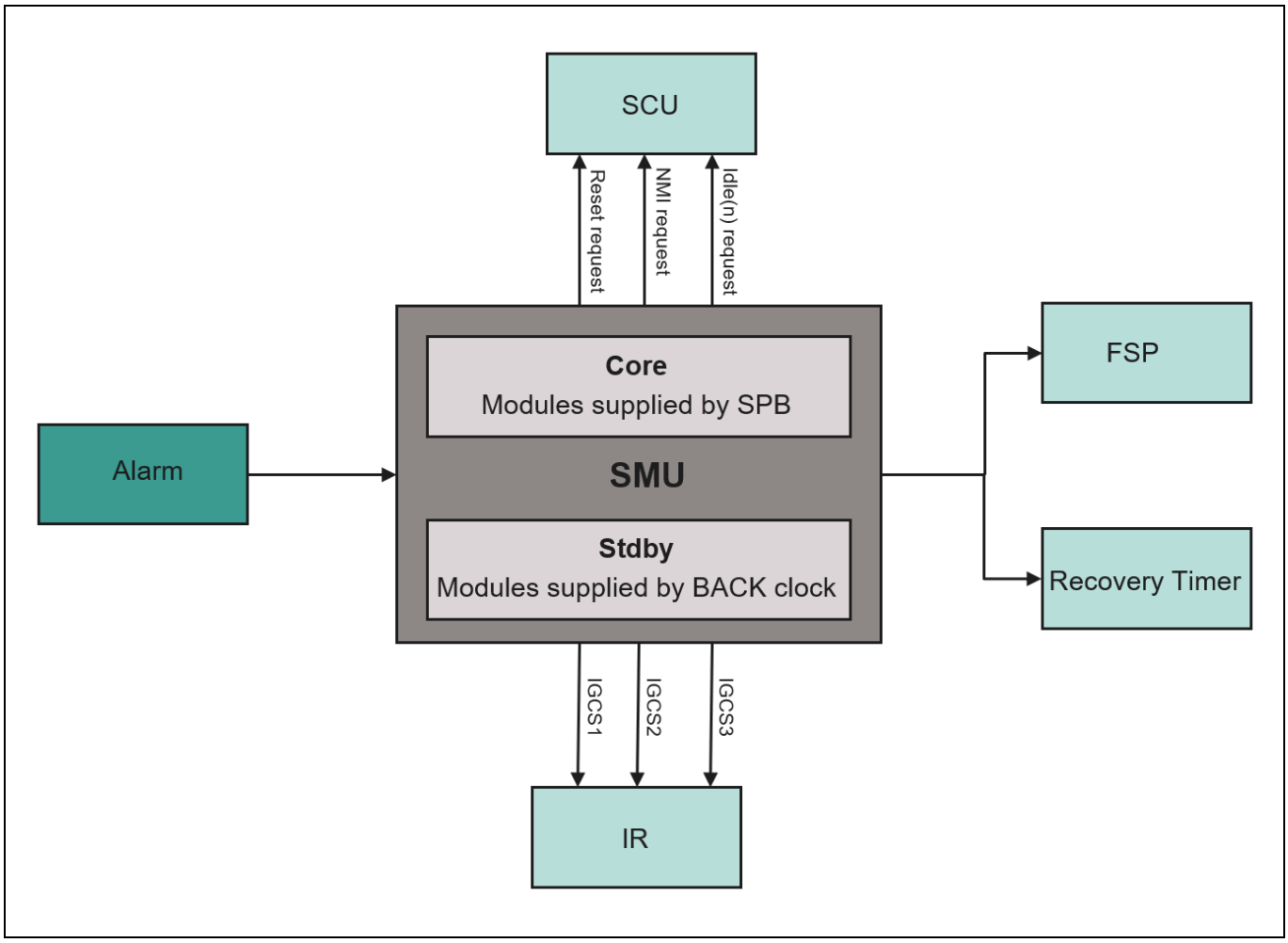

Figure 14 Simplified SMU concept

When the safety mechanism detects a failure, an SMU alarm event is generated. All inputs alarms are mapped to the alarm configuration registers. Each alarm group consists of up to 32 input alarms. By default, all alarm reactions are disabled, except for the watchdog timeout and recovery timer alarms. It is responsibility of the system integrator to determine the kind of action which need to be activated in response to a failure. For each alarm in the SMU, one of the following internal actions can be chosen:

- NO ACTION
  - The alarm handling is disabled, and no action is taken. This is the default value for all alarms
- INTERRUPT REQUEST
  - Generate an interrupt request to any of the CPUs, concurrent interrupts to several CPUs can be configured
  - The AGC.IGCSx register can controlled how the SMU triggers interrupt request to the interrupt router
- NMI TRAP REQUEST
  - An NMI request is forwarded to the SCU, which determines to which CPU issue the trap.
- RESET REQUEST
  - Depending on the configuration selected, an application or system reset request is forwarded to the SCU
- CPU RESET REQUEST
  - Triggers a reset to one or more CPUs

In parallel to internal actions, each alarm can trigger one or both of the following external actions:

- RECOVERY TIMER
  - The safety management unit (SMU) uses two recovery timers (RT0 and RT1) to monitor response times of resets, NMIs, or alarms triggered by interrupts. If a recovery timer is not serviced before it times out, an alarm is generated. Both instances are enabled by default, and they are required for CPU watchdog operation
- FAULT SIGNALING PROTOCOL
  - 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

4.1 Implementation

In safety lite kit, the SMU driver and SMU application software alarm codes are implemented.

4.1.1 SMU driver

The function that implements the SMC[SW]:SMU:CONFIG is `initSMUModule`, which is explained and shown in the */SafetyLiteKit/Smu/SMU/SMU.c* file. This function contains a test of the safety mechanism *SM:SMU:LOCK* to ensure proper SMU functionality. Afterwards, the structures are filled according to the *globalAlarmConfig* structure of each alarm. The alarm mapping is documented in the AURIX™ TC37x User's Manual Appendix [3].

4.1.2 SMU application software alarm

The function that implements the Safety Mechanism SMU:APPLICATION_SW_ALARM is `softwareCoreAlarmTriggerSMU` function which can be found in */SafetyLiteKit/Smu/SMU.c* file. This function can only trigger software alarms according to the alarm mapping, which can be found in the AURIX™ TC37x User's Manual Appendix [3].

4.2 OneEye interface

In safety lite kit OneEye interface, SMU driver and SMU application software alarm flags, inputs/outputs are shown in the SMU section of the menu 1 and menu-2 tab pages. The OneEye SMU interface is refreshed by calling the function `runtimeSmuOneEye`, which can be found in the *SafetyLiteKit/OneEye/OneEye_SMU.c* file.

The initial value of the software alarm number is set to the 255, as shown in Figure 15. This value, being much higher than a software alarm value, does not trigger any alarms. To trigger the software, it is required to enter a value of 15 or less in the line edit. For example, the software alarm number 10 can be triggered, as shown in Figure 16.

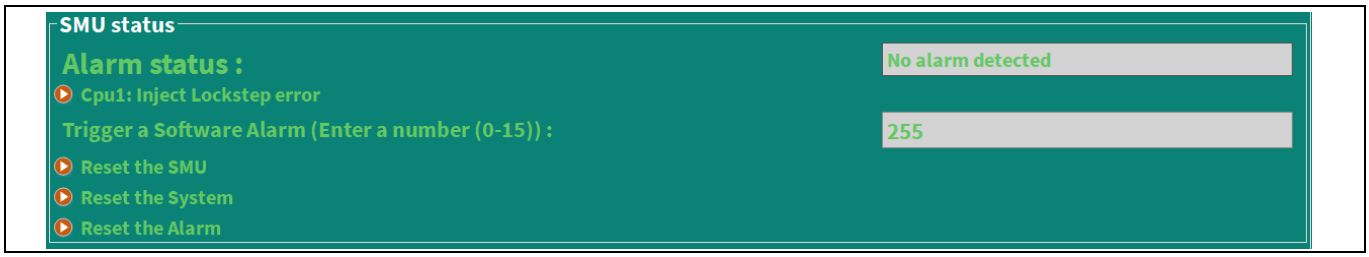

Figure 15 SMU section without any alarm detected

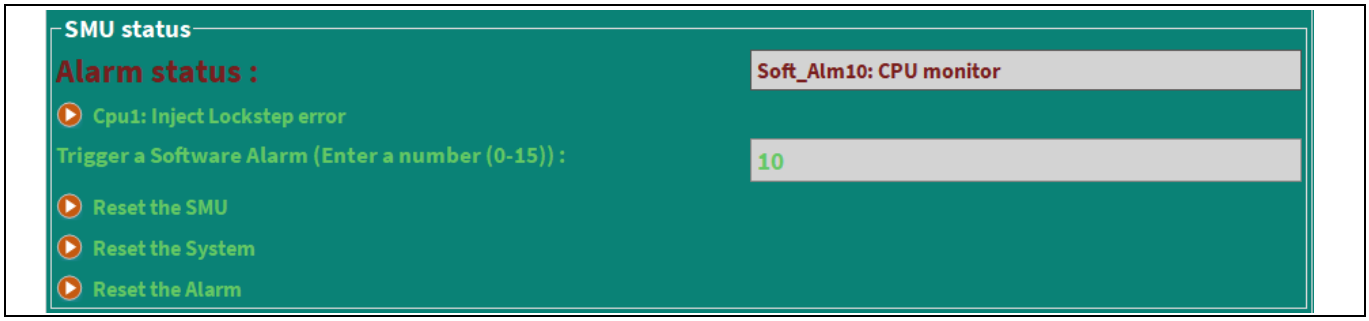

Figure 16 SMU section with the software alarm 10 triggered

Moreover, in this interface, it is possible to apply different resets, as shown in Figure 15 and Figure 16.

- **Reset the SMU**
  - This play icon calls a function that resets all SMU alarms
- **Reset the System**
  - This play icon calls a function to execute a system reset
- **Reset the Alarm**
  - This play icon calls a function to clear the latest activated alarm

4.2.1 OneEye SMU script file

The alarm status in the SMU status group in Figure 16 is based on the `IfxSmu_Alarm` enumeration which can be found in *Libraries/iLLD/TC37A/Tricore/_Impl/IfxSmu_cfg.h* file. In the OneEye GUI, the print functionality is done by using JavaScript. Therefore, a JavaScript file named `outputPrintSMU.js` is created, which can be found in *Libraries* folder in the project. This file contains the list of alarm naming corresponding to `IfxSmu_Alarm` enumeration and manipulates the naming of software alarm according to the dedicated flags

for that software alarm (if needed). The file is updated whenever there is change in the corresponding global variable, i.e. `g_returnSMU`.

4.2.2 Status of SMU driver

The status of the SMU status can be checked in the `initSMUModule` function, which can be found in */SafetyLiteKit/Smu/SMU.c* file. All the flags in the global variable `g_SlkStatus.smuStatus` (located in *SafetyLiteKit/Slk_Main.c*) must be set, except for two flags that are set in alive alarm test during software startup only under cold PORST, as shown in Figure 17.



Figure 17 SMU flags status

5 Safety related function

5.1 Register monitor test

There are many 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 involve a verification of the address logic. For the complete list of the internal hardware SMs, see the AURIX™ TC3xx Safety Manual [2].

The application software is responsible for running and checking the BIST for latent fault detection in the SRAM SFF once per driving cycle using the 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. Refer to the 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 in case of an error. The application software is responsible for clearing the affected registers after running the following test:

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

5.1.1 Implementation

The REG_MONITOR test is executed right after the Application SW startup (see Section 3.3). The implementation of this test can be found in */SafetyLiteKit/Safety_Mechanisms/Slk_RegMon.c*. The test consists of the following steps,

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

5.1.2 OneEye interface

Once the register monitor test for each functional block (FB) is done, the resultant test status, whether the test is passed or failed, is printed in OneEye GUI in menu 2 at the safe status group. In Figure 18, the highlighted red status bit is the test result status of all the register monitors.

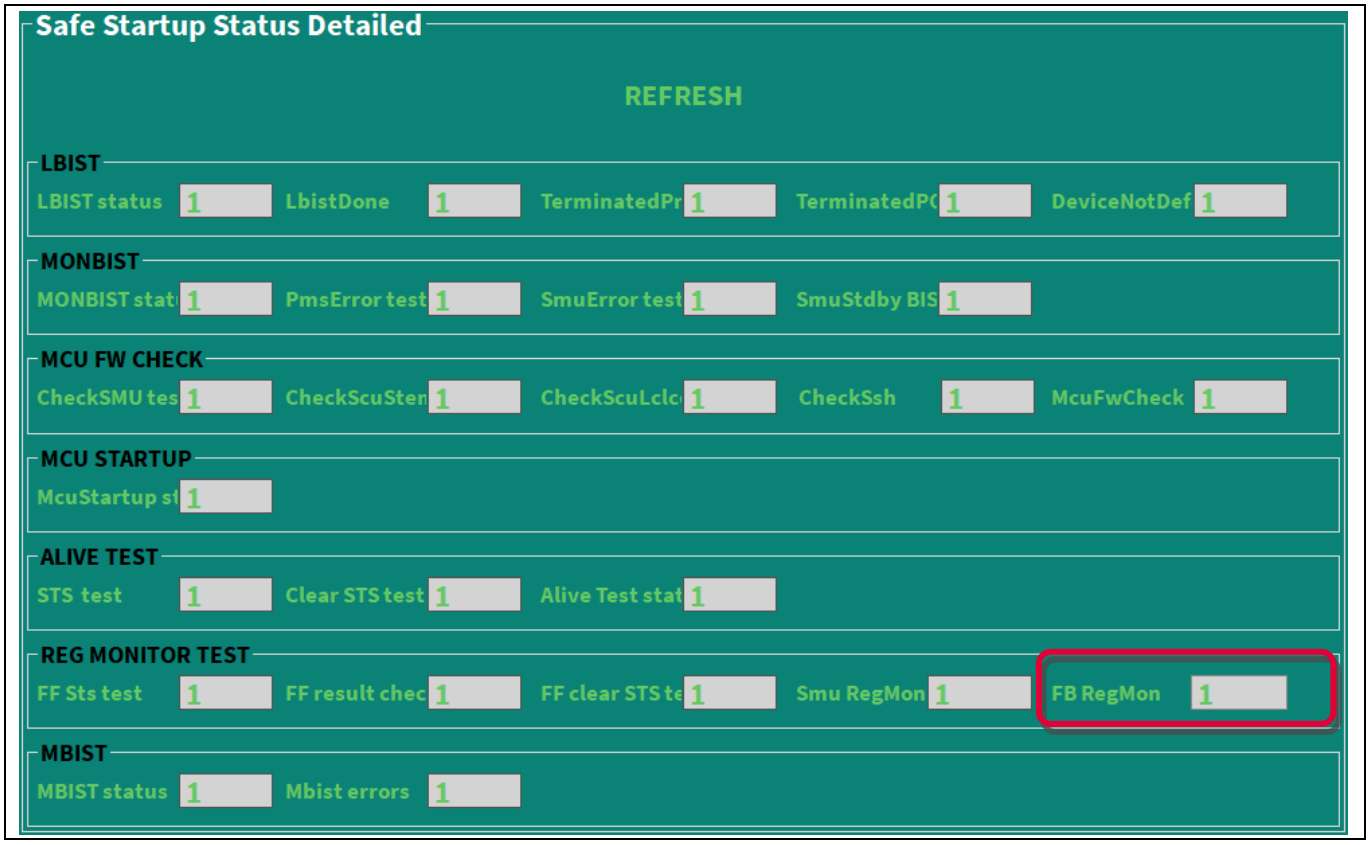

Figure 18 Register monitor test result

5.2 Safe computation NLS-CPU

5.2.1 CPU safe computation NLS-CPU concept

For CPU safe computation on non-lockstep (NLS) CPU, it is required to implement safety mechanisms in the non-lockstep core of MCU. For AURIX™ TC375, the only non-lockstep core is the CPU2(Figure 19).

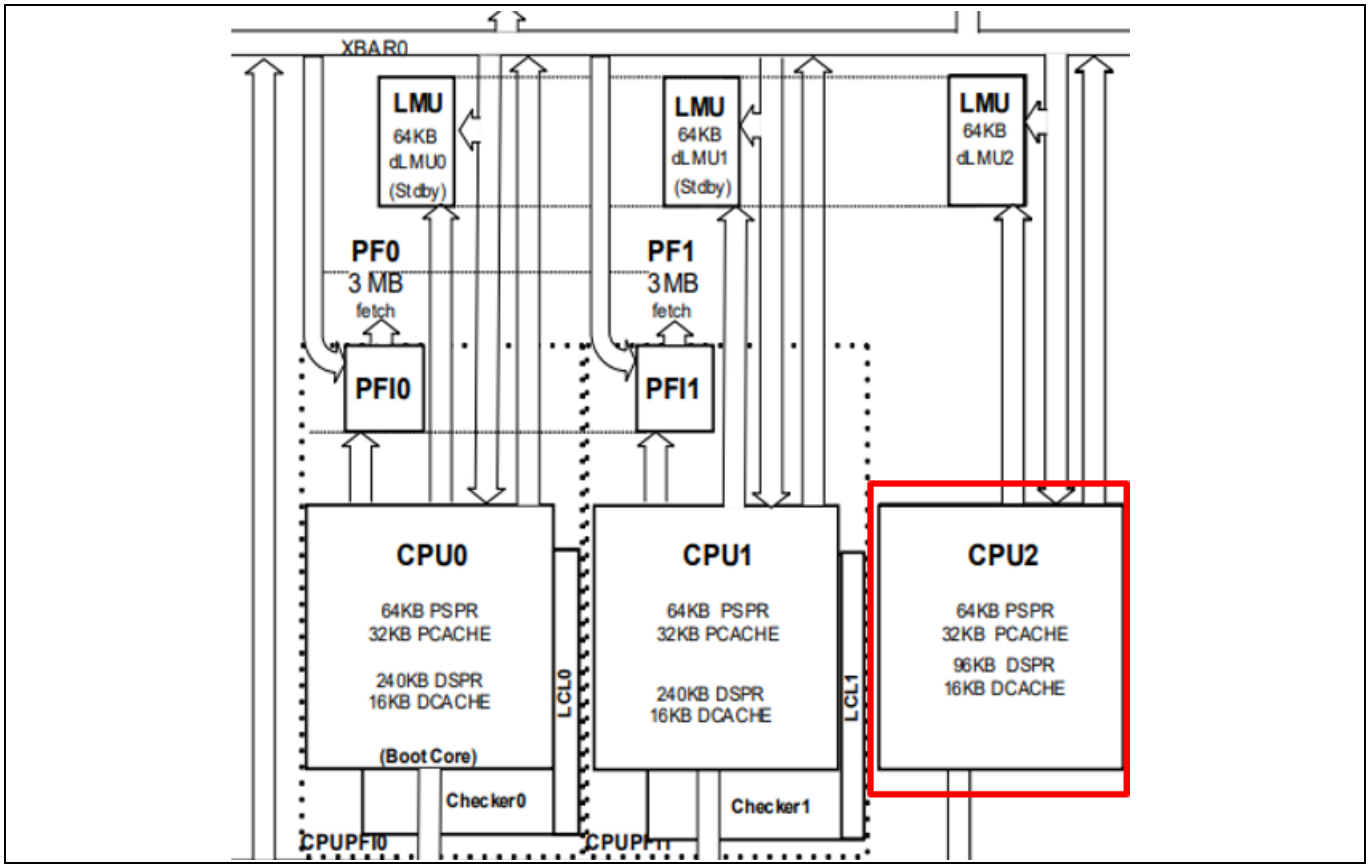

Figure 19 TC37x non-lockstep core

Figure 19 is an extract from AURIX™ TC3xx User's Manual part 1 [1], showing the block diagram of the AURIX™ TC37x. All safety mechanisms related to CPU safe computation are executed in the second core.
Safety Mechanisms implemented are:

- Safety Mechanism CPU:AP_CHECK
- Safety Mechanism CPU:BUS_MPU_INITCHECK
- Safety Mechanism CPU:CODE_MPU_CHECK
- Safety Mechanism CPU:DATA_MPU_CHECK
- Safety Mechanism CPU:INTERNAL_BUS_MONITOR
- Safety Mechanism CPU:SBST
- Safety Mechanism CPU:SFR_TEST
- Safety Mechanism CPU:SOFTERR_MONITOR
- Safety Mechanism SYS:SW_SUPERVISION

To fill the assumption of safe computation on NLS-CPU, the safety manual recommends implementing the following functional block with their safety mechanisms needed, as listed in Table 4.

Table 4 Safe computation requirement

Functional Block	SM needed	Notes
NLS-CPU	Safety Mechanism CPU.*:*	Refers to 4.2.1
NVM	Safety Mechanism NVM.*:*	–
LMU	Safety Mechanism LMU.RAM.*:*	No LMU for TC37x
EMEM	Safety Mechanism EMEM.*:*	No EMEM for TC37x
VMT	Safety Mechanism VMT.*:*	Refers to 3.3.10
FCE	Safety Mechanism FCE.*:*	–
DMA	Safety Mechanism DMA.*:*	–
SRI	Safety Mechanism SRI.*:*	–
FPI	N.A	–

For the implementation of safety mechanisms from functional blocks other than NLS-CPU, see Section 5.3.

5.2.2 Access protection configuration SMs

A part of the CPU's safe computation safety mechanisms requires the use of the register's readback mechanism.

5.2.2.1 Implementation

These safety mechanisms and their respective function (in */SafetyLiteKit/Safe_Computation/Cpu/Slk.c*) are:

- Safety Mechanism CPU:AP_CHECK, `slkCpuAccessProtection()`
- Safety Mechanism CPU:BUS_MPU_INITCHECK, `slkCpuBusInitCheck()`
- Safety Mechanism CPU:CODE_MPU_CHECK, `slkCpuCodeMpuCheck()`
- Safety Mechanism CPU:DATA_MPU_CHECK, `slkCpuDataMpuCheck()`
- Safety Mechanism CPU:SFR_TEST, `slkCpuSfrTest()`

All these functions are organized in the identical way. First, the registers associated with the corresponding safety mechanism are configured, and then a readback function checks if registers are set as expected. While configuring a register, it is important to configure with safety `ENDINIT` protection and to use a macro to assign a value to the register. A macro is only a text replacement, and it must not be used as a constant variable. However, it is also possible to store registers values into constants variables, but only in the CPU's non-LS memories. For the readback function (`slkReadbackCpuRegister()` located in /SafetyLiteKit/Safe_Computation/Cpu/Slk_Cpu.c), the register values are compared to their expected one. If it is not matching, a software alarm is triggered by the SMU, which is running on CPU0. For more information about SMU and its implementation, see Section 4. In this selection of SMs, the only one that needs a latent fault detection is Safety Mechanism CPU:SFR_TEST. This means that one time per FFTI, the `slkCpuSfrTest()` function is called. This is executed in the `runSlkStmIsr()` function (/SafetyLiteKit/Slk_Main.c) in the CPU2 section. The FFTI value is defined for all the application as 10 ms, but it can be changed in the /SafetyLiteKit/Slk_Cfg.h file.

5.2.2.2 OneEye interface

In safety lite kit OneEye interface, CPU safe computation on NLS-CPU register's readback mechanism status is shown in the menu 1 of the interface (see Figure 20).

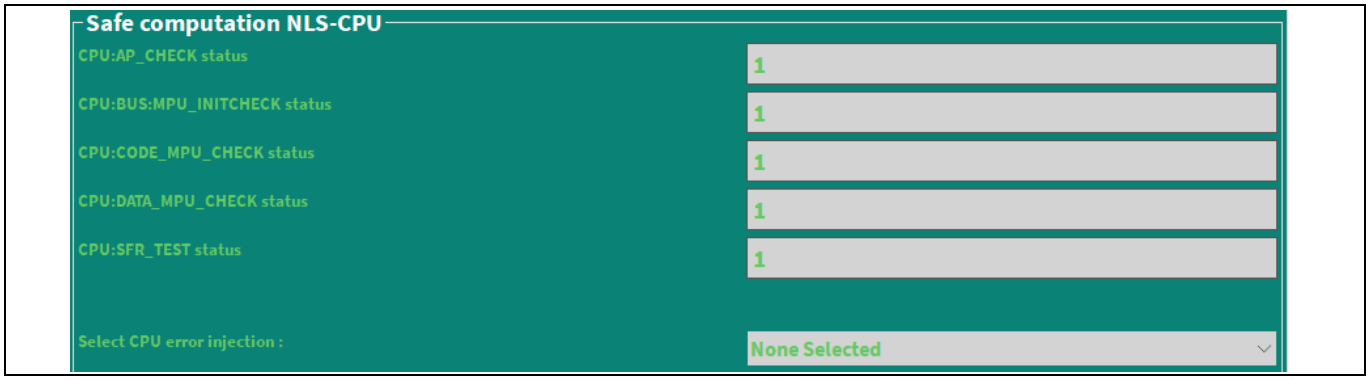

Figure 20 NLS-CPU safe computation SMs status

For error injections, the user can select error injection to any of the available SM by using the drop-down menu shown in Figure 21. The error will be injected accordingly, and the corresponding alarm message will be printed in SMU status. It is important to note that SMs that do not require a latent fault detection should not be called another time. To inject the error, the device should be reset with the error and then called only once.

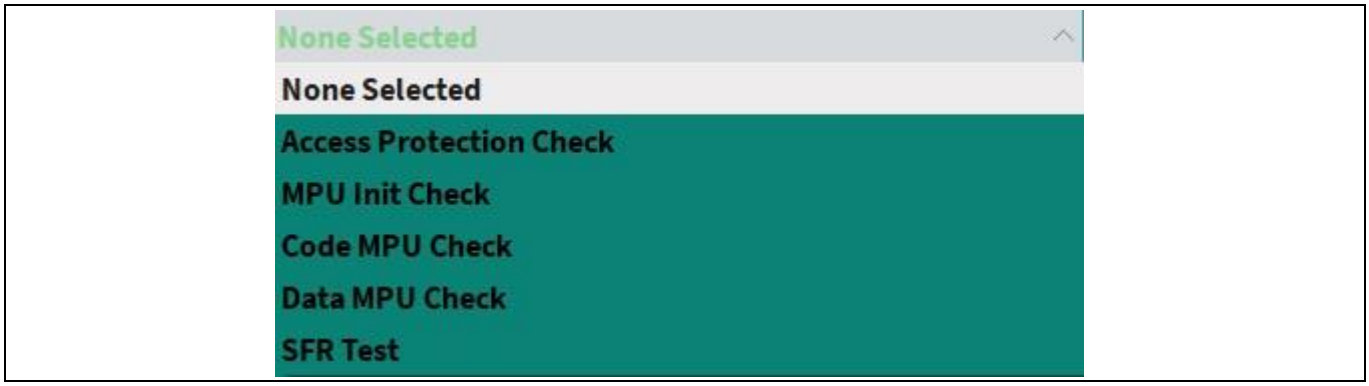

Figure 21 CPU error injection drop-down menu selection

5.2.3 Software based self-test (SBST)

The CPU safe computation on NLS-CPU also requires non-lockstep cores to implement the CPU SBST into the application. The SBST features are:

- Software pattern to ensure the functional integrity of a non-lockstep TriCore™ CPU core
- Detects 90% of permanent random hardware faults in the fetch unit and pipelines
- The CPU SBST shall be called for every diagnostic time interval
- Covers the AURIX™ 2nd Generation Safety Manual requirement from SM:CPU:SBST
- Required for applications with low ASIL level requirement on AURIX™ TC3xx non-lockstep CPUs

5.2.3.1 Implementation

As this project is only for demo purpose, only the dummy SBST files are provided with the safety lite kit, and the routine is explained in */SafetyLiteKit/Safe_Computation/Cpu/Slk_Cpu.c* within `initSBST()` and `runSBST()` functions.

Here are two approaches to executing the SBST test during each FTTI. The first approach involves executing the SBST by slices. Initially, the CRC seeding value is set with the `SBST_CPU_Seeding_CRC_Value()` function. Next, the first slice function is executed (`SBST_CPU_CPU_Slice()`, $i \in [0;8]$). In the next FTTI occurrence, the next slice (i+1) is executed, and this process continues until the last slice is executed. This mechanism is repeated in loop to execute one slice per FTTI occurrence. The SBST test is considered as complete when the 8th slice is executed, and the CRC result check is done for successful execution.

Another way to implement the SBST test is to execute the full test of each FTTI via the `SBST_CPU_run_test()` function and check the CRC results accordingly. One of these SBST executions can be selected from the OneEye menu 1.

5.2.3.2 OneEye interface

By default, when the full SBST test is running, the user can still select slice SBST test using the drop-down menu shown in the upper part of the Figure 22.

For the SBST, the error play icon activation calls the `SBST_CPU_Seeding_CRC_Value_Fault_Inj()` function used in *SafetyLiteKit/CPU_SBST/CPU_SBST.o* module. In this function, a wrong seeding value is injected, and the corresponding CRC value of the SBST_CPU_RESULT will be a wrong value, which triggers a dedicated software alarm, as shown in the lower part of Figure 22.

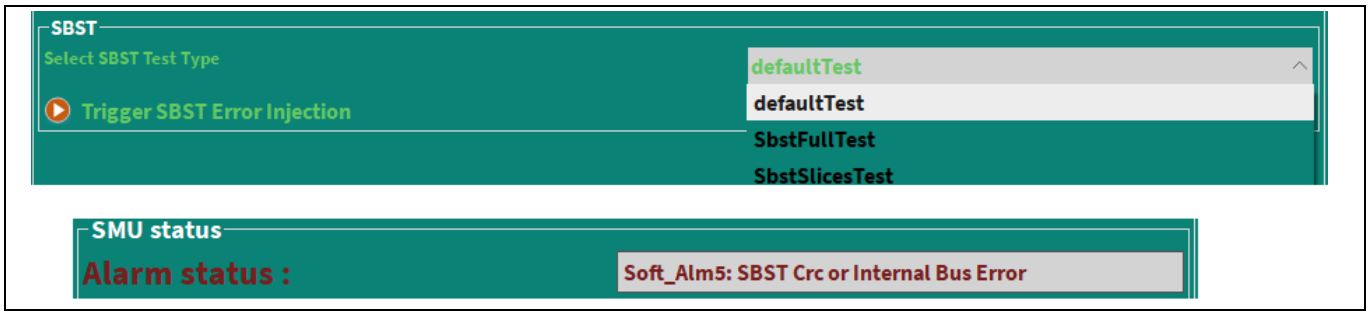

Figure 22 SBST safe computation

5.2.4 Internal bus SMs

5.2.4.1 Implementation

During each FTTI, after the CPU software-based self-test, the safety mechanism CPU:INTERNAL_BUS_MONITOR is implemented. This safety mechanism checks the internal FIFO, bus interface, and the interface to the DLMU. The pseudo-code is given in the “Appendix B” of the AURIX™ TC3xx Safety Manual [2]. This pseudo-code is adapted according to TASKING compiler.

5.2.4.2 OneEye interface

There is no widget defined for this SM in the OneEye GUI.

5.2.5 Monitor and supervision SMs

5.2.5.1 Implementation

The CPU:SOFTERR_MONITOR safety mechanism can be implemented by using the Safety Mechanism SYS:SW_SUPERVISION. The integration of a CPU watchdog is done and the CPUxWD is configured to be

reloaded every 8 ms and serviced during each FTTI at every 10 ms. This mechanism can be found in Cpu2_Main.c.
The timeout for the CPU watchdog is calculated using the following equation:

$$timeout \approx \frac{0xFFFF - REL}{\frac{100MHz}{16384}}$$

Equation 1 Timeout for CPU watchdog

The CPU frequency is defined in the “CPU Clock Control” section of the AURIX™ TC3xx user’s manual [1]. The $CPU2DIV$ register is disable; therefore, the CPU frequency is equal to the SRI frequency.

$$f_{CPU} = f_{SRI} = 100\ MHz$$

Equation 2 CPU clock frequency

The CPU watchdog timeout calculation is described in “Timer Operation” section of the AURIX™ TC3xx user manual [1]. For a maximum timeout of 8 ms, the REL register is set to 0xFFCE. For more information, see the Watchdog_1 for KIT_AURIX_TC375_LK training videos [4], available in AURIX™ TC3xx Expert Trainings.

The other possibility to implement this safety mechanism is to configure a trap handling. The trap handling can be used within the debugger and can be tested with the CPU2 each FTTI in the debugger mode. The global variable structure `g_provokeTraps` located in */SafetyLiteKit/Safe_Computation/Cpu/CPU_Trap_Recognition.c* file, can trigger different type of traps. The link between these global variables and traps is described in Table 5. More information on traps generation is explained in */SafetyLiteKit/Safe_Computation/Cpu/CPU_Trap_recognition.c* and trap handling is explained in */SafetyLiteKit/Safe_Computation/Cpu/TrapHandler.c*. It is important to note that the IFX_CFG_EXTEND_TRAP_HOOKS variable in `Configurations/Ifx_Cfg.h` is required to be uncommented to enable traps handling.

It is important to note that traps and their safe reaction can be triggered directly from the OneEye GUI. But in a safety application, a system reset needs to be performed at the end of the trap handling.

Table 5 Traps provocation for traps handling

Traps	Class	Name	Synch. /Asynch.	HW/SW	Definition
g_provokeSynchronousHardwareTrap	4	DSE	Synch.	HW	Data Access Synchronous Error
g_provokeAsynchronousHardwareTrap	4	DAE	Asynch.	HW	Data Access Asynchronous Error
g_provokeSynchronousSoftwareTrap	5	OVF	Synch.	SW	Arithmetic Overflow
g_provokeProtectionTrap	1	MPN	Synch.	HW	Memory Protection Null Address

Traps	Class	Name	Synch. /Asynch.	HW/SW	Definition
g_provokeNmiTrap	7	NMI	Asynch.	HW	Non-Maskable Interrupt

5.2.5.2 OneEye interface

For the trap handling, a specific one can be selected in the dedicated drop-down menu which led to trigger a software alarm and inform to trigger a system reset after it to ensure a safe state. The drop-down menu for Trap error injection is shown in Figure 23.



Figure 23 Trap error injection drop-down menu selection

5.3 Safe computation LS-CPU

5.3.1 Safe computation LS-CPU concept

The safety-related function "Safe computation LS-CPU" describes the common use of the MCU for the execution of SW and internal data transfers in safety-related applications.

To implement the safe computation on LS-CPU, the application SW shall implement the following SMs listed in Table 6.

Table 6 Safe computation LS-CPU requirement

Functional block	SM needed	Notes
LS-CPU	Safety Mechanism CPU.*:*	Refers to Section 5
NVM	Safety Mechanism NVM.*:*	–
LMU	Safety Mechanism LMU.RAM.*:*	No LMU for TC37x
AMU	Safety Mechanism AMU.*:*	–
EMEM	Safety Mechanism EMEM.*:*	No EMEM for TC37x
STM	Safety Mechanism STM.*:*	–
VMT	Safety Mechanism VMT.*:*	Refers to Section 3.3.10
FCE	Safety Mechanism FCE.*:*	–
SCU	N.A	–
DMA	Safety Mechanism DMA.*:*	–

Functional block	SM needed	Notes
SRI	Safety Mechanism SRI.*:*	–
FPI	N.A	–

5.3.2 Program flash - 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. Their compute performance is optimized by using a point-to-point interface to minimize latency and maximize 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 are the structure term used for the PFLASH module:

- **Flash Module**: Contains a Non-Volatile Memory (NVM) with its own digital logic and in the PFLASH there are one more PFp banks
- **Bank**: Support concurrent operation with some limitation. One bank equal to three Physical sector 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 1M Bytes
- **Logical Sector**: One logical sector is equal to 16k Bytes, and each logical sector is equal to a 32-word line. It is the smallest unit to erase PFLASH
- **Word line**: One word line is equal to 512 Bytes, which is equivalent to 16 pages
- **Page**: It is smallest unit that can be programed. One page is equal to 32 Bytes

The detailed description, features, and multiple sector partitions can be found in the AURIX™ TC3xx User's Manual [1].

5.3.2.1 Implementation

The list of PFLASH NVM SMs implemented safe computation on lockstep CPU are:

- Safety Mechanism NVM.PFLASH:INTEGRITY_CHECK
- Safety Mechanism NVM.PFLASH:UPDATE_CHECK
- Safety Mechanism NVM.PFLASH:WL_FAIL_DETECT

SM:NVM.PFLAHS:INTEGRITY_CHECK

For the implementation of this SM, a dummy safety data is stored at PF0 in an array at 0xA0100000 address. 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. For more information about the FCE, see Section 5.3.3. The data stored at PF0 can be found in the */SafetyLiteKit/Safe_Computation/Slk_NvmPflash.c* file. The Safety Manual recommends performing this scan during one driving cycle; the SM through the `runIntegrityCheckPFLASH()` function is called only one time in `/SafetyLiteKit/Slk_Main.c`.

SM:NVM.PLASH.UPDATE_CHECK

For the implementation of this SM, a dummy safety data is stored at PF1 in an array at 0xA0400000 address, where the expected CRC value is also known. Whenever this safety data is updated, it is required to calculate the CRC of this updated safety data via the FCE module. After calculation the CRC, the application SW compares

the calculated CRC with the expected value. If the values do not match the expected CRC, a software alarm will be generated.

There is also the possibility to inject single bit error, double bit error, and multiple bit error via the play icon on the OneEye interface. The single-bit and double-bit error can be detected and corrected, whereas the multiple-bit error is only be detected, not corrected, in the AURIX™ TC3xx microcontroller.

The following are the steps to inject an error into the flash memory:

- Erase and write flash to the dedicated data
- Read the PFLASH
- Automatically ECC value of One page will be calculated and stored in *MODULE_PFI1.ECCR.B.RCODE*
- Store the ECC in WCODE, i.e., *MODULE_DMU.HF_ECCW.B.WCODE = MODULE_PFI1.ECCR.B. RCODE*
- Now change the configuration, that next time when read the flash take ECC value from WCODE
- Write flash the injected error data. You can only enter error from 0 to 1 but not the other way around
- Not you can change configuration again to automatic ECC
- Make dummy read again to trigger ECC error

For the implementation of this SM, the initialization of the data is performed by the `initUpdateDataPFLASH()` called in */SafetyLiteKit/Slk_Main.c*. All actions such as erase and write are performed in the */SafetyLiteKit/Safe_Computation/Slk_Pflash_Programming.c* file, while memory initialization and CRC performed with FCE are computed in the */SafetyLiteKit/Safe_Computation/Slk_NvmPflash.c* file.

SM:NVM.PFLASH:WL_FAIL_DETECT

If a Double Bit Error (DBE) has occurred, there is a high probability that the complete wordline is corrupted. Therefore, upon every DBE-related SMU alarm, the application software shall verify the integrity of PFLASH wordline in which the DBE occurred by reading the next 10 pages and monitoring the multiple bit errors.

The execution of `runWordlineFailDetectPFLASH()` is executed in the */SafetyLiteKit/Slk_Main.c* file when a DBE has occurred.

5.3.2.2 OneEye interface

In the OneEye interface, errors can be injected for the PFLASH NVM using three play icons, which are implemented as buttons. These play icons can trigger single-bit error (SBE), the double-bit error (DBE), and multiple bit error (MBE). This section is highlighted in red in Figure 24 .

As mentioned in Section 5.3.2.1, only the SBE and the DBE are detected and corrected. When the MBE injection is triggered, it is not corrected so a trap of class 4 is detected, and a software alarm is triggered because of the trap handling. See section 5.2.4 for more information about the trap handling. As shown in Table 7, reset the trap handling alarm to observe the MBE alarm.

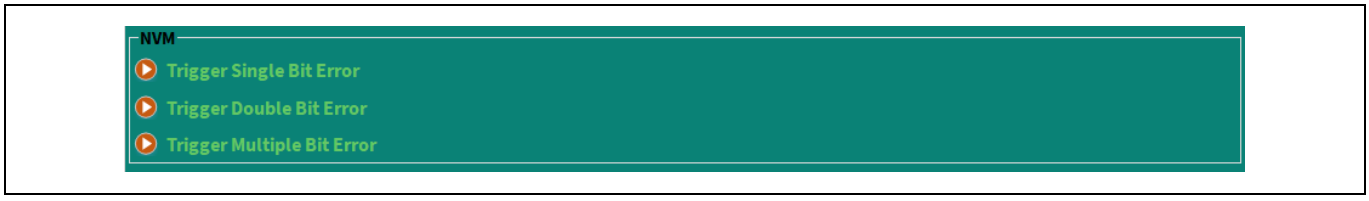

Figure 24 NVM Safe computation

Table 7 Bit error injection SMU alarms

NVM error type	Alarm
Single Bit Error (SBE)	
Double Bit Error (DBE)	
Multiple Bit Error (MBE)	 

5.3.3 Flexible CRC Engine (FCE)

The FCE is a hardware unit connected as a slave to the SPB that provides an acceleration engine for CRC algorithms. For more information about the FCE module, see the Flexible CRC Engine (FCE) Chapter in the AURIX™ TC3xx User's Manual part 1 [1].

5.3.3.1 Implementation

For the Safety lite kit, the FCE module is used to perform CRC32 bits checks. The SM:FCE:CRC_CFG which describes the implementation of the FCE module in */SafetyLiteKit/Safe_Computation/Slk_FCE.c* in the source code.

5.3.3.2 OneEye interface

For the FCE module, the CRC32 is used to perform integrity check for safety mechanisms, but there is no specific interface implementation for this module in OneEye.

5.3.4 Direct Memory Access (DMA)

The Direct Memory Access controller moves data from source addresses to destination addresses without the intervention of the CPU or other on chip devices. 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 a single 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.

5.3.4.1 Implementation

The safety lite kit implements all SM related to DMA are in */SafetyLiteKit/Safe_Computation/Slk_Dma.c*. Cyclically during the runtime, the `initAndRunDmaTransaction()` function is called in */SafetyLiteKit/Slk_Main.c* to initialize the DMA communication and move a memory array from a source to a destination.

5.3.4.2 OneEye interface

For the implementation of error injection for a DMA transaction, it is performed with play icon (used as a button) as describe in Figure 25.

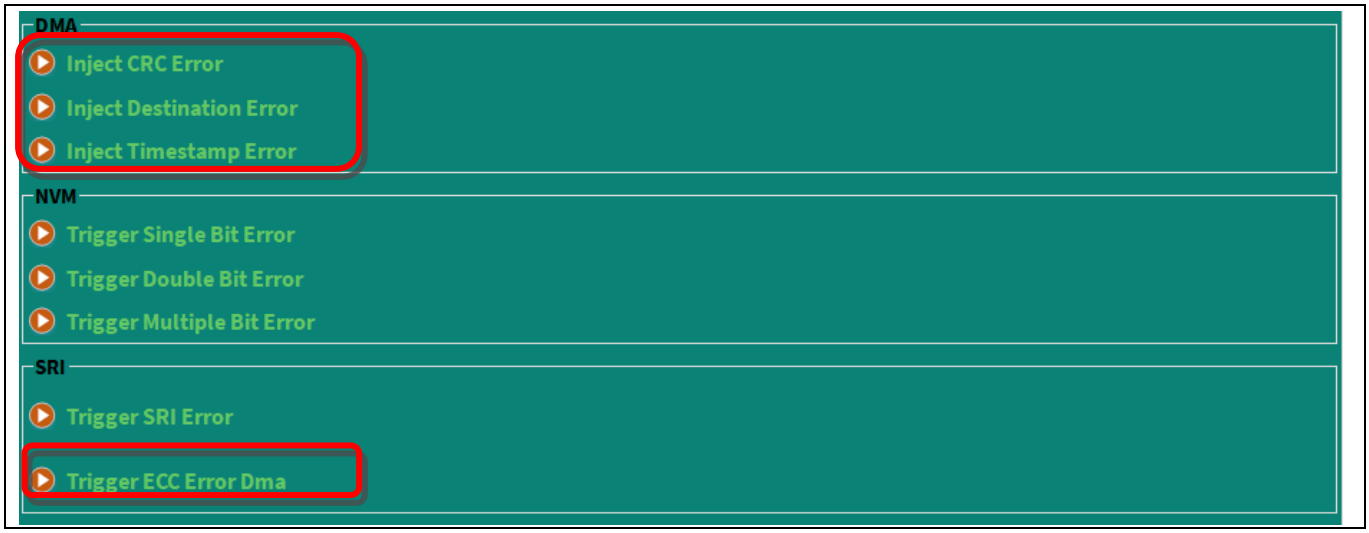

Figure 25 DMA error injection

All safe reactions corresponding to DMA errors injection are listed in Figure 26. The DMA software alarm is the same for all error injection scenarios in DMA Safe Computation, but with an additional dedicated alarm message for each one. For the SRI DMA error, a specific alarm is configured in the SMU.

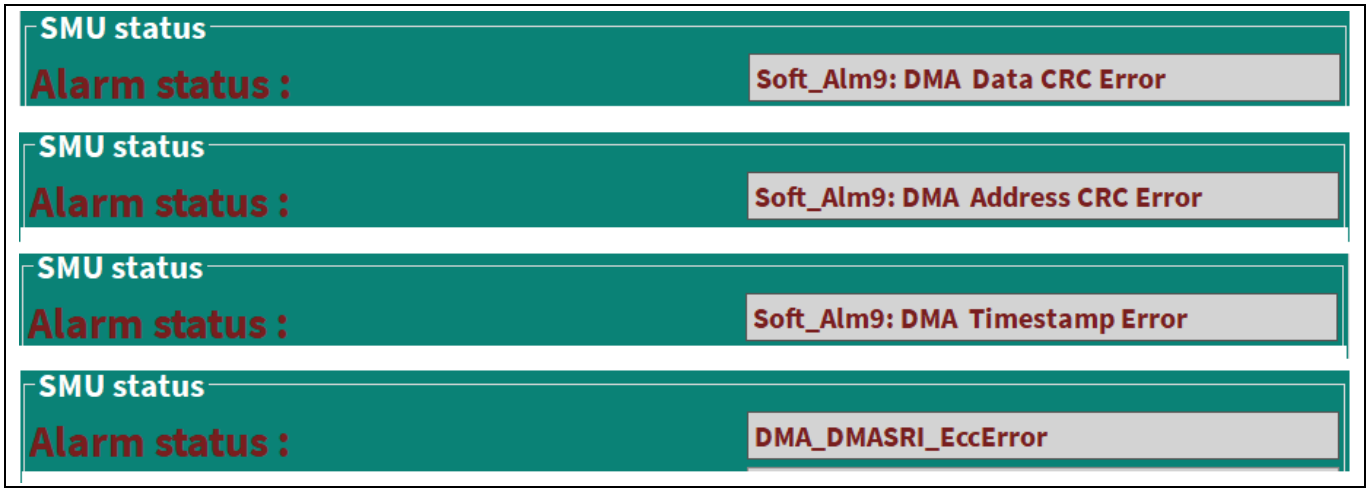

Figure 26 DMA alarm

5.3.5 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)
- Back Bone Bus (BBB)

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

The 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.

The three types of error are:

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

5.3.5.1 Implementation

The safety application kit contains the implementation of Safety Mechanism SRI:ERROR_HANDLING, as described in the Safety Manual V2.0, which says that if SRI triggers an error interrupt request, the application SW shall check the type of error SRI and stored SRI diagnostic information. The application SW shall evaluate the type of error and triggers 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. For more information, see the function `storeSriDiagnosticInfo()` and `runSriErrorInjection()` function at path *SafetyLiteKit\Safe_Computation\Slk_Sri_Error_Handling.c*.

5.3.5.2 OneEye interface

For the implementation of error injection for SRI error injection, it is performed with play icon (used as a button), as shown in Figure 27.



Figure 27 Safe computation – SRI error injection

The safe reaction of SRI error is shown in Figure 28.



Figure 28 Safe computation – SRI safe reaction

5.3.6 System Timer (STM)

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

Permanent and transient hardware faults in the STM may corrupt the timer value or the interrupt generation. As the STM is not part of the duplication area of the CPU, the Application Software is in charge of executing plausibility checks using an independent timer (refer to *Safety Mechanism STM:MONITOR* in the Safety Manual [2]).

5.3.6.1 Implementation

The *Safety Mechanism STM:MONITOR* states that the plausibility check should be implemented using a secondary timing module. Since each CPU has a dedicated system timer, one STM can be used to monitor the other, and vice versa. In this example, it is implemented in a way that every CPU should call the *runStmMonitoring* function periodically. By calling the 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 its own 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 2 ms, an appropriate reaction shall be triggered (e.g., application reset).

The `runSTMMonitoring()` function located in the path */SafetyLiteKit/Safe_Computation/Slk_StmMon.c* is triggered every millisecond in the `runSlkStmIsr()` function in */SafetyLiteKit/Slk_Main.c* by each CPU.

5.3.6.2 OneEye interface

Error injection for STM is implemented using a play icon (used as a button), as described in Figure 29.



Figure 29 STM error injection

The safe reaction of STM is shown in Figure 30. An application reset is the minimum required action after this safe reaction.



Figure 30 STM alarm

5.4 Digital acquisition

The following section covers various application examples for the acquisition of safety-related digital signals by the usage of the 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). For more information on the nominal functionality of these modules, see the AURIX™ TC3xx User Manual [1].

As described in the AURIX™ TC3xx safety manual [2], the safety-related function “digital acquisition" is composed of five different use cases for utilizing the timer modules to actuate safety-related digital signals across various applications.

The following list and Table 8 present an overview of all digital acquisition-related functional use cases mentioned in the Safety Manual [2], and along with a list of functional blocks and SMs required for their implementation.

- FUC**0**: Digital acquisition with redundant TIM/TIM channels
- FUC**1**: Digital acquisition with redundant TIM/TIM channels and HW diversity
- FUC**2**: Digital acquisition with redundant CCU6/TIM channels
- FUC**3**: Digital acquisition with redundant CCU6/GPT12 channels
- FUC**4**: Digital acquisition with single CCU6/GPT12 channels

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

FUC	Functional block involved	SMs required
Digital acquisition FUC0	• 2x GTM TIM	• *Safety Mechanism TIM_REDUNDANCY* • *Safety Mechanism TIM_CLOCK_MONITORING*
Digital acquisition FUC1	• 2xGTM TIM (diversity)	• *Safety Mechanism TIM_REDUNDANCY* • *Safety Mechanism TIM_CLOCK_MONITORING*
Digital acquisition FUC2	• GTM TIM • CCU6	• *Safety Mechanism GTM_CCU6_REDUNDANCY* • *Safety Mechanism TIM_CLOCK_MONITORING*
Digital acquisition FUC3	• CCU6 • GPT12	• *Safety Mechanism CCU6_CAPTURE_MON_BY_GPT12*
Digital Acquisition FUC4	• CCU6 • GPT12	• *Safety Mechanism CCU6_CAPTURE_MON_BY_GPT12*

5.4.1 Overview

Table 9 shows the digital signals which are involved in the implementation of digital acquisition.

Table 9 Overview of digital signals and their GTM/CCU6/GPT12 functionality

ID	Signal	Input pin/ symbol	GTM/GPT12/CCU6 functionality
1	TIM mission signal	P02.3	GTM TIM1 channel 3
2	TIM monitor signal	P33.12	GTM TIM2 channel 0
3	TOM safety-related digital output signal. Dummy signal to simulate a digital safe input signal as TOM mission signal in digital actuation.	P00.0	GTM TOM0 channel 4, TOUT9
4	TOM safety-related digital output signal. Second dummy signal (HW diversity) to simulate a digital safe input signal as TOM mission signal in digital actuation.	P15.4	GTM TOM2 channel 15, TOUT75
5	CCU6 Monitor Signal	P00.1	CCU60 CC60INB input
6	External clock 0 (ECLK0) output signal (GTM external clock to port mapping)	P23.1	GTM_CLK0 signal (alternate output 4)
7	GTM TIM ECLK input signal	P15.2	GTM TIM2 channel 5
8	CCU6 Mission Signal (redundant)	P00.1	CCU60 CC60INB input
9	GPT12 Monitor Signal (redundant)	P33.7	GPT12 T2INB
10	CCU6 Mission Signal (single)	P00.7	CCU61 CC60INC
11	GPT12 Monitor signal (single)	P00.7	GPT12 T2INA

All the functional use cases (FUC0 – FUC4) need a special pin connection via a connecting cable. Their pin connections are explained in each FUC, and the user needs to connect accordingly before enabling the particular FUC, otherwise, alarm will be triggered.

5.4.1.1 User interface

Figure 31 shows the selection menu (in Menu 2 tab) for digital acquisition and digital actuation function use cases. By selecting one of the eight available options, the specific FUC is initialized. Further information of the FUC can be observed via the debugger `g_digitalAquisitionStatus`. The highlighted section in Figure 31 is the dedicated selection for digital acquisition use cases.

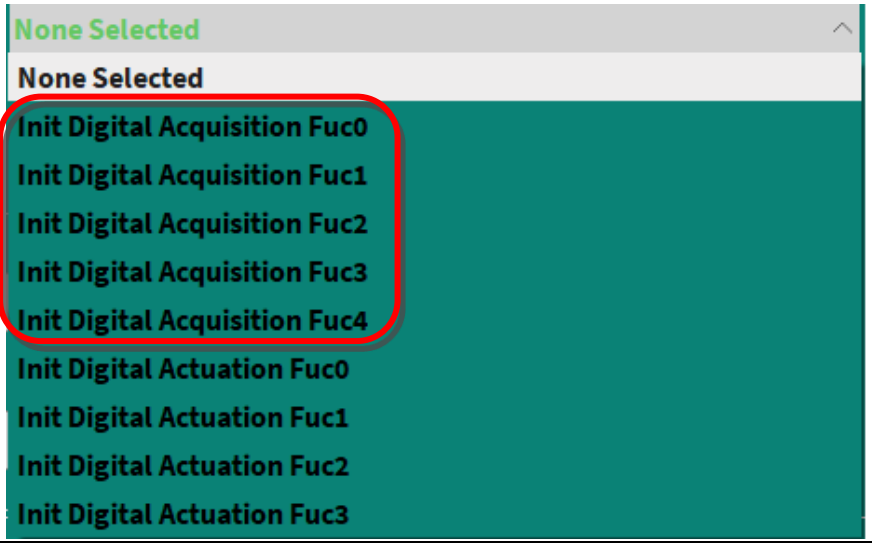

Figure 31 Digital acquisition function use cases selection

5.4.1.2 Software structure

The current implementation of safety lite kit includes FUC0, FUC1, FUC2, and FUC3 of digital acquisition. The SMs required for each FUC are fully implemented according to the recommendation of the Safety Manual [2]. As shown in Figure 32, a global source and header file, and additional files for each use cases are provided, along with a clock monitor file. 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.

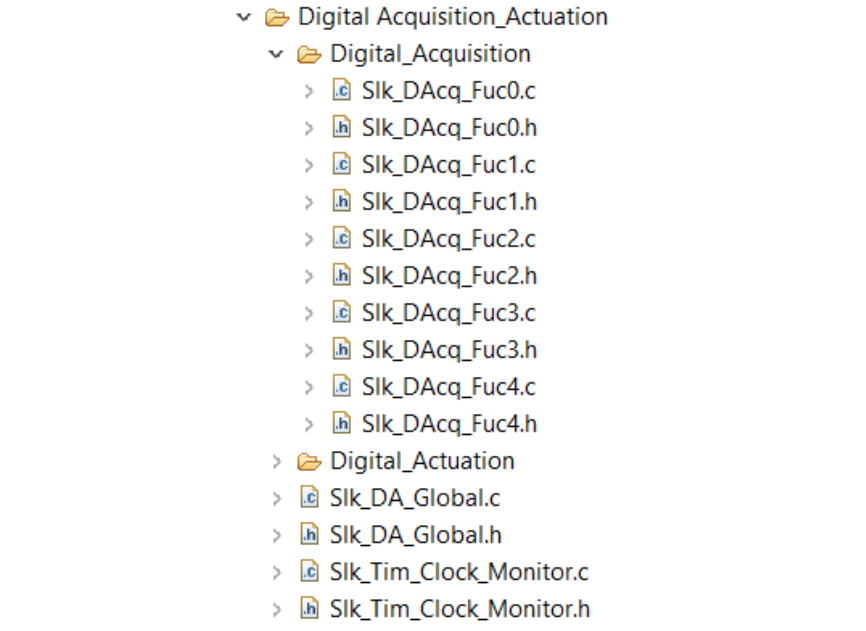

Figure 32 Digital acquisition source files

5.4.2 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.

5.4.2.1 Implementation

An illustrative example of the *Safety Mechanism TIM_REDUNDANCY* is shown in Figure 33.An overview of the Application Kit Safety implementation including the *Safety Mechanism TIM_CLOCK_MONITORING* and the pin connections is shown in Figure 34. The port pins used for this SM are as follows:

- P02.3 (TIM1_CH3, 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 feeds to S1 and S2

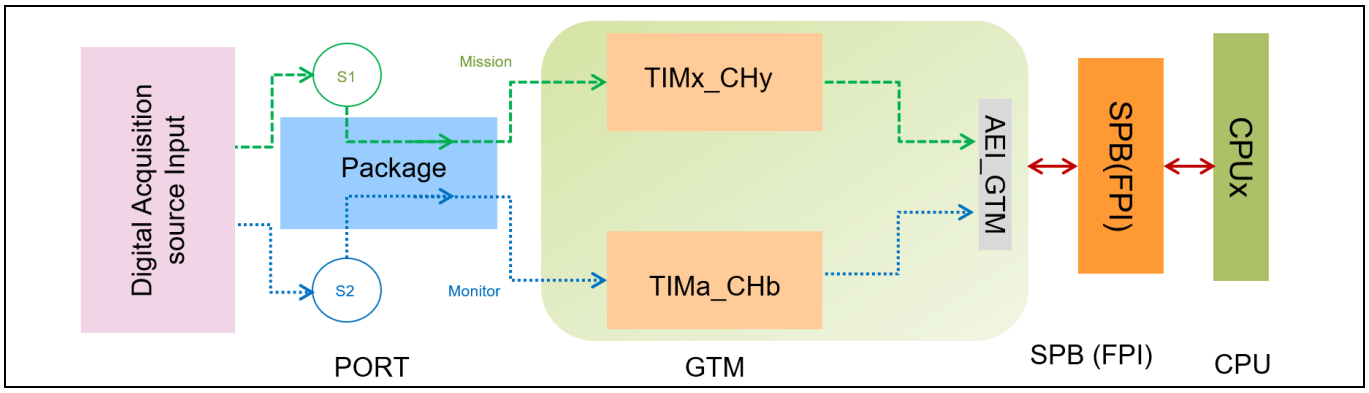

Figure 33 Illustrative example of safety mechanism TIM_REDUNDANCY

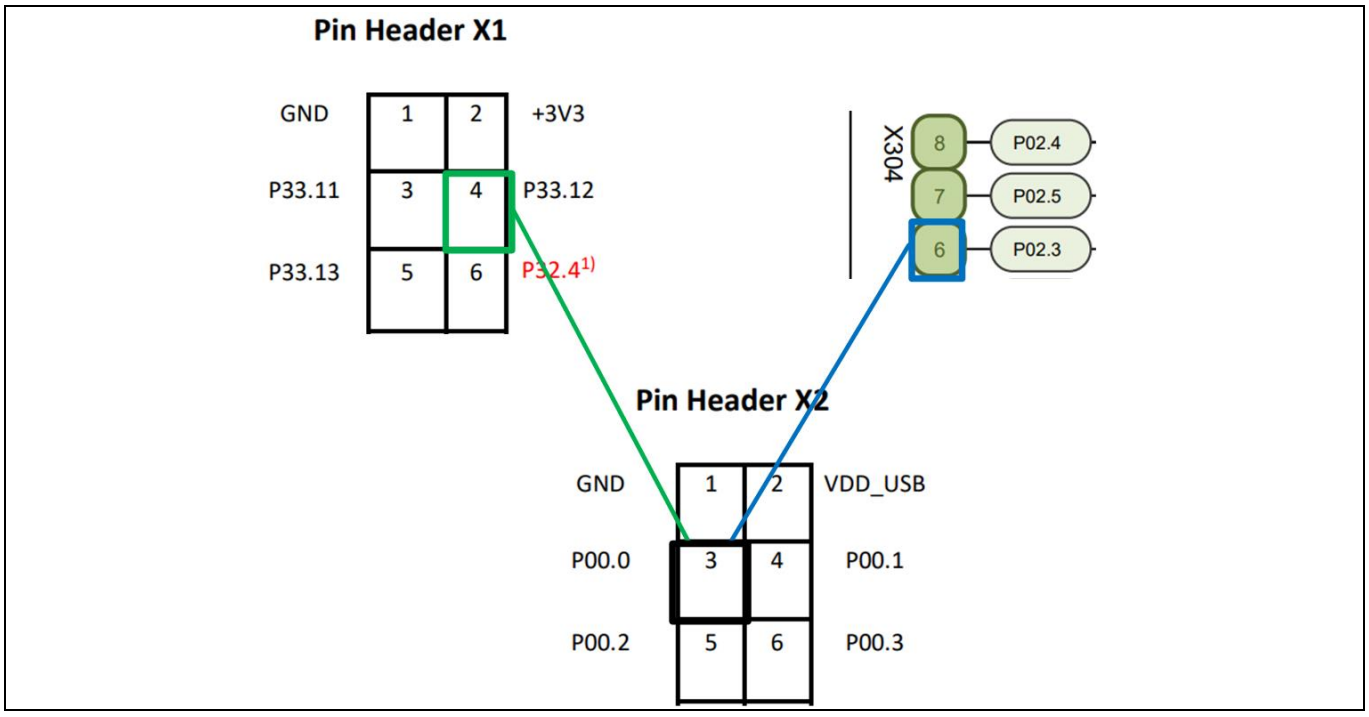

Figure 34 Pin connection for TIM_REDUNDANCY

5.4.2.2 OneEye interface

Once the connection is done, select FUC0 to run this function use case. To inject an error, disconnect the dummy PWM signal either from S1 or S2 will be trigger the corresponding alarm, as shown in Figure 35.

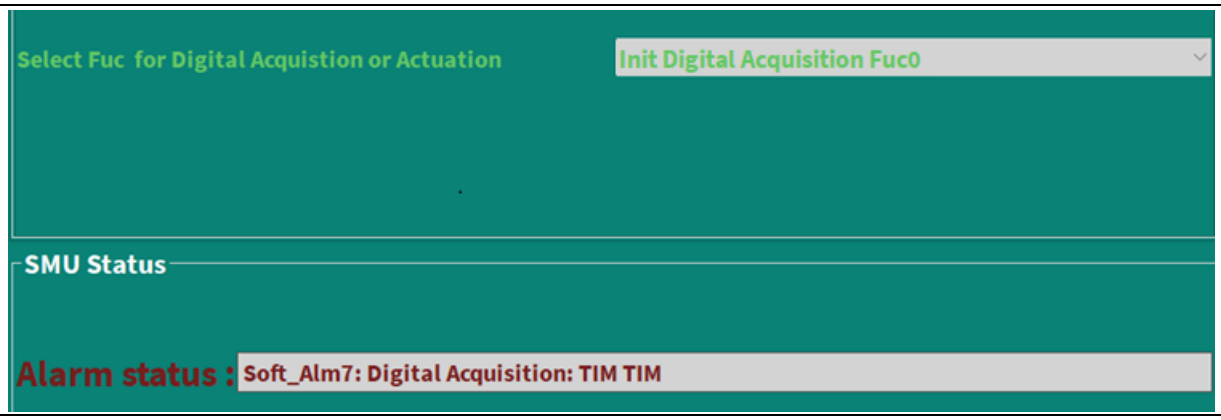

Figure 35 TIM_REDUNDANCY alarm

5.4.3 Redundant TIM/TIM channels and HW diversity (FUC1)

In FUC0 of the safety-related function “digital acquisition”, two diverse 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.

5.4.3.1 Implementation

An illustrative example of the *Safety Mechanism TIM_REDUNDANCY and HW Diversity* is shown in Figure 36. An overview of the Application Kit Safety implementation including the *Safety Mechanism TIM_CLOCK_MONITORING* and the pin connections is shown in Figure 37. The port pins used for this SM are as follows:

- P02.3 (TIM1_CH3, IN) as S1 mission “TIMx_CHy”
- P33.12 (TIM2_CH0, IN) as S2 monitor “TIMa_CHb”
- P00.0 (TOM0_CH4, TOUT9, OUT) generate dummy PWM1 which feeds to S1
- P15.4 (TOM2_CH15, TOUT75, OUT) generate dummy PWM2 which feeds to S2

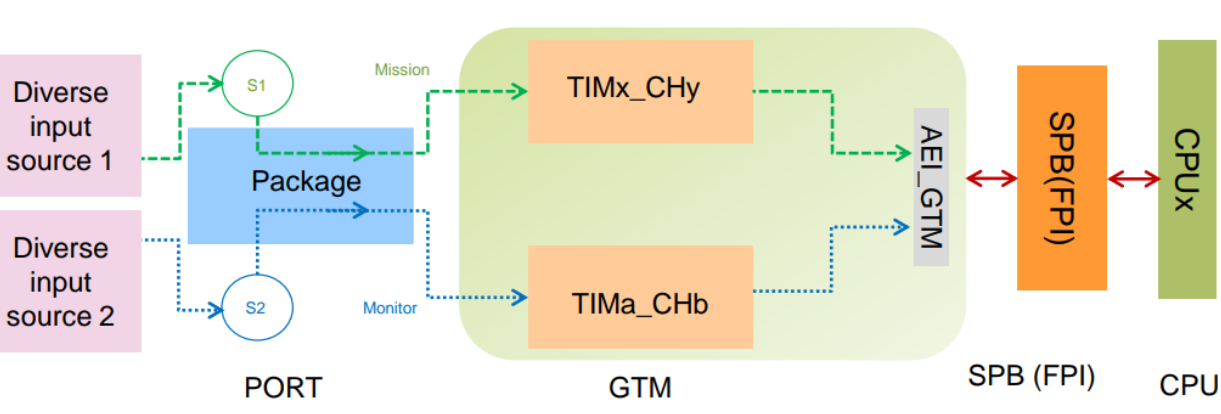

Figure 36 Illustrative example of safety mechanism TIM_REDUNDANCY and HW diversity

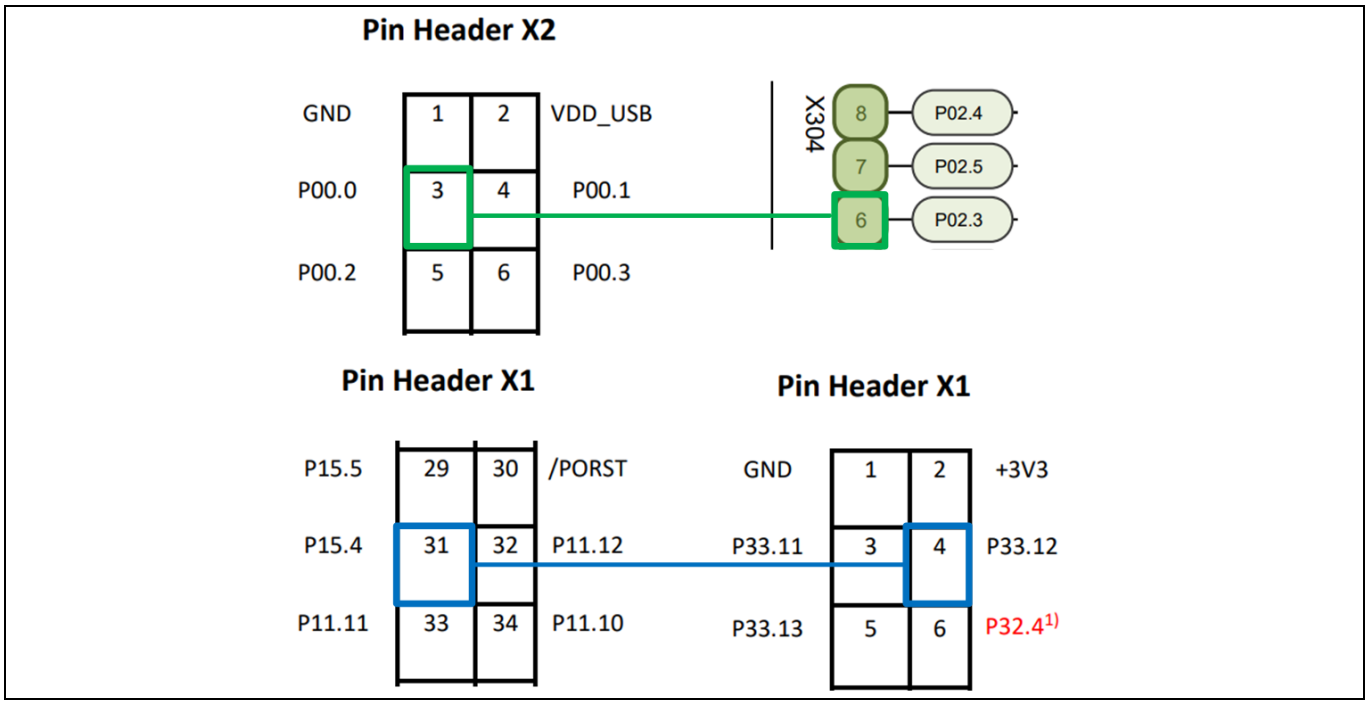

Figure 37 Pin connection for TIM_REDUNDANCY HW diversity

5.4.3.2 OneEye interface

Once the connection is done, select FUC1 in the OneEye GUI to run this functional use case. To inject an error, disconnect the dummy PWM1 signal from S1 or the dummy PWM2 signal from S2. The corresponding alarm will be triggered as shown in Figure 38.

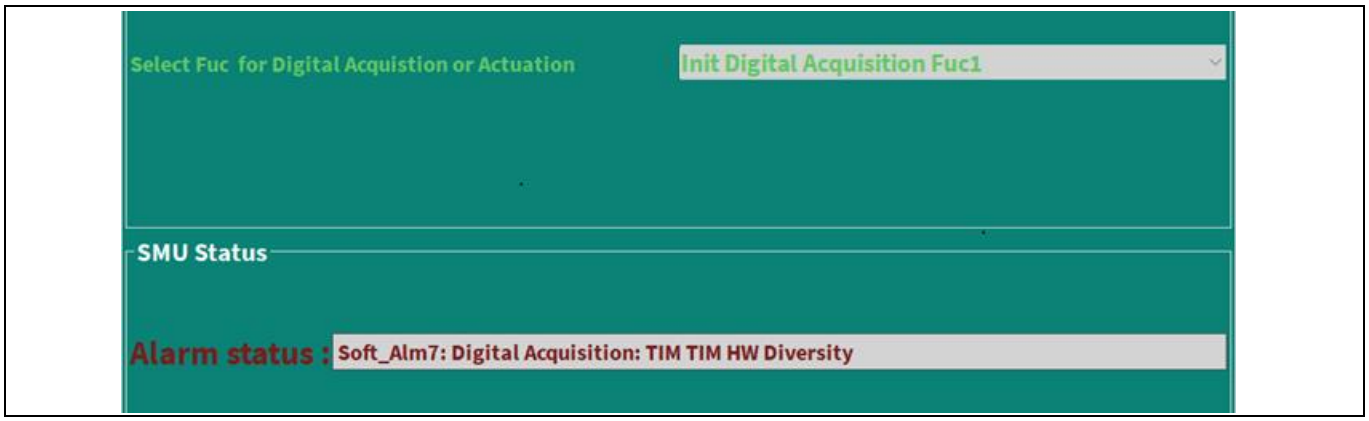

Figure 38 TIM_REDUNDANCY HW diversity alarm

5.4.4 Redundant CCU6/TIM channels (FUC2)

In FUC2 of the safety-related function “digital acquisition”, two redundant digital signals are acquired by redundant GTM Timer Input Module(TIM) resources 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 FUC2.

5.4.4.1 Implementation

An illustrative example of the *Safety Mechanism GTM_CCU6_REDUNDANCY* is shown in Figure 39. An overview of the safety lite kit implementation including the *Safety Mechanism TIM_CLOCK_MONITORING* and the pin connections is shown in Figure 40. The port pins used for this SM are as follows:

- P33.12 (TIM2_CH0, IN) as S1 monitor “TIMx_CHy”
- P00.1 (CCU60_CC60INB, IN) as S2 monitor “CCU6”
- P00.0 (TOM0_CH4, TOUT9, OUT) generate dummy PWM which feedback to S1 and S2

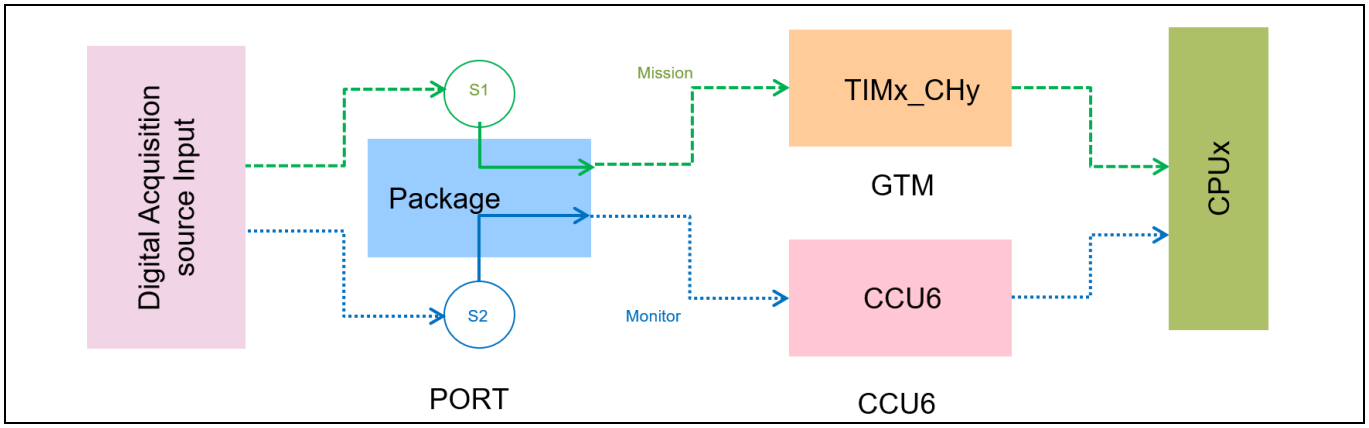

Figure 39 Illustrative example of safety mechanism GTM_CCU6_REDUNDANCY

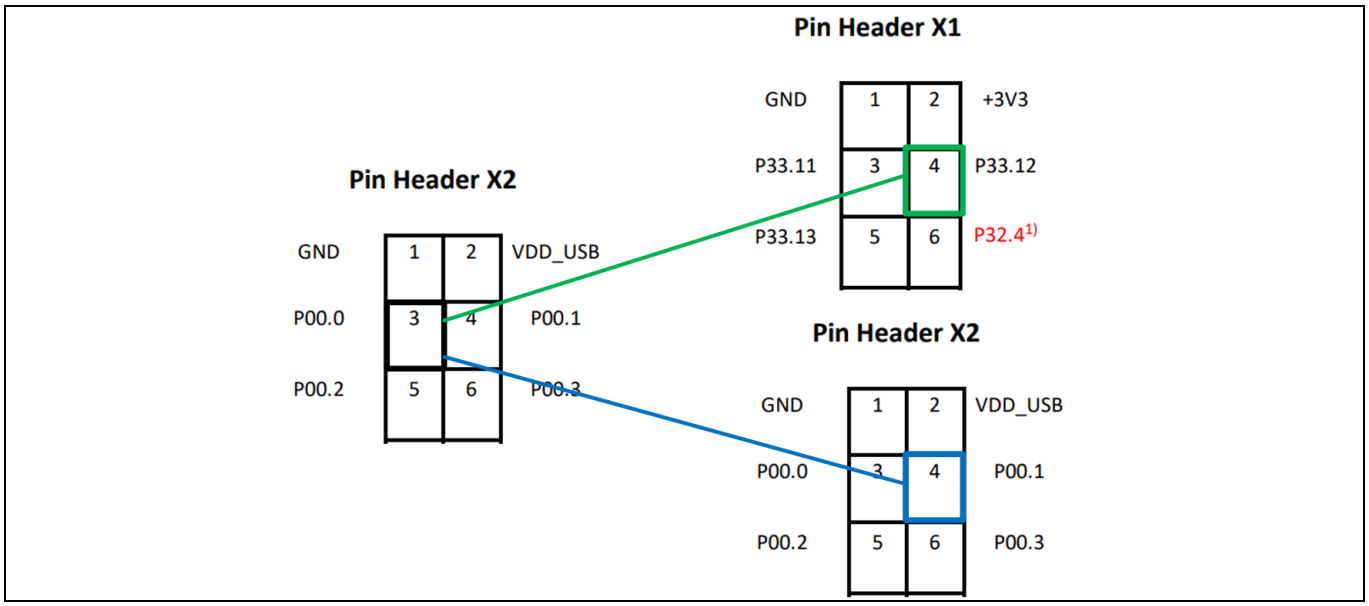

Figure 40 Pin connection for GTM_CCU6_REDUNDANCY

5.4.4.2 OneEye interface

Once the connection is done, select FUC2 in the OneEye GUI to run this function use case. To inject an error, disconnect the dummy PWM signal from S1 or from S2. The corresponding alarm will be triggered as shown in Figure 41.

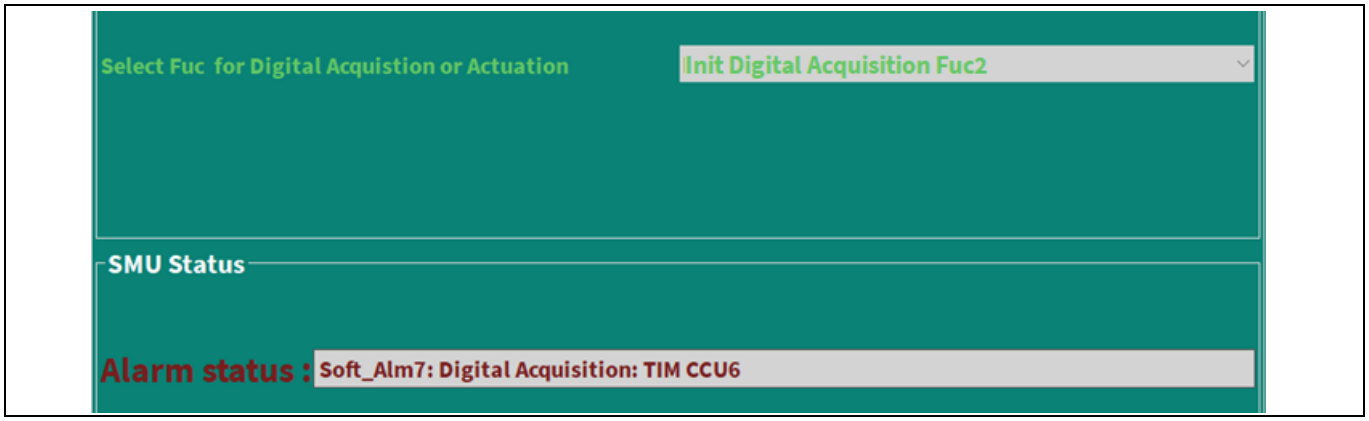

Figure 41 GTM_CCU6_REDUNDANCY alarm

5.4.5 Redundant CCU6/GTP12 channels (FUC3)

In FUC3 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/frequency 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 FUC3.

5.4.5.1 Implementation

An illustrative example of the *Safety Mechanism CCU6_CAPTURE_MON_BY_GPT12* is shown in Figure 42. An overview of the safety lite kit implementation and the pin connections is shown in Figure 43. The port pins used for this SM are as follows:

- P00.1 (CCU60_CC60INB, IN) as S1 mission “CCU6_Tx”
- P33.7 (GPT12 T2INB) as S2 monitor “GPT12_Ta”
- P00.0 (TOM0_CH4, TOUT9, OUT) generate dummy PWM which feeds S1 and S2

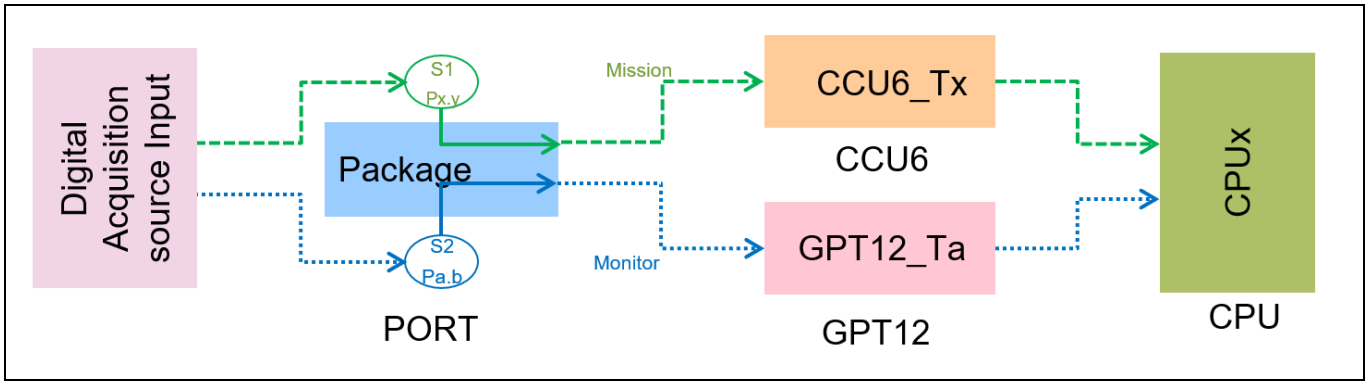

Figure 42 Illustrative example of safety mechanism CCU6_CAPTURE_MON_BY_GPT12

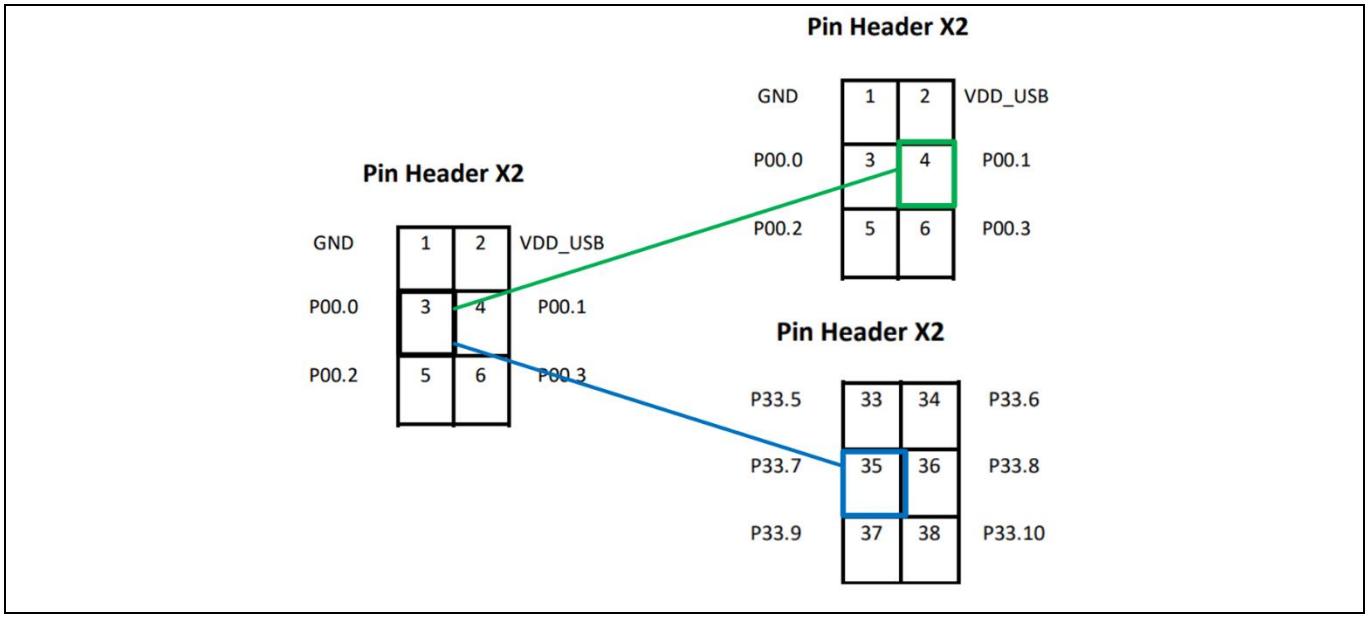

Figure 43 Pin connection for CCU6_CAPTURE_MON_BY_GPT12

5.4.5.2 OneEye interface

Once the connection is done, select FUC3 in the OneEye GUI to run this functional use case. To inject an error, disconnect the dummy PWM signal from S1 or from S2. The corresponding alarm will be trigger as shown in Figure 44.

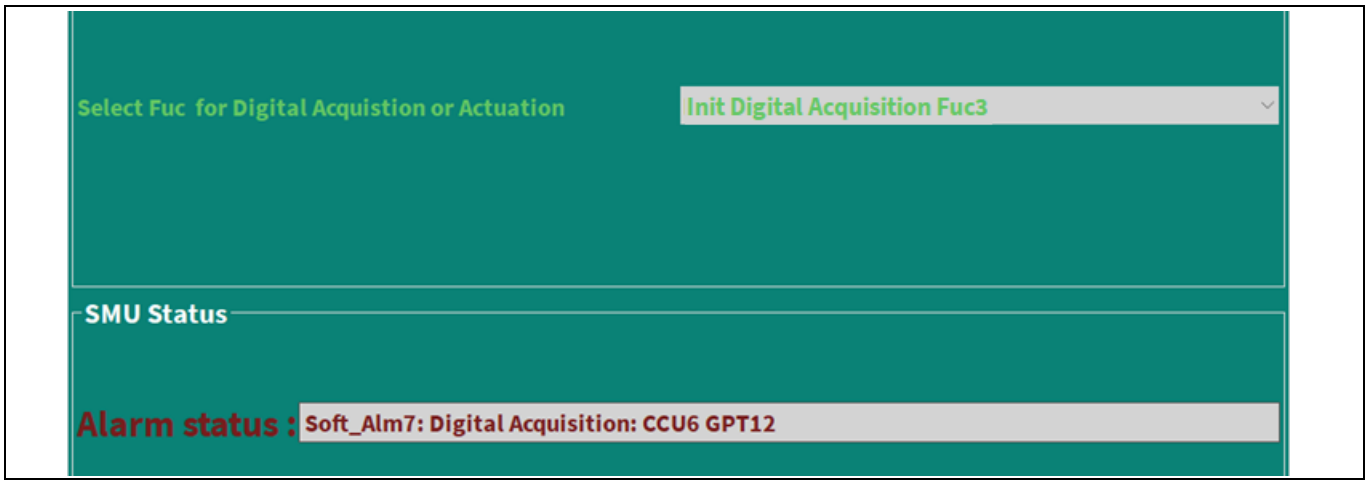

Figure 44 CCU6_CAPTURE_MON_BY_GPT12alarm

5.4.6 Single CCU6/GPT12 channels (FUC4)

In FUC4 of the safety-related function “digital acquisition”, the MCU receives single signal, and it is acquired by both CCU6 and GPT12. The signal measurement results (e.g., period/frequency) must be read and compared by the CPU. The SMs *CCU6_CAPTURE_MON_BY_GPT12* are required for the implementation of this FUC3.

5.4.6.1 Implementation

An illustrative example of the *Safety Mechanism CCU6_CAPTURE_MON_BY_GPT12* is shown in Figure 45. An overview of the safety lite kit implementation and the pin connections is shown in Figure 46. The port pins used for this SM are as follows:

- P00.7 (CCU61_CC60INC, IN) as S1 mission “CCU6_Tx”
- P00.7 (GPT12 T2INA) as S2 monitor “GPT12_Ta”
- P00.0 (TOM0_CH4, TOUT9, OUT) generate dummy PWM which feeds S1 and S2

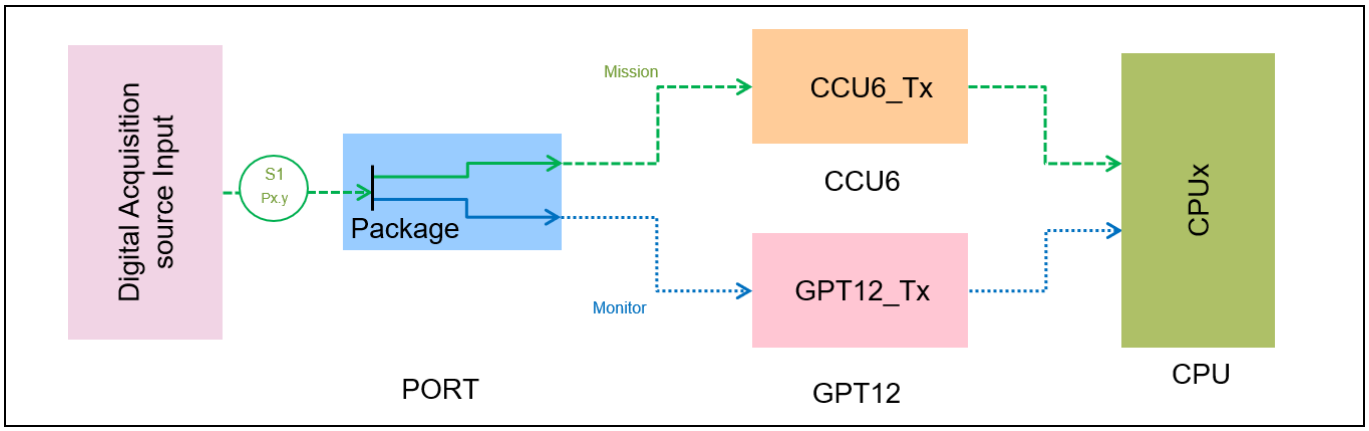

Figure 45 Illustrative example of safety mechanism CCU6_CAPTURE_MON_BY_GPT12

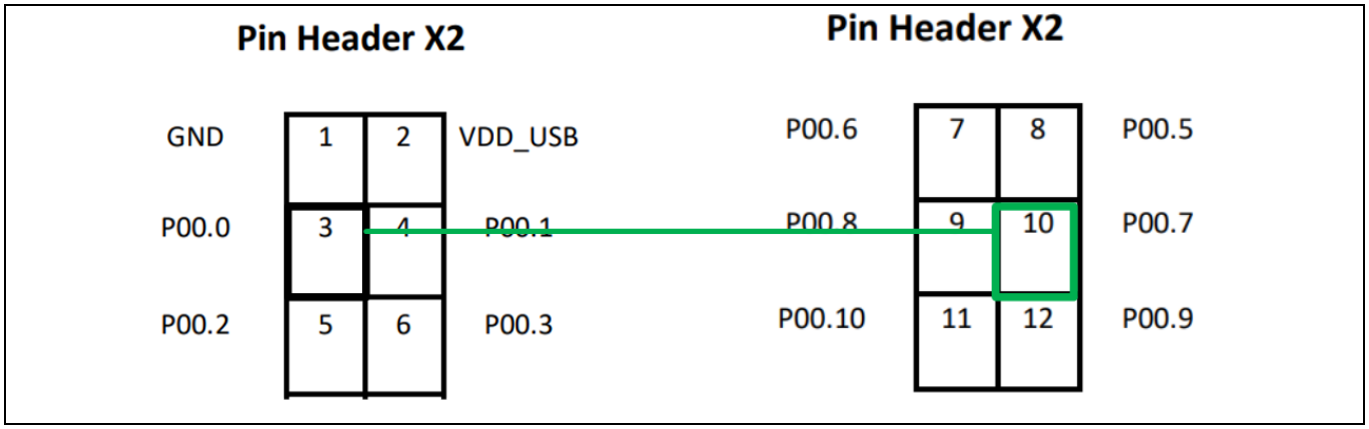

Figure 46 Pin connection for CCU6_CAPTURE_MON_BY_GPT12

5.4.6.2 OneEye interface

Once the connection is done, in the OneEye GUI, FUC4 need to be selected to run this function use case. To inject an error, disconnect the dummy PWM signal from S1 or from S2, the according alarm will be triggered as shown in Figure 47.

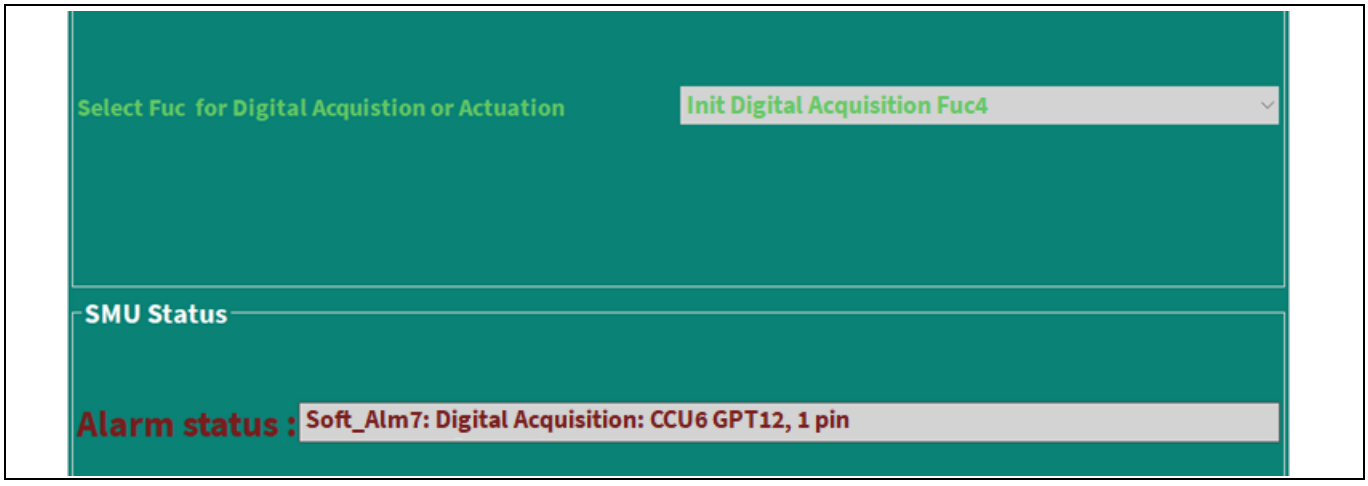

Figure 47 CCU6_CAPTURE_MON_BY_GPT12 alarm

5.5 Digital actuation

As described in the AURIX™ TC3xx Safety Manual [2], the safety-related function “digital actuation" is composed of four different use cases of these timer modules for actuation of safety-related digital signals in different applications.

The following list and Table 8 present an overview of all functional use cases related to digital actuation mentioned in the AURIX™ TC3xx Safety Manual [2]. It also provides a list of functional blocks and SMs required for their implementation.

- 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 actuation functional use cases

FUC	Functional block involved	SMs required
Digital actuation FUC**0**	• GTM TOM • IOM	• *Safety Mechanism IOM_ALARM_CHECK* • *Safety Mechanism TIM_CLOCK_MONITORING*
Digital actuation FUC**1**	• GTM TOM • 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

5.5.1 Overview

Table 11 shows the digital signals involved in the implementation of digital actuation function use cases.

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

ID	Signal	Input pin/ symbol	GTM/IOM/CCU6 functionality
1	TOM mission signal	P00.0	GTM TOM0 channel 4, TOUT9
2	TIM Monitor signal	P02.3	GTM TIM1 channel 3
3	TOM monitor signal	P02.4	GTM TIM1 channel 12, TOU4
4	CCU6 monitor output	P02.4	CCU6 CC62 out
5	IOM GTM source input pin	P33.1	IOM GTM monitor input
6	IOM GTM reference input signal	GTM TOUT4	IOM GTM reference input
7	IOM CCU6 reference input signal	CCU60 Cc62	IOM CCU6 reference input
8	External clock 0 (ECLK0) output signal (GTM external clock to port mapping)	P23.1	GTM_CLK0 signal (alternate output 4)
9	GTM TIM ECLK input signal	P15.2	GTM TIM2 channel 5
10	CCU6 Output Mission	P00.0	CCC60 COUT63
11	GPT12 Monitor Signal	P33.7	GPT12 T2INB

5.5.1.1 User interface

Figure 48 shows the selection menu (in menu 2 tab) for digital acquisition and digital actuation function use case. By selecting one of the nine available options, the specific FUC is initialized. Further information on the execution of the FUC can be observed via the debugger (*DigitalAcqActStatusDatatype*). The highlighted part in the below figure is the dedicated selection for digital actuation use cases.

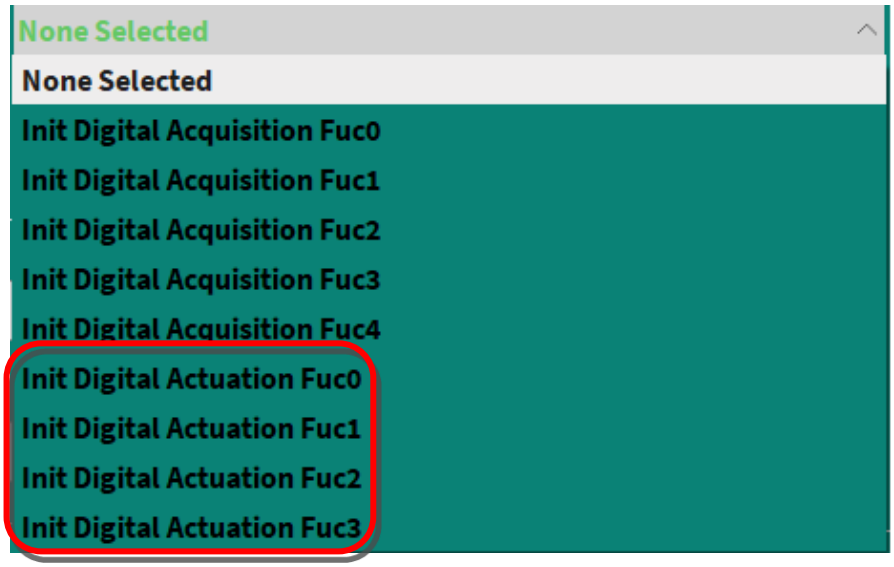

Figure 48 Digital actuation function use cases selection

5.5.1.2 Software overview

The current implementation of safety lite kit includes FUC0, FUC1, FUC2, and FUC3 of digital actuation. The SMs required for each FUC are fully implemented according to the recommendation of the Safety Manual [2]. As shown in Figure 49, a global source and header file, and additional files for every individual use case are provided, along with a configuration readback file. 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.

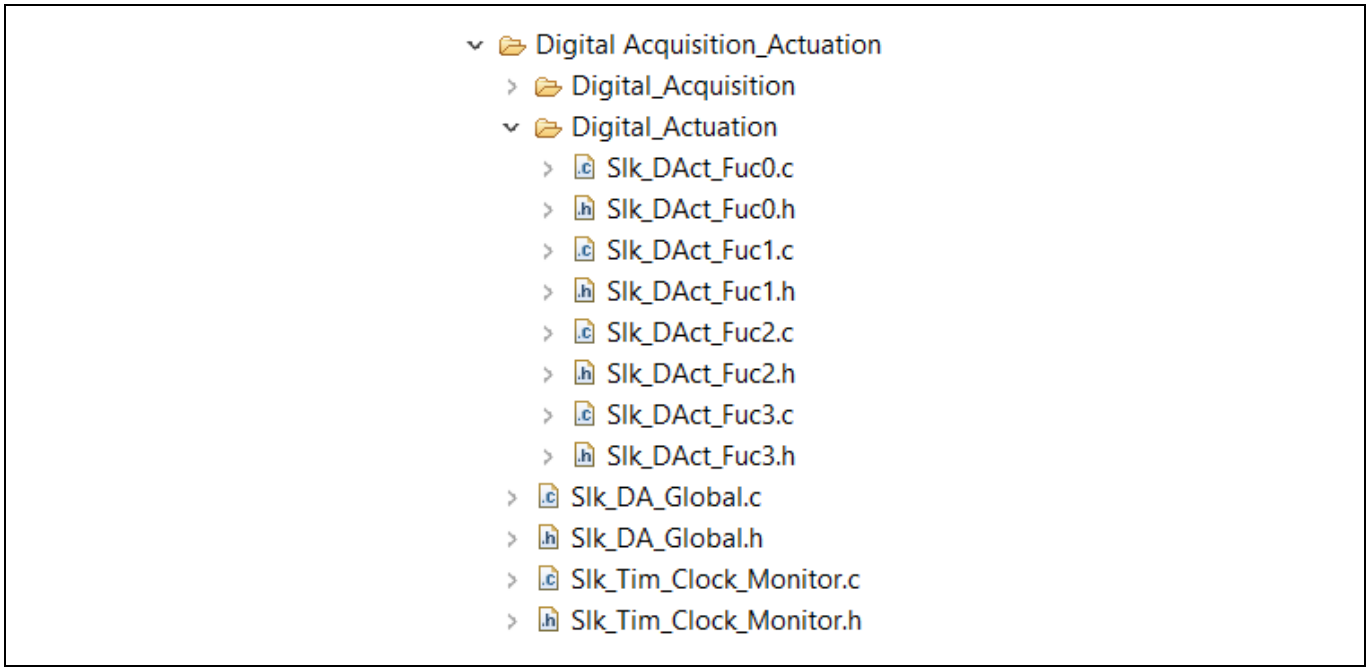

Figure 49 Digital actuation files

5.5.2 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.

5.5.2.1 Implementation

In order to demonstrate this FUC with the safety lite kit, 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 SMs required for this FUC are fully implemented according to the recommendations of the AURIX™ TC3xx Safety Manual [2].
An illustrative example of the *Safety Mechanism IOM_ALARM_CHECK* and an overview about the safety lite kit implementation can be observed in Figure 50. The port pins used for this SM are as follows:

- P00.0 (TOM0_CH4, TOUT9, OUT) as S1 mission “TOMx_CHy”.

- P33.1 (Iom_MonInput) as S2 IOM source
- T02.4 (TOM1_CH12, TOUT4, OUT) as S3
- GTM_TOUT4 (Iom_RefInput) as IOM reference

For this FUC, *Safety Mechanism TIM_CLOCK_MONITORING* is also required to check if the GTM is clocked. Therefore, the ECLK signal shall be readback by the TIM and evaluated if the value is within the expected timing. The port pins used for this SM are

- P23.1 port pin
- P15.2 (TIM2_CH5) as readback signal

The pin connections for both *Safety Mechanism IOM_ALARM_CHECK* and *Safety Mechanism TIM_CLOCK_MONITORING* shall be done as shown in Figure 51.

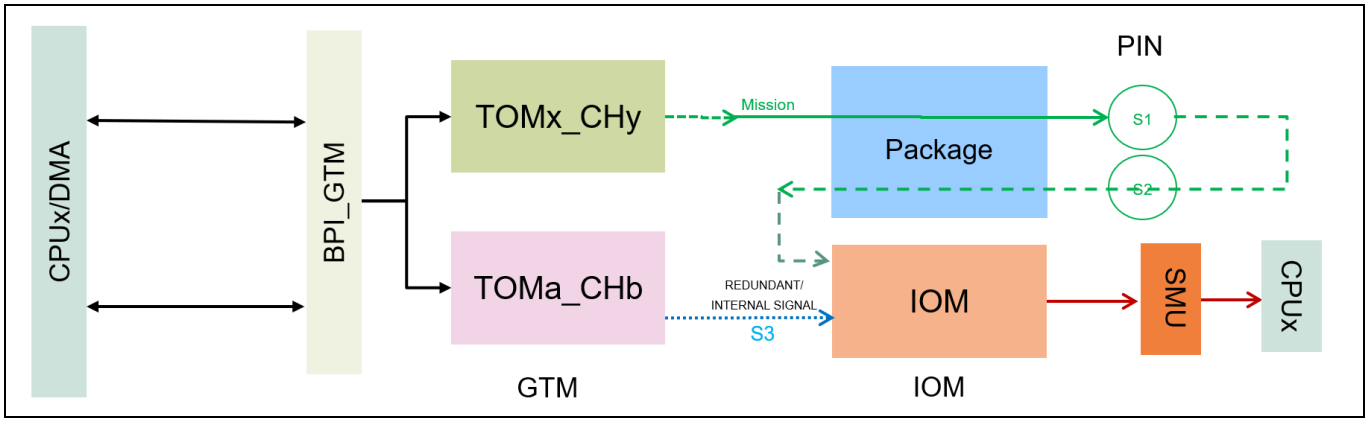

Figure 50 Illustrative example of safety mechanism IOM_ALARM_CHECK

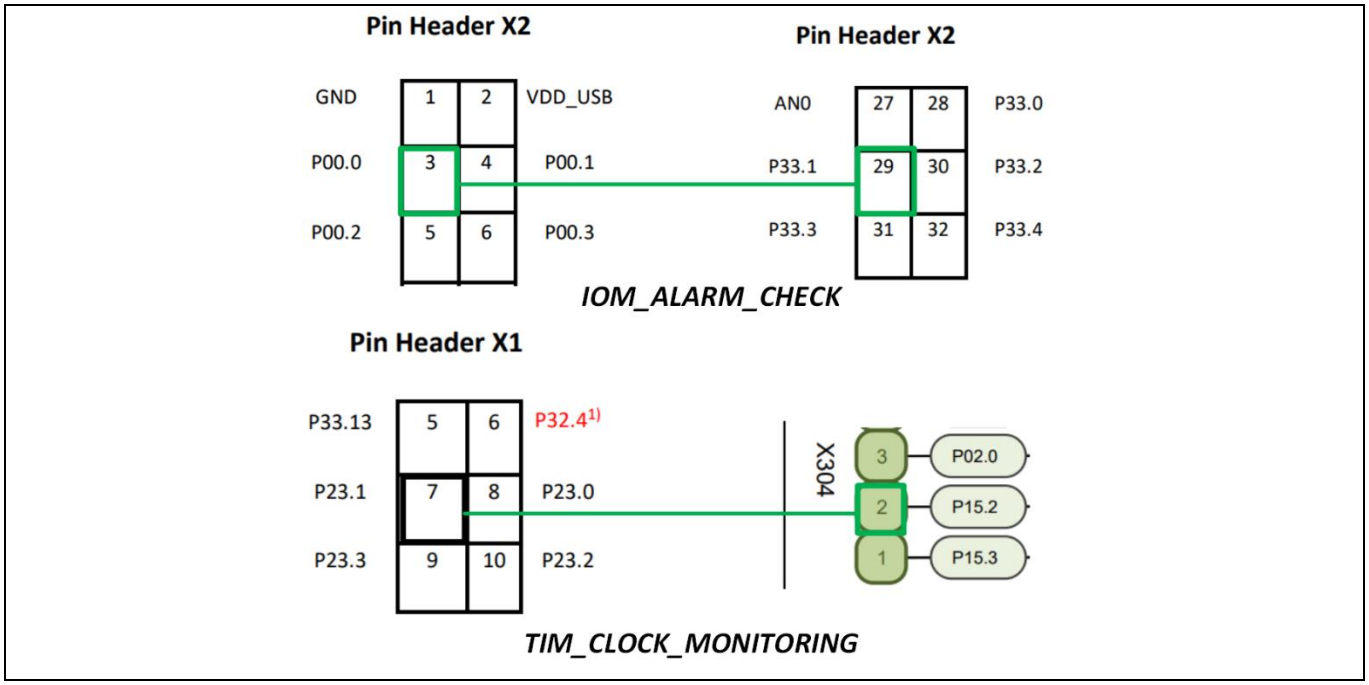

Figure 51 Pin connection for IOM_ALARM_CHECK

5.5.2.2 OneEye interface

Once the connection is done, select FUC0 in the OneEye GUI to run this functional use case. To inject an error, disconnect the PWM signal S1, and the corresponding alarm will be triggered, as shown as Figure 52.

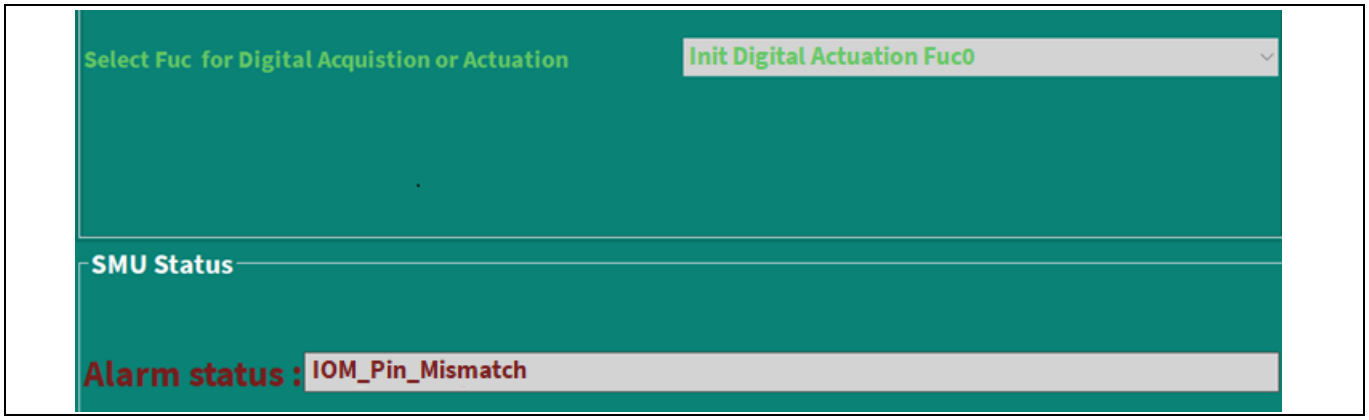

Figure 52 IOM alarm

5.5.3 Redundant TOM/CCU6 channels and IOM comparison (FUC1)

In the 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. A 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. *Safety Mechanism IOM_ALARM_CHECK* is required for this FUC.

5.5.3.1 Implementation

To demonstrate this FUC with the 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 *Safety Mechanism TOM_CCU6_MONITORING_WITH_IOM* and an overview about the safety lite kit implementation of this FUC including *Safety Mechanism TIM_CLOCK_MONITORING* can be observed in Figure 53 the pin connection shall be done as shown in Figure 54. The port pins used for this SM are as follows:

- P00.0 (TOM0_CH4, TOUT9, OUT) as S1 mission “TOMx_CHy”.
- P33.1 (Iom_MonInput) as S2 IOM source
- P02.4 (CCU60_CC62, OUT) as S3 redundant (reference) internal signal “CCU6_Ta”
- CCU60_CC62 (Iom_RefInput) as IOM reference

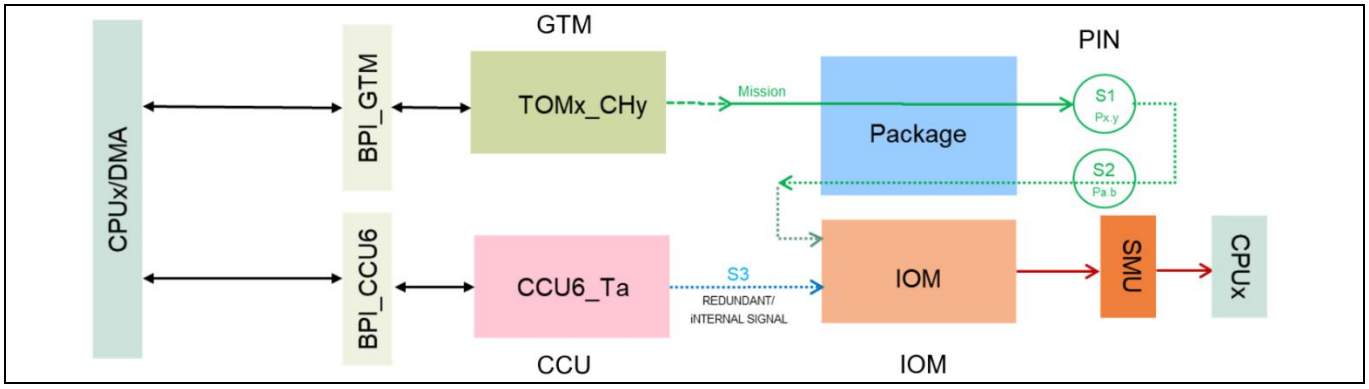

Figure 53 Illustrative example of safety mechanism IOM_ALARM_CHECK

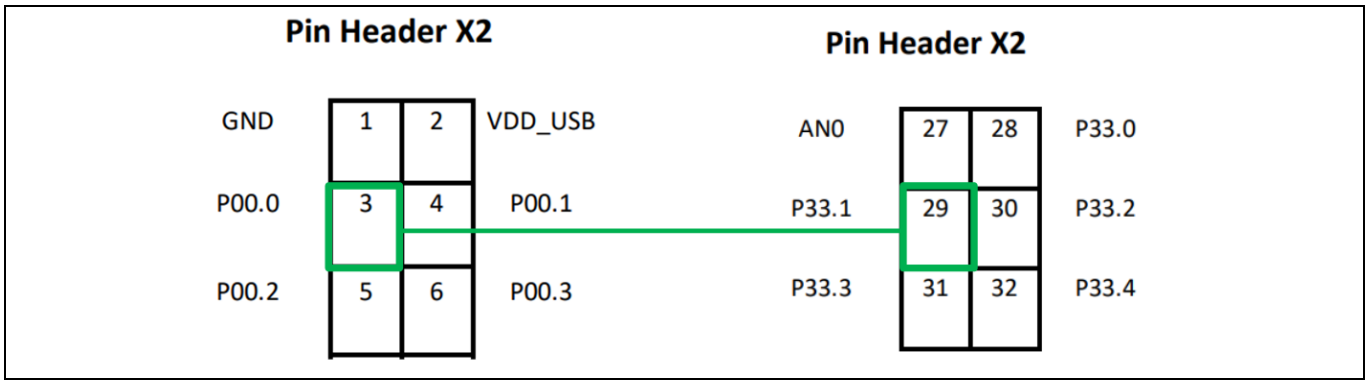

Figure 54 Pin connection for IOM_ALARM_CHECK

5.5.3.2 OneEye interface

Once the connection is done, select FUC1 in the OneEye GUI to run this functional use case. To inject an error, disconnect the PWM signal S1, and the corresponding alarm will be triggered, as shown in Figure 55.

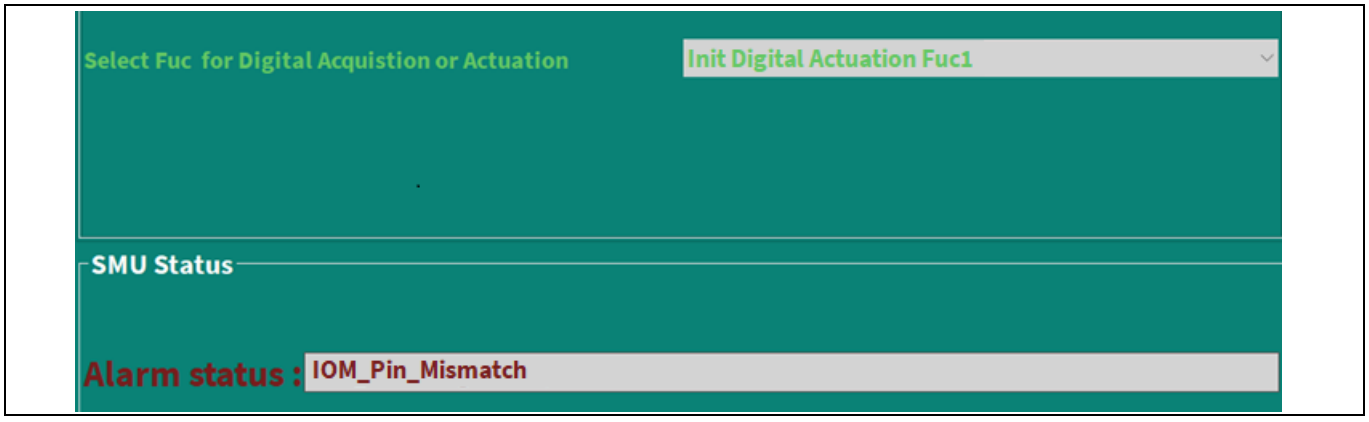

Figure 55 IOM alarm

5.5.4 Redundant TOM/TIM channels and application SW comparison (FUC2)

In the 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. A 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. *Safety Mechanism TOM_TIM_MONITORING* and *Safety Mechanism TIM_CLOCK_MONITORING* are required for this FUC.

5.5.4.1 Implementation

In order to demonstrate this FUC with the Safety kt lite, 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 SMs required for this FUC are fully implemented according to the recommendations of the AURIX™ TC3xx Safety Manual [2].
An illustrative example of the *Safety Mechanism TOM_TIM_MONITORING* and an overview about the safety lite kit implementation can be observed in Figure 56. The port pins used for this SM are as follows:

- P00.0 (TOM0_CH4, TOUT9, OUT) as S1 mission “(A)TOMa_CHb”.
- P02.3 (TIM1_CH3, IN) as S2 monitor “TIMxCHy”

For this FUC, *Safety Mechanism TIM_CLOCK_MONITORING* is also required where the GTM shall check if the GTM is clocked. Therefore, the ECLK signal shall be readback by the TIM and evaluated, if the value is within the expected timing. The port pins used for this SM are:

- P23.1 port pin
- P15.2 (TIM2_CH5) as readback signal

The pin connections for both *Safety Mechanism IOM_ALARM_CHECK* and *Safety Mechanism TIM_CLOCK_MONITORING* shall be done as shown in Figure 57.

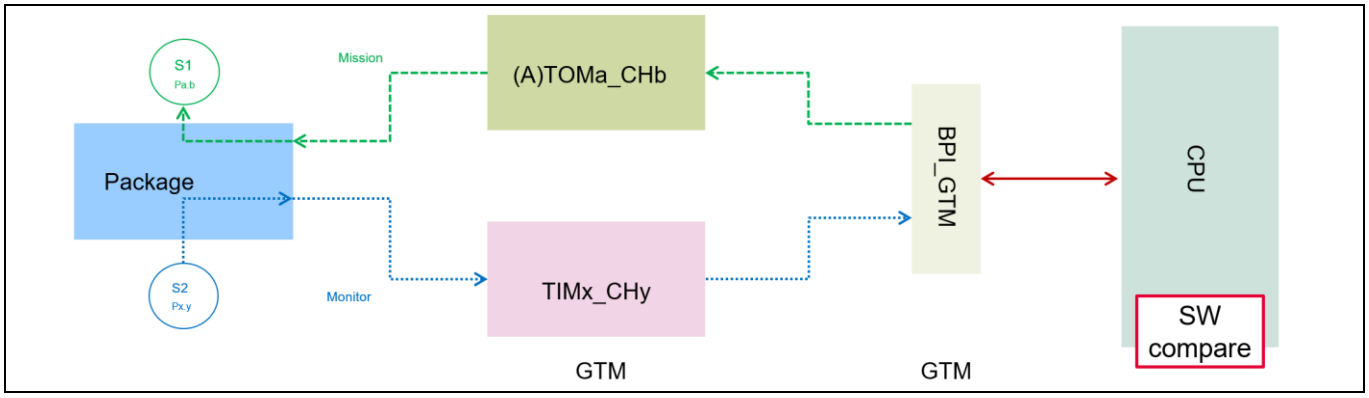

Figure 56 Illustrative example of safety mechanism TOM_TIM_MONITORING

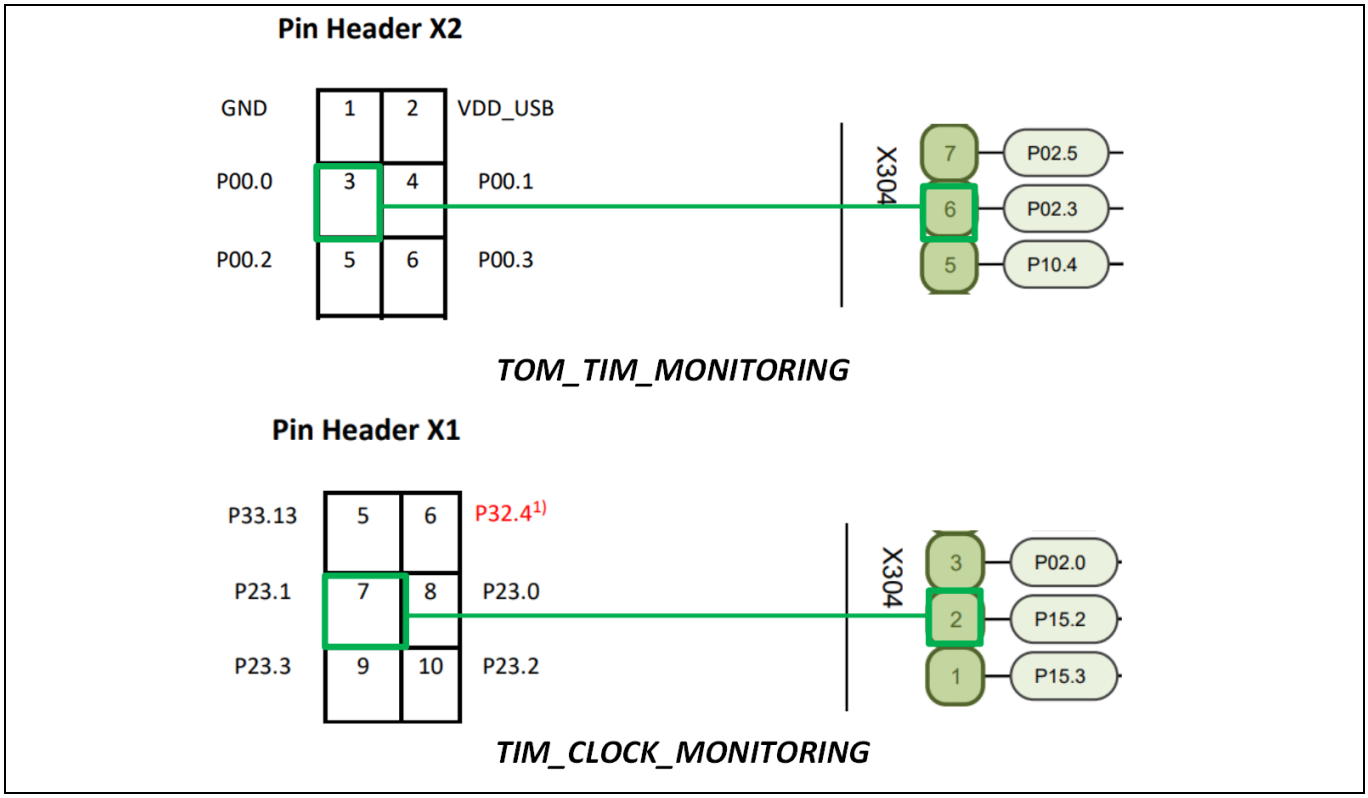

Figure 57 Pin connection for TOM_TIM_MONITORING

5.5.4.2 OneEye interface

Once the connection is done, select FUC2 in the OneEye GUI to run this functional use case. To inject an error, disconnect the PWM signal S1, the corresponding alarm will be triggered as shown in Figure 58.

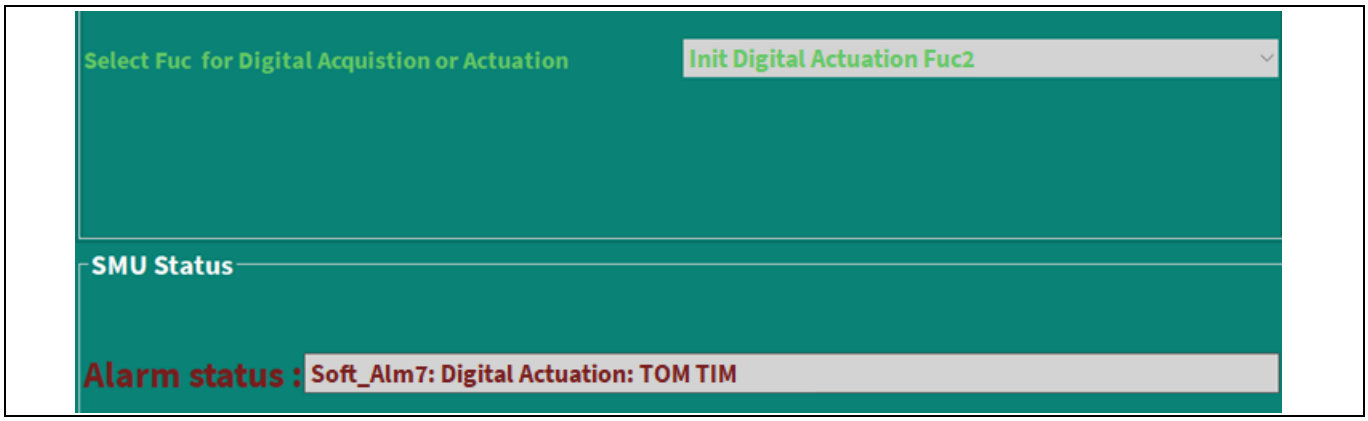

Figure 58 TOM_TIM_MONITORING alarm

5.5.5 Redundant CCU6/GPT12 channels and application SW comparison (FUC3)

In the 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. A CCU6 output resource is used to generate this signal, which 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. *Safety Mechanism CCU6_GPT12_MONITORING* is required for this FUC.

5.5.5.1 Implementation

To demonstrate this FUC with the safety lite kit, 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. This SM implementation is based on the recommendations provided in the Safety Manual [2].
An illustrative example of the *Safety Mechanism CCU6_GPT12_MONITORING* and an overview about the safety lite kit implementation of this FUC including *Safety Mechanism TIM_CLOCK_MONITORING* can be observed in Figure 59 and the pin connection shall be done as shown in Figure 60. The port pins used for this SM are as follows:

- P00.0 (CCU60_COUT63, OUT) as S1 mission CCU6_Tx”
- P33.7 (GPT12 T2INB) as S2 monitor “GPT12_Ta”

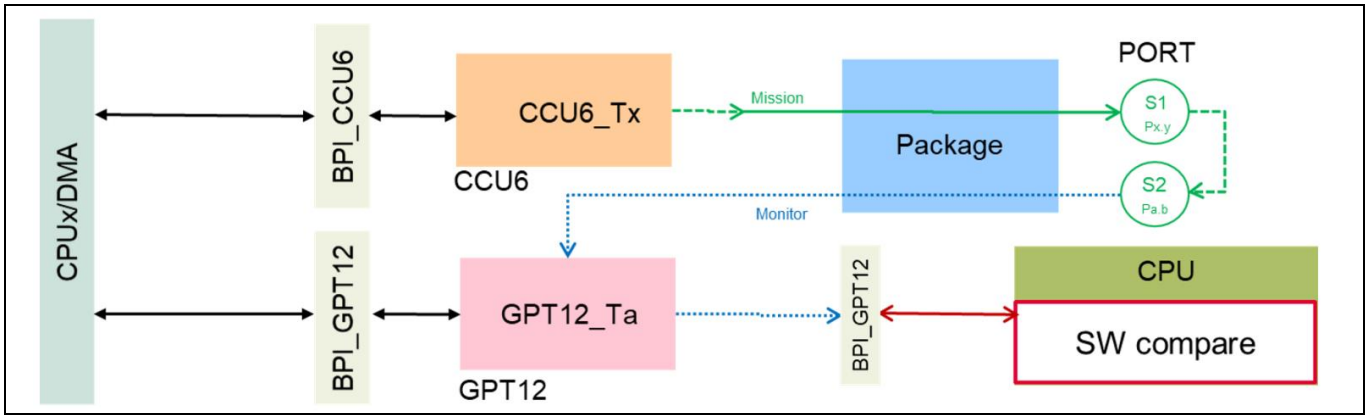

Figure 59 Illustrative example of safety mechanism CCU6_GPT12_MONITORING

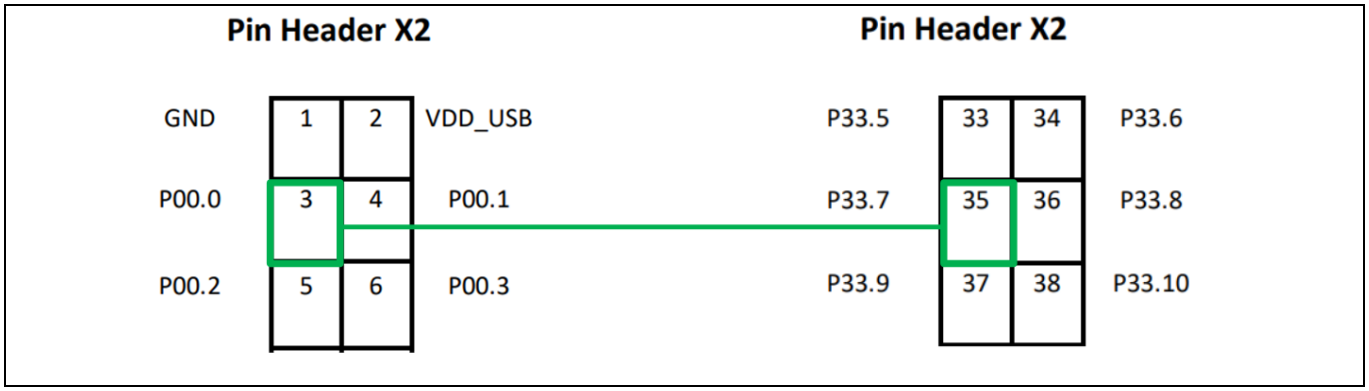

Figure 60 Pin connection for CCU6_GPT12_MONITORING

5.5.5.2 OneEye interface

Once the connection is done, select FUC3 in the OneEye GUI to run this functional use case. To inject an error, disconnect the PWM signal S1, the according alarm will be triggered, as shown in Figure 61.

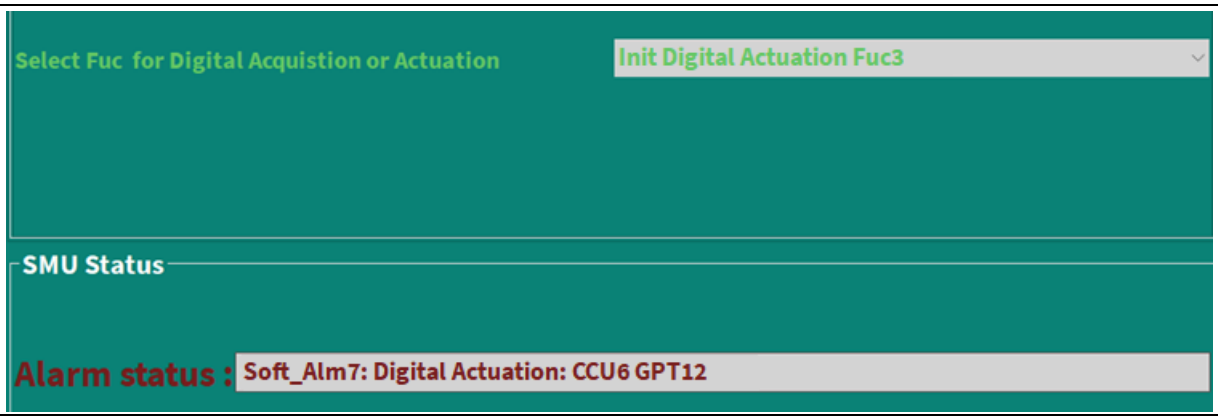

Figure 61 CCU6_GPT12_MONITORING alarm

5.6 Analog acquisition

This section presents different use-cases for the acquisition of safety-related analog input signals. For this, 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 AURIX™ TC3xx User Manual [1] for more information regarding the nominal functionality of these modules).
This application note covers five different use cases mentioned in the AURIX™ TC3xx Safety Manual. These use cases explain how to use the EVADC and the EDSADC for performing an acquisition of analog signals in different applications.
The following lists and the Table 12 present an overview of all the functional use cases mentioned in the AURIX™ TC3xx Safety Manual [2], including the type of functional blocks and SMs required for their implementation:

- 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 12 Overview about the SMs required for Analog Acquisition Functional Use Cases

Functional block	SM needed	Required for FUC(s)
CONVCTRL	• Safety Mechanism CONVCTRL:CONFIG_CHECK (verification if 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**

Functional block	SM needed	Required for FUC(s)
EVADC	• Safety Mechanism EVADC:PLAUSIBILITY (comparison of mission and monitor channels values + system-specific limit values check) • Safety Mechanism EVADC:VAREF_PLAUSIBILITY (verification of the ADC reference voltage) • Safety Mechanism EVADC:DIVERSE_REDUNDANCY *(redundancy on module level)* • Safety Mechanism EVADC:CONFIG_CHECK (verification if safety relevant register is properly configured)	• Analog Acquisition FUC**0** • Analog Acquisition FUC**2** • Analog Acquisition FUC**3** • Analog Acquisition FUC**4**
EDSADC	• Safety Mechanism EDSADC:PLAUSIBILITY • (comparison of mission and monitor channels values + system-specific limit values check) • Safety Mechanism EDSADC:VAREF_PLAUSIBILITY (verification of the ADC reference voltage) • Safety Mechanism EDSADC:DIVERSE_REDUNDANCY *(redundancy on module level)*	• Analog Acquisition FUC**1** • Analog Acquisition FUC**2** • Analog Acquisition FUC**4**

Note: *The system integrator should also consider Common Cause Failures (CCF) when selecting the redundant port pins.*

5.6.1 Overview of analog signals

Table 13 shows the analog signals which are involved in the implementation of analog acquisition.

Table 13 Overview about analog input signals and their EVADC / EDSADC functionality

ID	Signal	Input pin / symbol	EVADC / EDSADC functionality
1	Mission EVADC input	AN0	EVADC Analog input group 0, channel 0
2	Mission EVADC input	P00.2	EVADC Analog input group 9, channel 10
3	EVADC reference voltage (VAREF)	AN8	EVADC Analog input group 1, channel 0 (VDDK)
4	Mission EDSADC input	AN0	EDSADC positive analog input channel 3, pin A
5	Monitor EDSADC input	P00.2	EDSADC positive analog input channel 5, pin A
6	EDSADC reference voltage (VAREF)	AN44	EDSADC positive analog input channel 1, pin C
7	Single pin Mission EVADC input 1	AN5	EVADC Analog input group 0, channel 5
8	Single pin Monitor EVADC input 2	AN5	EVADC Analog input group 11, channel 1
9	Broken wire potentiometer EVADC input	AN0	EVADC Analog input group 0, channel 0

ID	Signal	Input pin / symbol	EVADC / EDSADC functionality
10	Broken wire potentiometer EDSADC input	AN0	EDSADC positive analog input channel 3, pin A

All functional use cases (FUC0 – FUC4) need special pin connection via a connecting cable. These pin connections are explained in each FUC, and the user must connect accordingly before enabling the particular FUC, otherwise, an alarm will be triggered.

5.6.1.1 User interface

Figure 62 shows the selection menu (in Menu 2 tab) for the analog acquisition function use case. By selecting one of the five available options, the specific FUC is initialized, and the according real-time data is displayed.

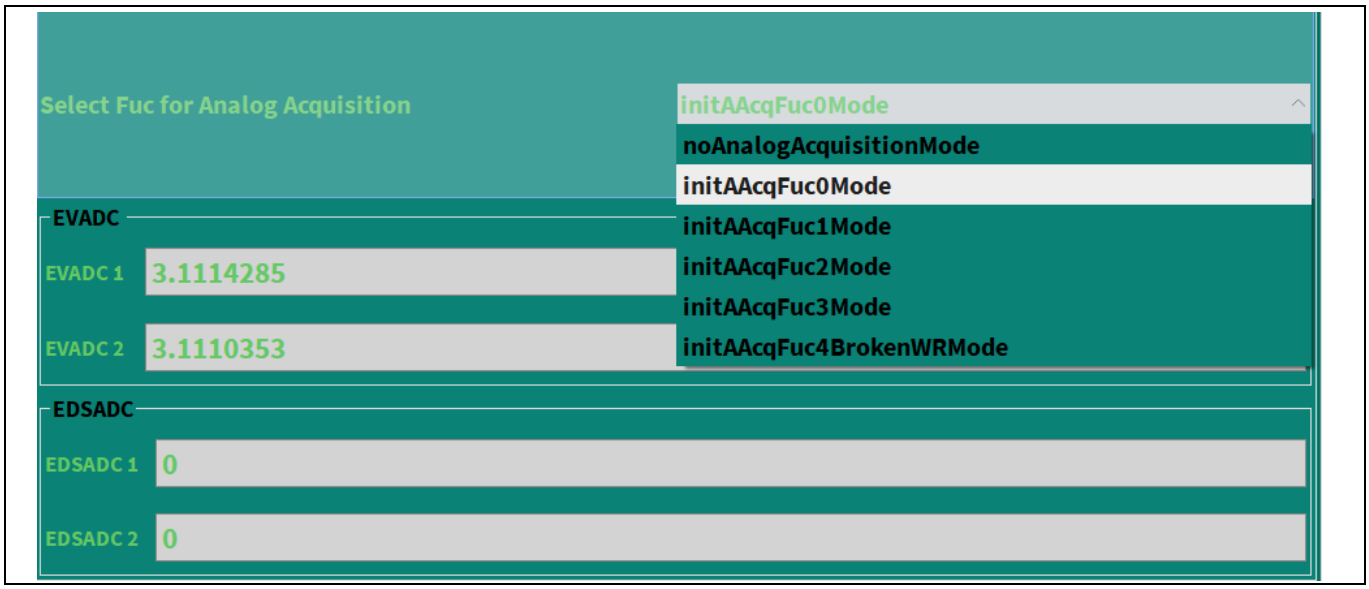

Figure 62 Analog acquisition function use cases selection

5.6.1.2 Software structure

The current implementation of safety lite kit includes FUC0, FUC1, FUC2, FUC3, and FUC4 of analog acquisition. The SMs required for each FUC are fully implemented according to the recommendation of the Safety Manual [2]. As shown in Figure 63, a global source and header file, and additional files for each use case are arranged in analog acquisition folder. 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 in software project.

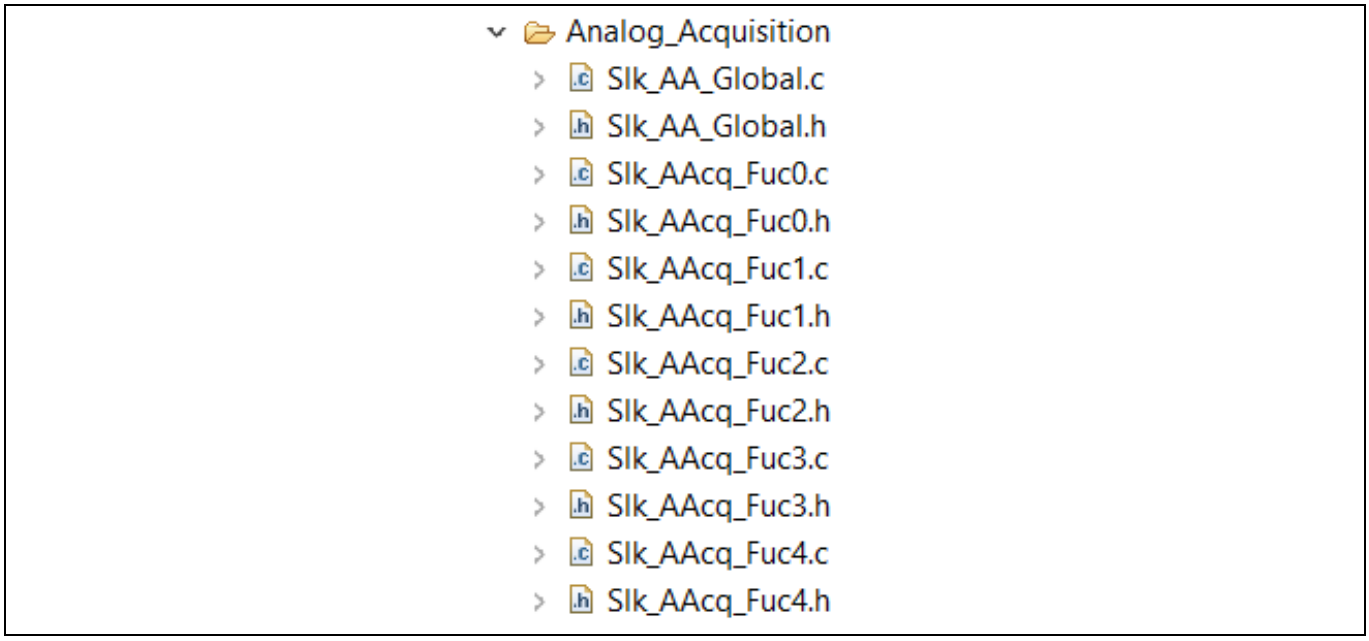

Figure 63 Analog acquisition files

5.6.2 Redundant EVADC channels (FUC0)

For FUC0, two redundant signals of 3.3 V are sampled with different channels of two EVADC modules (see signal 1 and 2 in Table 13). As specified in the AURIX™ TC3xx Safety Manual [2], an ADC reference voltage (VAREF) measurement is also required for this use case. The AURIX™ TC3xx EVADC module includes an internal diagnostic voltage signal (VDDK) provided by an internal bandgap source. The main purpose of the VDDK signal is to support the implementation of the EVADC reference supply plausibility check. The Safety Mechanism *EVADC:VAREF_PLAUSIBILITY* is implemented via DDK internal bandgap source.

5.6.2.1 Implementation

An illustrative example of the *Safety Mechanism EVADC:PLAUSIBILITY* and *EVADC:DIVERSE_REDUNDANCY* is shown in Figure 64. The two redundant signals are compared by the CPU, and an alarm is triggered if they do not meet the expected value. The pin connections required for this FUC shall be done as shown in Figure 65. The port pins used for this SM are as follows:

- AN0 (EVADC_G0_CH0, IN) as S1 mission (connected to potentiometer (0 – 3.3 V))
- P00.2 (EVADC_G9_CH10, IN) as S2 monitor (shall be connected to any 3.3 V pin)

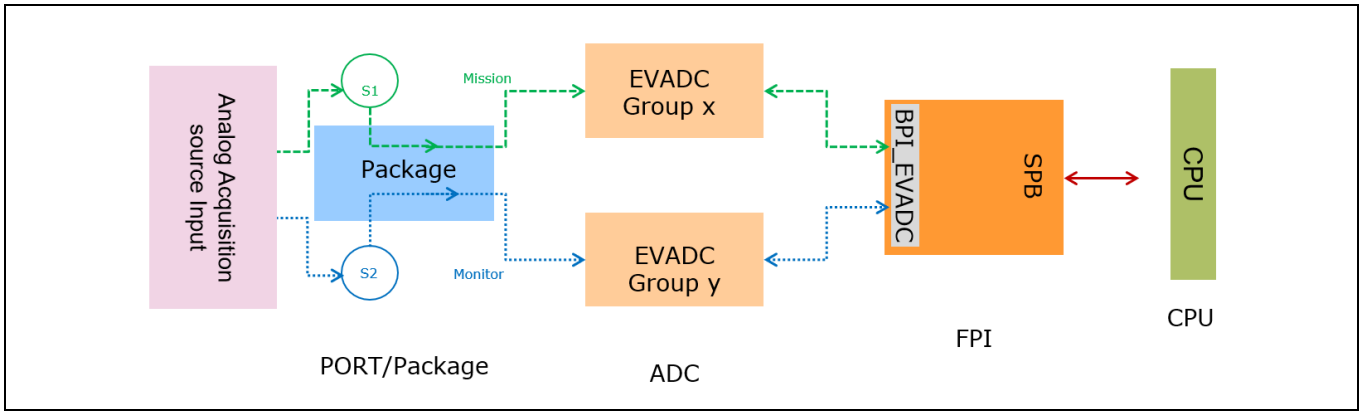

Figure 64 Illustrative example of EVADC/EVADC

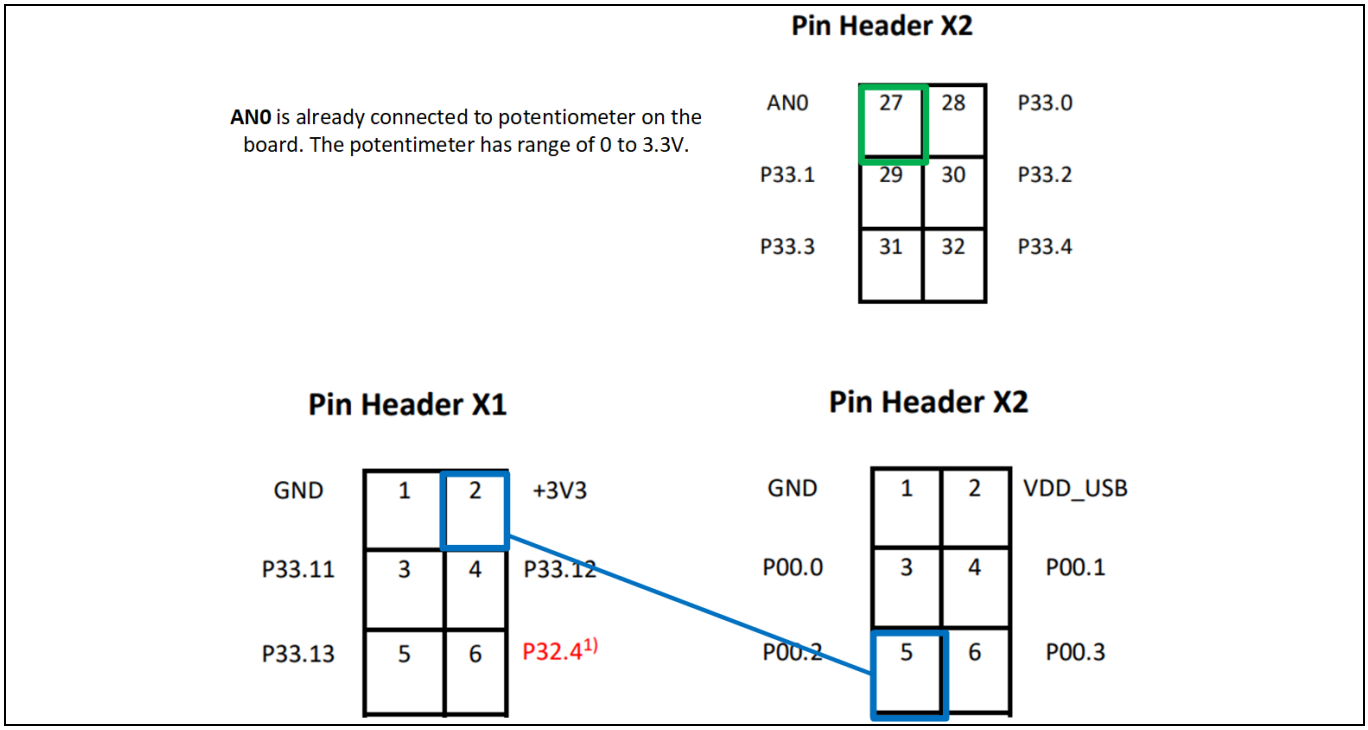

Figure 65 Pin connection for EVADC/EVADC

5.6.2.2 OneEye interface

Once the monitor pin is connected to 3.3 V and the mission pin, which is already connected to a potentiometer, shall be bought to 3.3 V by adjusting the potentiometer on the board. Then, in the OneEye GUI, select FUC0 to run this function use case. To inject an error, adjust the potentiometer to reduce the voltage; and once the limit is crossed, the corresponding alarm will be triggered, as shown in Figure 66.

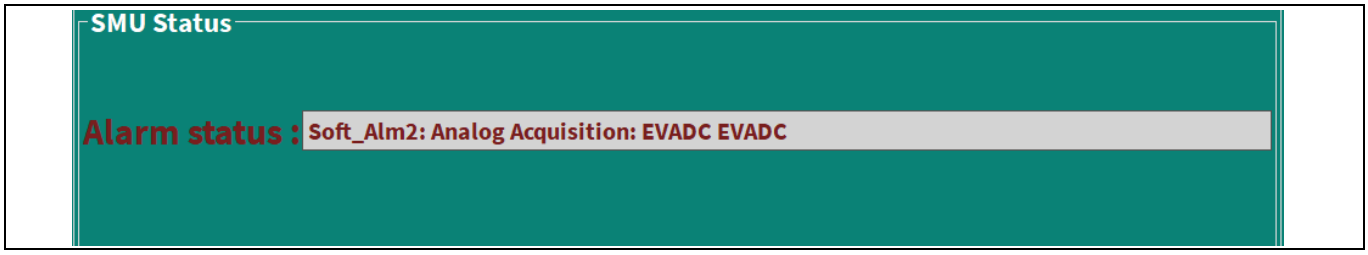

Figure 66 EVADC/EVADC alarm

5.6.3 Redundant EDSADC channels (FUC1)

The FUC1 is almost like the FUC0 but instead here the EDSADC module is used. For the FUC1 of analog acquisition, two redundant signals are sampled with different channels of two EDSADC modules (refer to signal 4 and 5 in Table 13). As also specified in the AURIX™ TC3xx Safety Manual [2], an ADC reference voltage (VAREF) measurement is also required for this use case.

5.6.3.1 Implementation

An illustrative example of the *Safety Mechanism EDSADC:PLAUSIBILITY* and *EDSADC:DIVERSE_REDUNDANCY* is shown in Figure 67 and Figure 68. The two redundant signals are compared by the CPU and trigger alarm, if it

does not meet the expected value. The pin connections required for this FUC shall be done as shown in Figure 68. The port pins used for this SM are as follows:

- AN0 (EDSADC_DS3PA, IN) as S1 mission (connected to potentiometer (0 – 3.3 V))
- P00.2 (EDSADC_DS5PA, IN) as S2 monitor (shall be connected to any 3.3 V pin)
- AN44 (ESDADC_DS1PC, IN) as Vref for reference plausibility

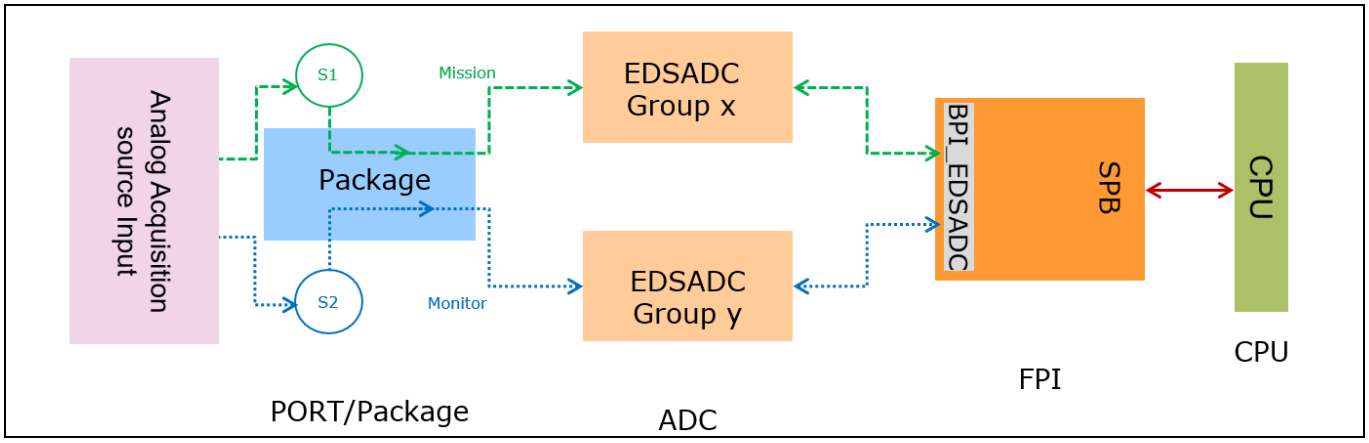

Figure 67 Illustrative example of EDSADC/EDSADC

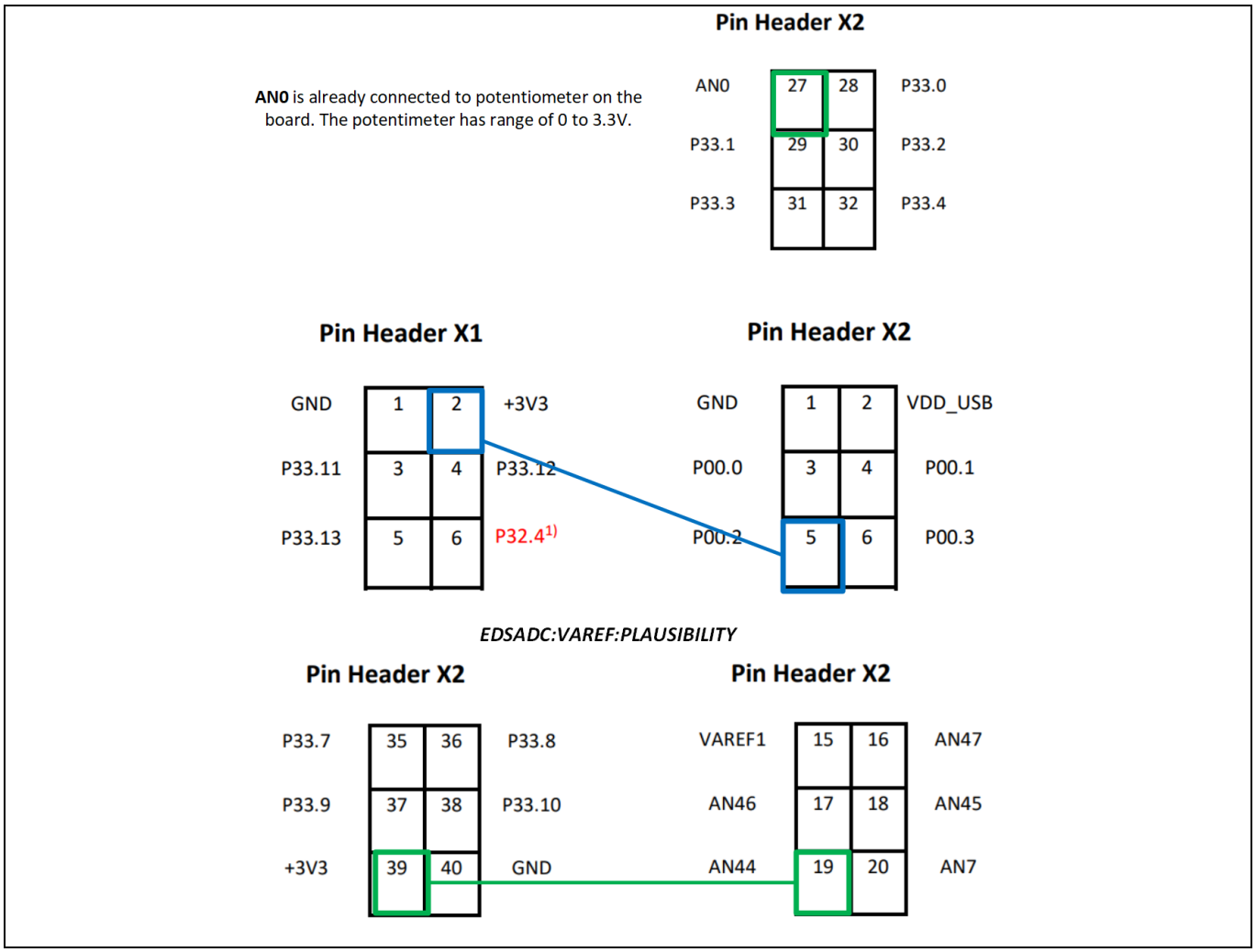

Figure 68 Pin connection for EDSADC/EDSADC

5.6.3.2 OneEye interface

Once the monitor pin is connected to 3.3 V and the mission pin which is already connected to shall bring to 3.3 V by adjusting the potentiometer on the board, then in the OneEye GUI, FUC1 need to be selected to run this function use case. To inject an error, adjust the potentiometer to reduce the voltage and once the limit is crossed, the according alarm will be triggered as shown in Figure 69.

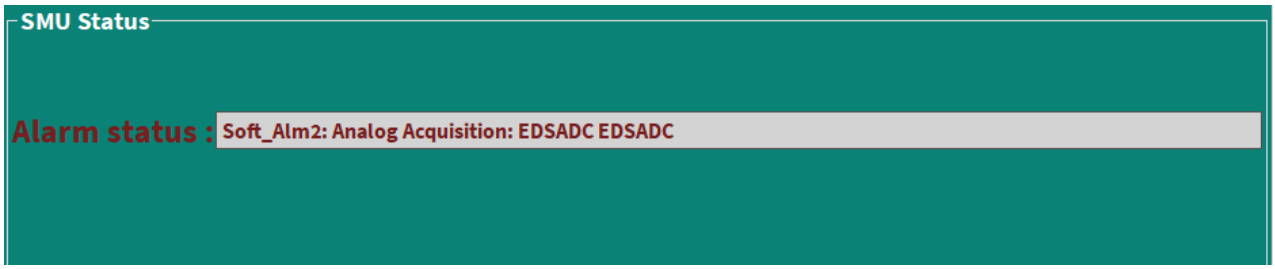

Figure 69 EDSADC/ EDSADC alarm

5.6.4 Redundant EVADC and EDSADC channels (FUC2)

For the FUC2 of analog acquisition, two redundant signals are sampled with different channels, one vi EVADC and other via EDSADC modules (refer to signal 1 and 4 in Table 13). As also specified in the AURIX™ TC3xx Safety Manual [2], an ADC reference voltage (VAREF) measurement is also required for this use case.

5.6.4.1 Implementation

An illustrative example of the *Safety Mechanism EDSADC:PLAUSIBILITY, EDSADC:DIVERSE_REDUNDANCY and Safety Mechanism EVADC:PLAUSIBILITY* and *EVADC:DIVERSE_REDUNDANCY* is shown in Figure 70. The two redundant signals are compared by the CPU and trigger alarm, if it does not meet the expected value. The pin connections required for this FUC shall be done as shown in Figure 71. The port pins used for this SM are as follows:

- AN0 (EVADC_G0_CH0, IN) as S1 mission (connected to potentiometer i.e. 0 to 3.3 V)
- EVADC Varef plausibility is done bandgap method (VDDK)
- P00.2 (EDSADC_DS5PA, IN) as S2 monitor (shall be connected to any 3.3 V pin)
- AN44 (ESDADC_DS1PC, IN) as Vref for reference plausibility

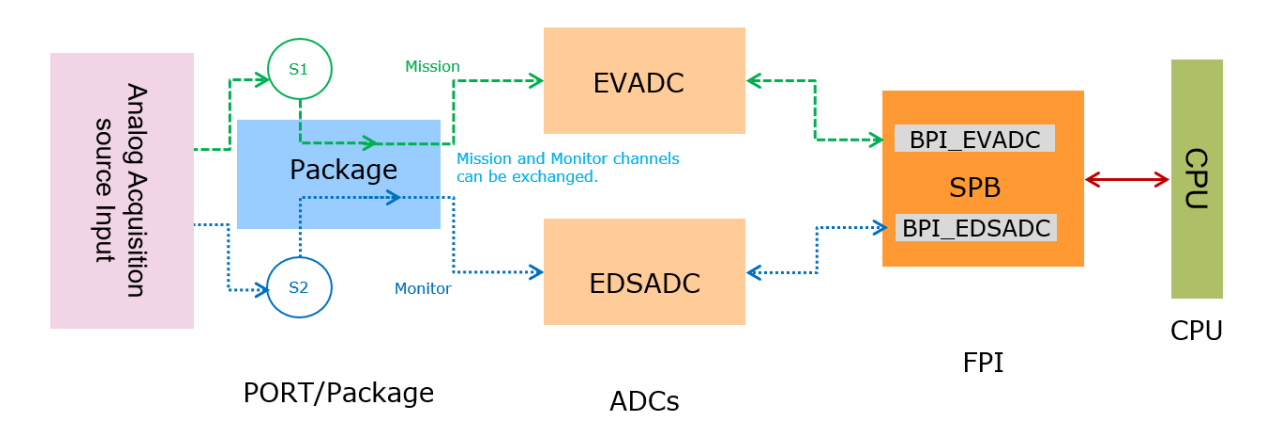

Figure 70 Illustrative example of EVADC/EDSADC

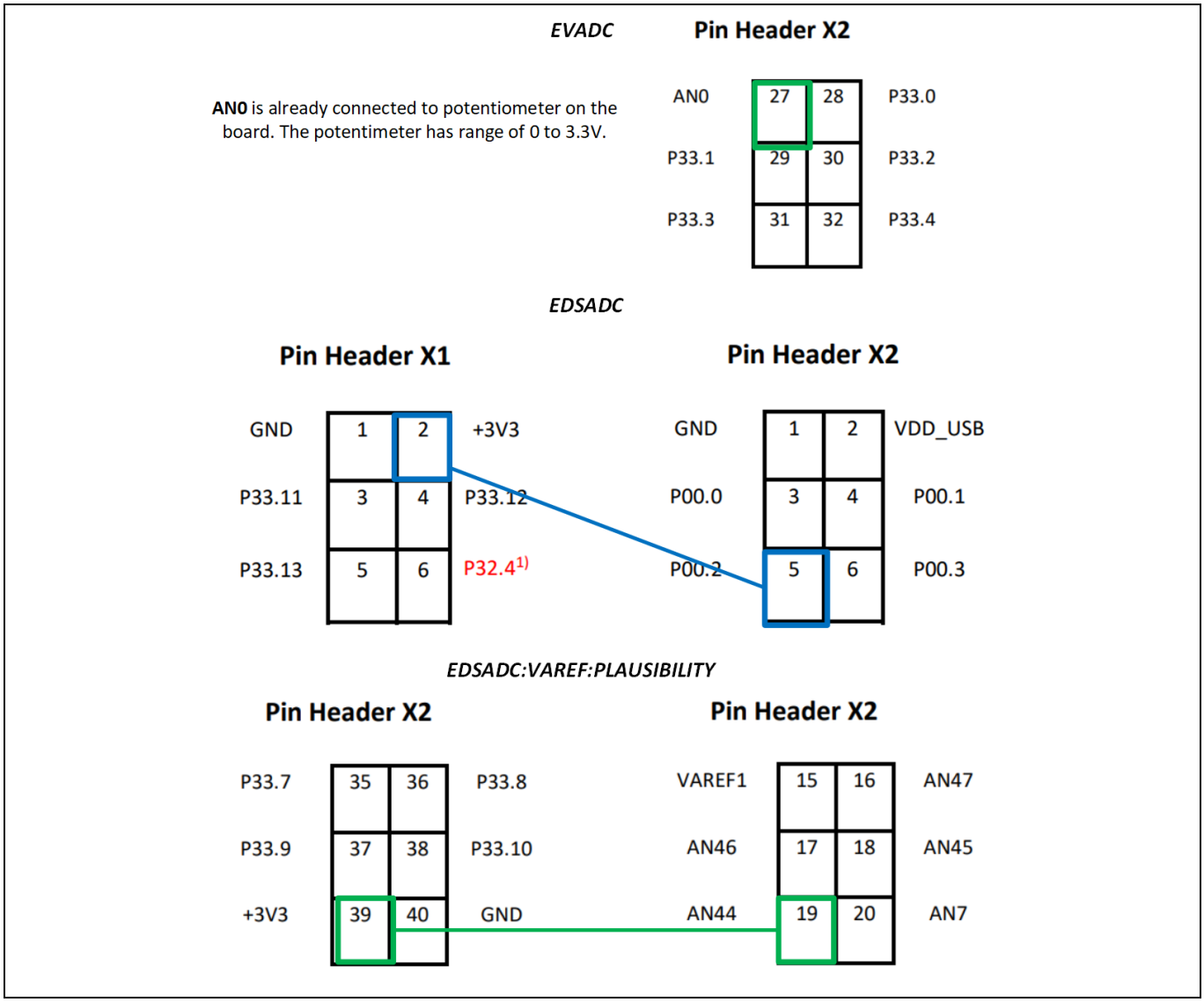

Figure 71 Pin connection for EVADC/EDSADC

5.6.4.2 OneEye interface

Once the monitor pin is connected to 3.3 V and the mission pin, which is already connected to the potentiometer, shall be brought to 3.3 V by adjusting the potentiometer on the board. Then, in the OneEye GUI, select FUC1 to run this function use case. To inject an error, adjust the potentiometer to reduce the voltage; and once the limit is crossed, the corresponding alarm will be triggered, as shown in Figure 72.



Figure 72 EVADC/EDSADC alarm

5.6.5 Single pin with EVADC channels (FUC3)

For the FUC3 of analog acquisition, a safety-related signal is received on a single analog pin and then it is redundantly processed by the two different internal resources of the EVADC module (refer to signal 7 and 8 in Table 13). As specified in the AURIX™ TC3xx Safety Manual [2], an ADC reference voltage (VAREF) measurement is also required for this use case.

5.6.5.1 Implementation

An illustrative example of the *Safety Mechanism EVADC:PLAUSIBILITY* and *EVADC:DIVERSE_REDUNDANCY* is shown in Figure 73. The two redundant signals are compared by the CPU, and an alarm is triggered if they does not meet the expected value. The pin connections required for this FUC shall be done as shown in Figure 74. The port pins used for this SM are as follows:

- AN5 (EVADC_G0_CH5, IN) as S1 mission (shall be connected to any 3.3 V pin)
- AN5 (EVADC_G11_CH1, IN) as S2 monitor (shall be connected to any 3.3V pin)
- EVADC Varef plausibility is done via bandgap method (VDDK)

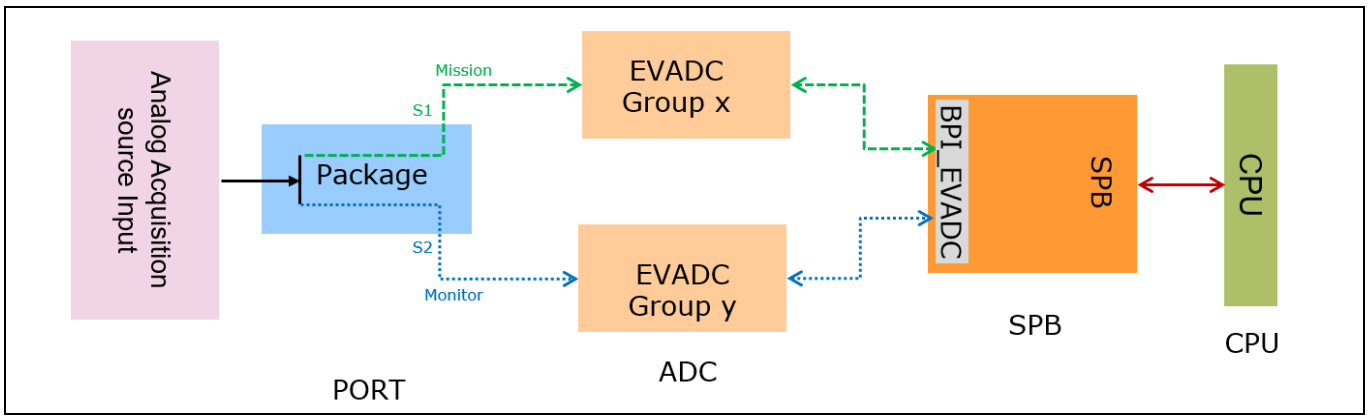

Figure 73 Illustrative example of single pin EVADC/EVADC

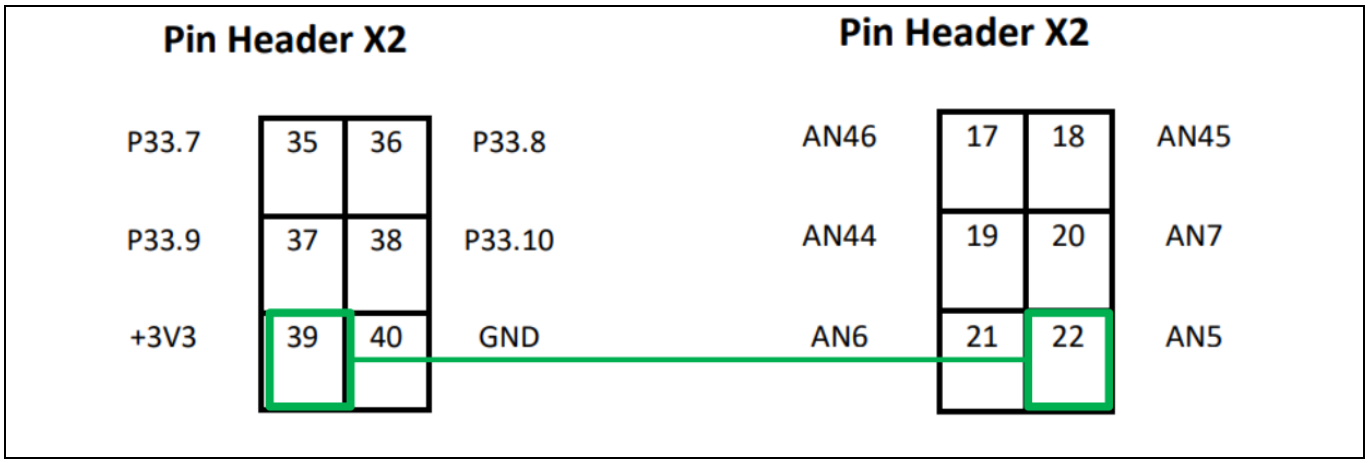

Figure 74 Pin connection for single pin EVADC

5.6.5.2 OneEye interface

Once the monitor pin is connected to 3.3 V and the mission pin, which is already connected to the potentiometer, shall be brought to 3.3 V by adjusting the potentiometer on the board. Then, in the OneEye GUI,

select FUC1 to run this function use case. To inject an error, adjust the potentiometer to reduce the voltage; and once the limit is crossed, the corresponding alarm will be triggered, as shown in Figure 75.

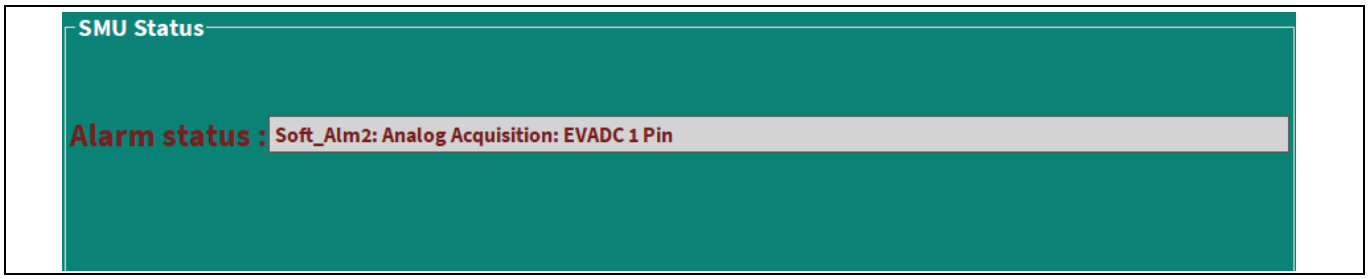

Figure 75 Single pin EVADC alarm

5.6.6 Single pin with EVADC and EDSADC channels and broken Wire (FUC4)

For the FUC4 of analog acquisition, safety-related signal is received on a single analog pin and then it is redundantly processed by two different internal resources of EVADC and EDSADC module (refer to signal 9 and 10 in Table 13). As also specified in the AURIX™ TC3xx Safety Manual [2], an ADC reference voltage (VAREF) measurement is also required for this use case.

5.6.6.1 Implementation

An illustrative example of the *Safety Mechanism EDSADC:PLAUSIBILITY, EDSADC:DIVERSE_REDUNDANCY and Safety Mechanism EVADC:PLAUSIBILITY,* and *EVADC:DIVERSE_REDUNDANCY* is shown in Figure 76. The two redundant signals are compared by the CPU, and an alarm is triggered if they do not meet the expected value. The pin connections required for this FUC shall be done as shown in Figure 77. The port pins used for this SM are as follows:

- AN0 (EVADC_G0_CH0, IN) as S1 mission (connected to potentiometer, i.e.0 – 3.3 V)
- EVADC Varef plausibility is done bandgap method (VDDK)
- PAN0 (EDSADC_DS3PA, IN) as S2 monitor (connected to potentiometer, i.e.0 – 3.3 V)
- AN44 (ESDADC_DS1PC, IN) as Vref for reference plausibility

This example also includes a demonstration of the broken-wire detection hardware safety features of the EVADC and hardware configuration of the limit check for *Safety Mechanism EVADC:PLAUSIBILITY*. This demo also utilizes the potentiometer X4 to generate the captured analog input signal. When the potentiometer is completely move in the clockwise direction, it will reach the GND.

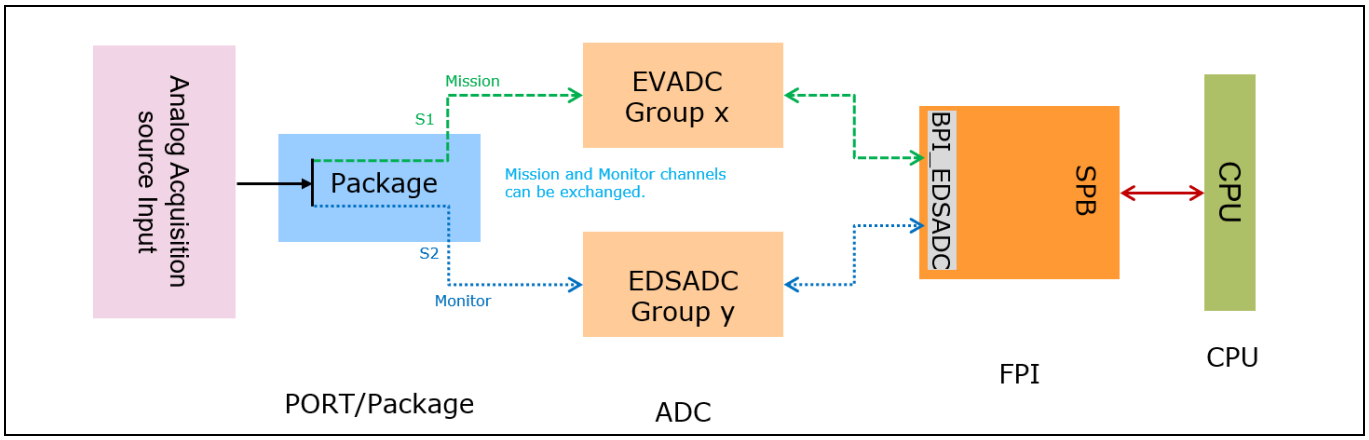

Figure 76 Illustrative example of single pin EVADC EDSADC

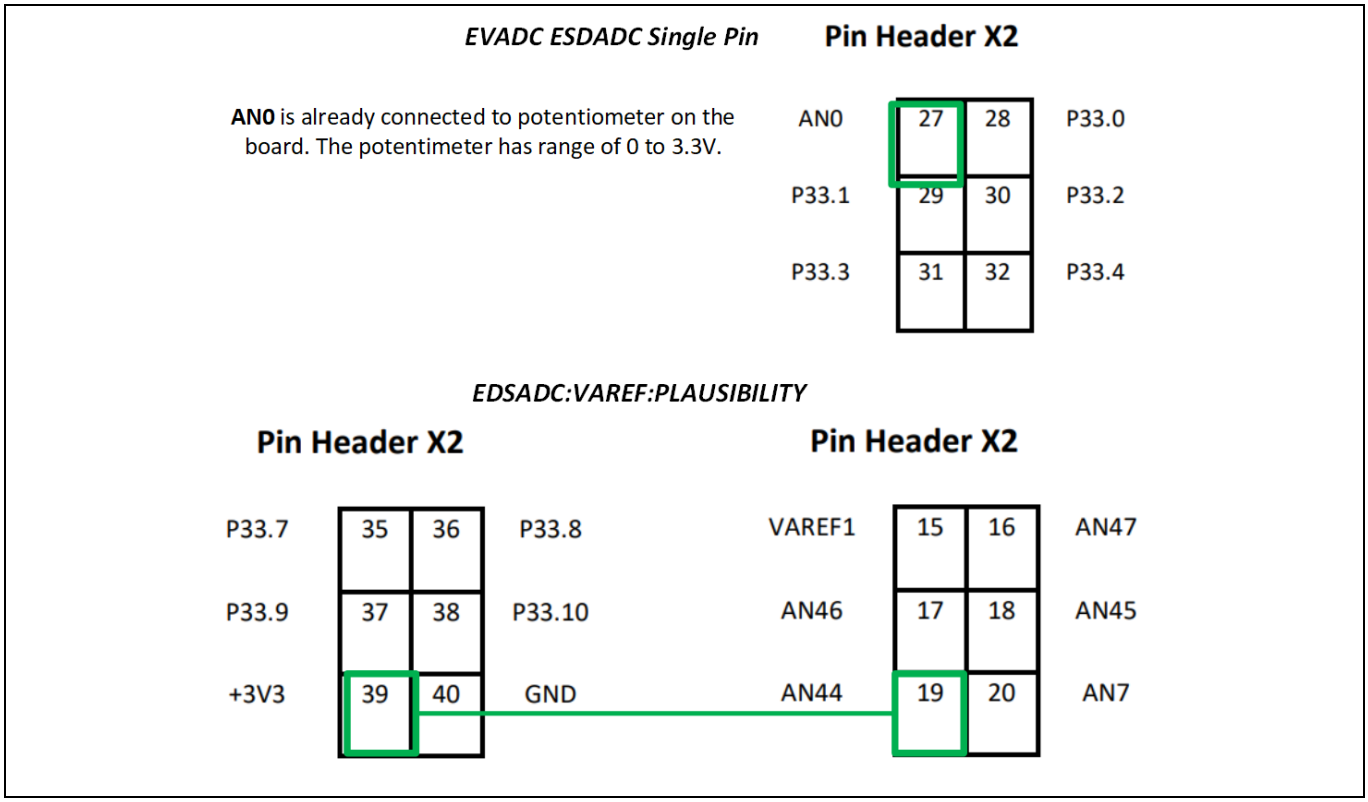

Figure 77 Pin connection for single pin EVADC EDSADC

5.6.6.2 OneEye interface

Once the monitor pin is connected to 3.3 V and the mission pin, which is already connected to the potentiometer, shall be brought to 3.3 V by adjusting the potentiometer on the board. Then, in the OneEye GUI, select to run this function use case. To inject an error, adjust the potentiometer to reduce the voltage; and once the limit is crossed, the corresponding alarm will be triggered, as shown in Figure 78.

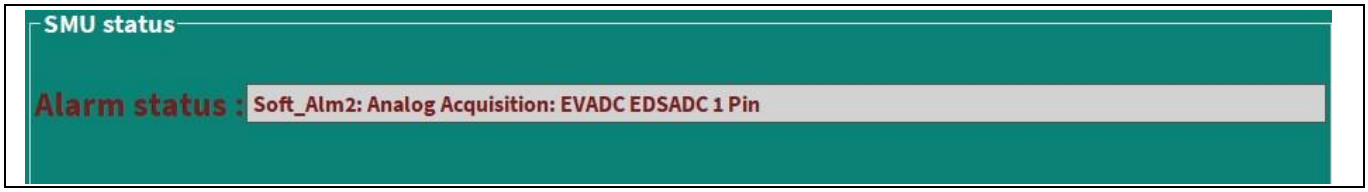

Figure 78 Single pin EVADC EDSADC alarm

5.7 Avoidance or detection of CCF

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

To avoid CCF, it is recommended that the mission and monitor signals shall have at least a distance of 3 pins on the package. This distance between the mission and the monitor signals is to avoid a possible short-circuit on the leadframe or between bond wires in the package. This distance also ensures that shorts between

neighboring pins at the package outside do not cause a common cause failure for the mission and the monitor signals.

5.7.1 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.

5.7.1.1 Implementation

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

The PORT:LOOPBACK safety mechanism is implemented as shown in Figure 79, where S1 as mission signal on Px.y (P33.5) works as the output and is given to the outer world. The same signal is looped back to the S2 monitor signal and fed back to Pa.b (P00.2) pin as the input. The application SW then compares both signals and responds accordingly.

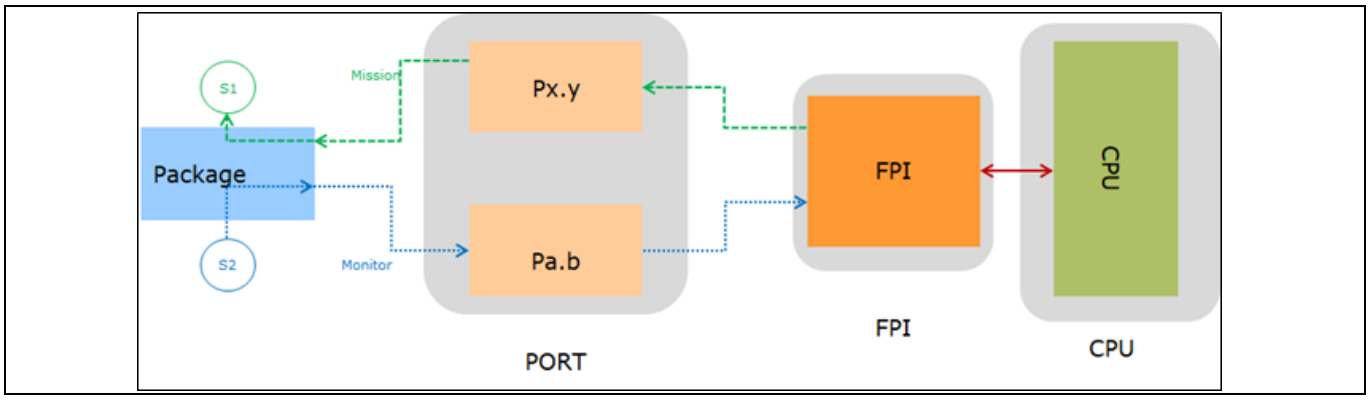

Figure 79 Safety Mechanism PORT:LOOPBACK overview

The *Safety Mechanism PORT:REDUNDANCY* is also implemented as shown in Figure 80 in the Safety lite kit. The application SW receives a safety-critical input value (called signal S1 “Mission”) on pin Px.y (P00.8) from an external sender. 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:

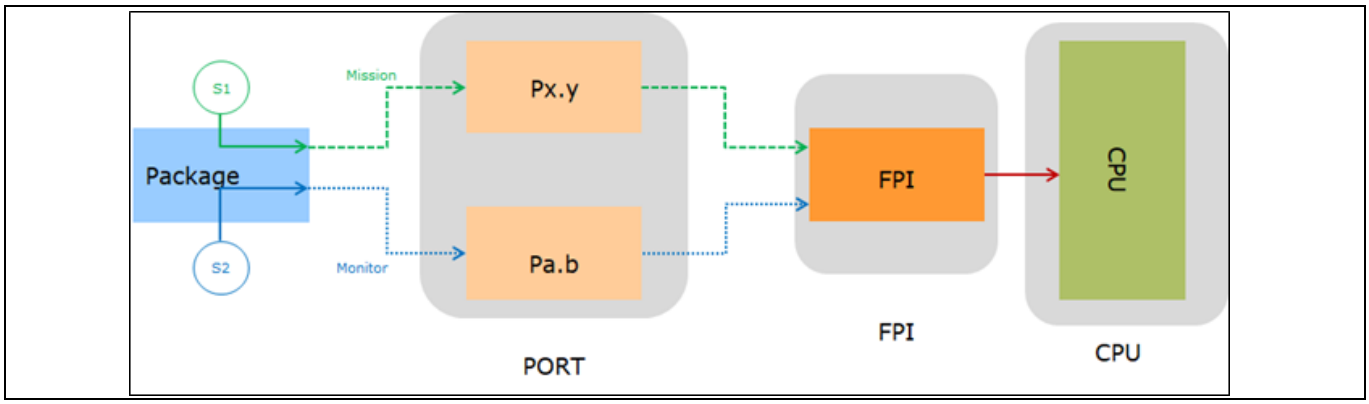

Figure 80 Safety Mechanism PORT:REDUNDANCY overview

Figure 81 shows the pin connection that needs to be done on the board before enabling th alarm for these two SMs.

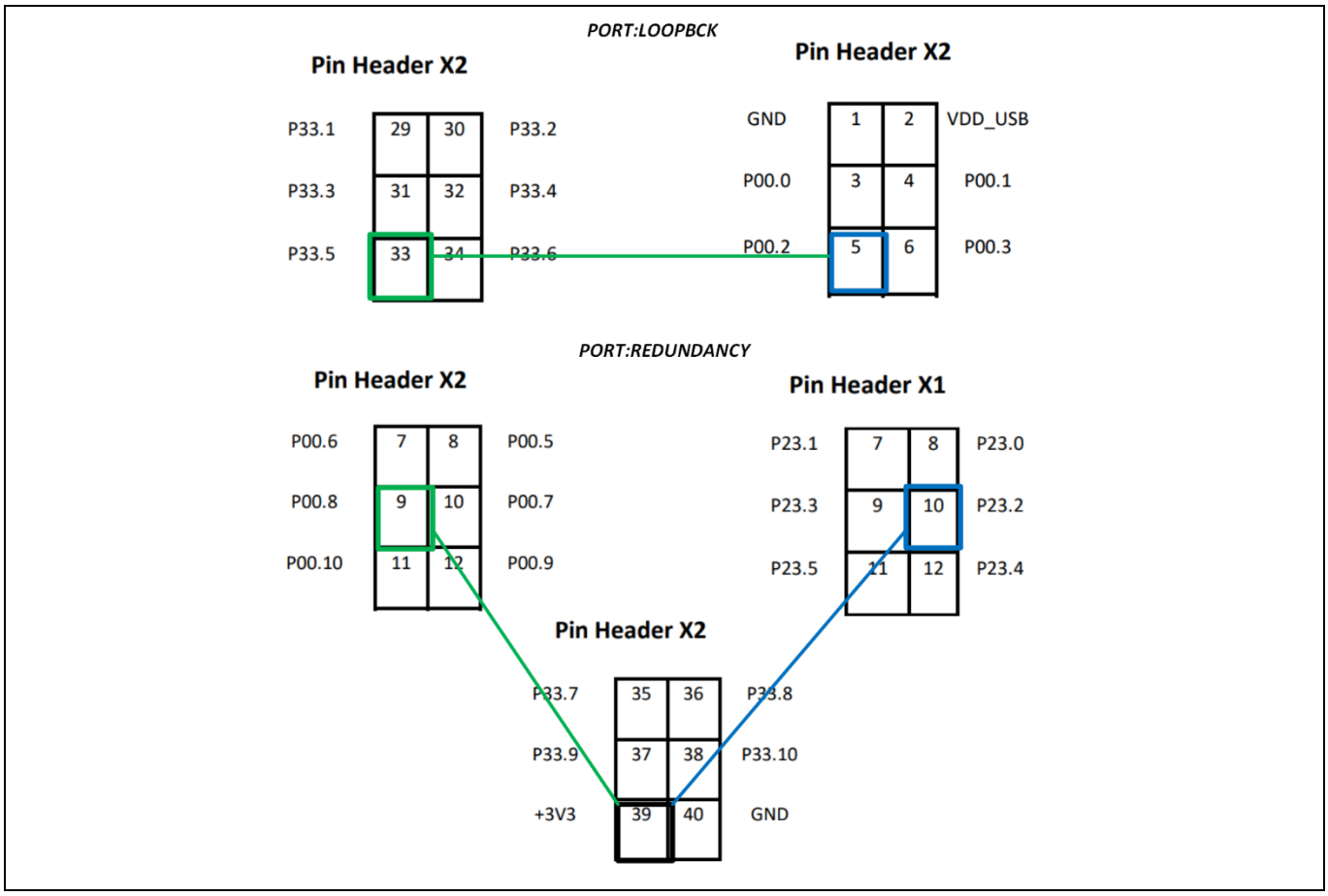

Figure 81 PORT loopback and redundancy pin connection

5.7.1.2 OneEye interface

One the connection is done, enable both Port loopback and redundancy alarm in the OneEye GUI, as shown in Figure 82, to see if there is any mismatch occur for any of the SMs. By default, the alarms are not enabled, so no alarm is triggered if there is no connection or mismatch.



Figure 82 Enable alarm for PORT loopback and redundancy

Once the pin connection is established and alarms are enabled as discuss above, to inject an error, disconnect any of the signal, the alarm will be generated as shown in Figure 83 .



Figure 83 Alarm for PORT loopback and redundancy

5.7.2 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].

5.7.2.1 Implementation

Safety Lite Kit contains the implementation of the *Safety Mechanism CLOCK:PLAUSIBILITY* (Slk_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.

The QSPI3 module is used to implement this SM. The following steps are used:

- Initialize the QSPI3 module
  - Master mode
  - XXL mode
  - DMA enabled
  - Baud rate of 10 MHz
  - Enabled phase transition (PT) interrupt
- Fill data into a transmit buffer of size 125 bytes
- Start the QSPI master data transfer
- Capture the time via the STM timer on two PT interrupts
  - First Interrupt will occur at the start of the frame (SoF)
  - Another interrupt will occur at the end of the frame (EoF)
- Calculate the different EoF time and SoF time taken via the STM timer
- Compare the resulting value with the range of theoretical values limit and respond accordingly

5.7.2.2 OneEye interface

In the OneEye GUI, *Safety Mechanism CLOCK:PLAUSIBILITY* real-time data is displayed, see Figure 84. This data includes the QSPI data one frame transfer time. If this time falls below the lower limit or exceeds the upper limit value, an alarm will be triggered. Notably, no pin connection is required, and no error injection possibility is given for the SMs.

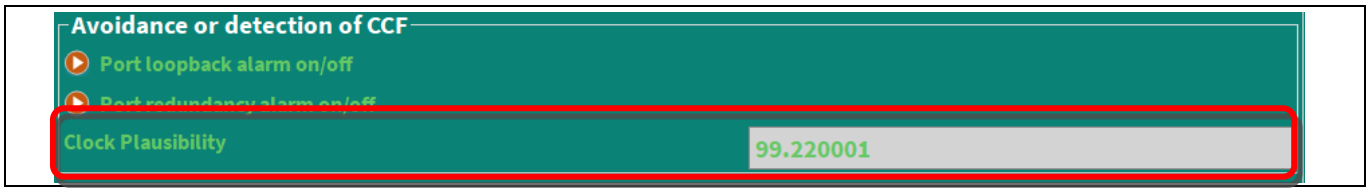

Figure 84 Clock Plausibility real time data

5.7.3 Die temperature sensors (DTS)

A 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.

5.7.3.1 Implementation

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.

Both temperatures are compared to each other. If the difference exceeds 9°C, an SMU software alarm will be triggered (see *Slk_DieTemp.* * files in the source code)

5.7.3.2 OneEye interface

In the OneEye GUI, real-time temperature values of Die and Core are displayed (Figure 85). When the temperature difference between the two varies above 9.0 °C, the corresponding alarm will be generated. For this, no pin connection is required, and no error injection possibility is given for the SM.

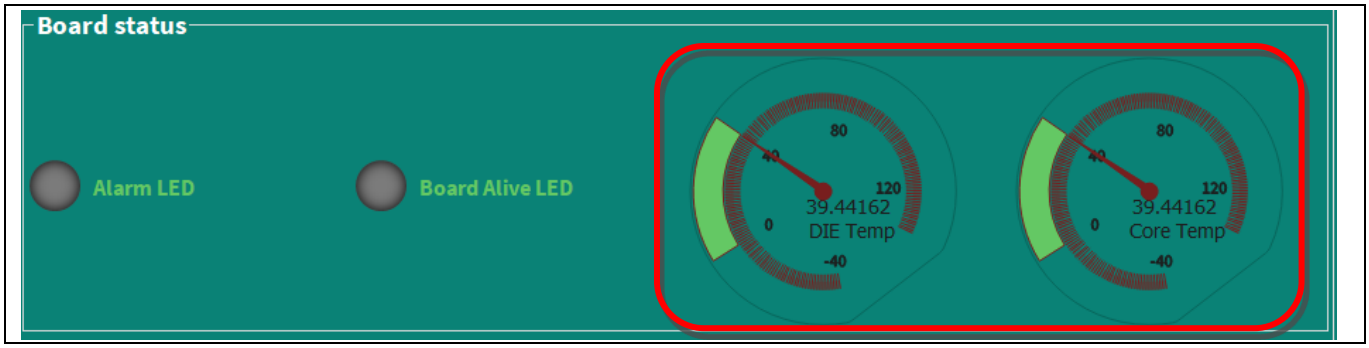

Figure 85 Die and Core temperature value display

5.8 External communication

5.8.1 QSPI

The main purpose of the Queued Synchronous Peripheral Interface (QSPI) module is to provide a synchronous serial communication with the 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 more information on 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 the CRC, which is based on the payload data of the 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.

5.8.1.1 Implementation

In safety lite kit, to implement this SM, it is required to connect two different QSPI instances with each other using the loopback concept. The QSPI1 instance works as a master, while the QSPI2 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 86.

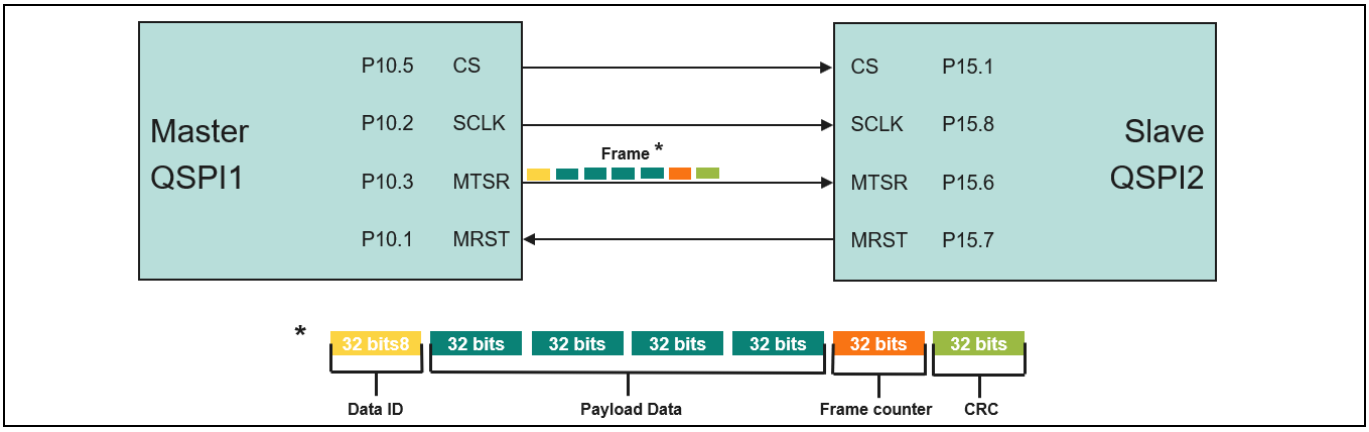

Figure 86 QSPI communication protocol

Figure 87 shows the pin connection that needs to be done on the board before enabling the alarm for this safety mechanism.

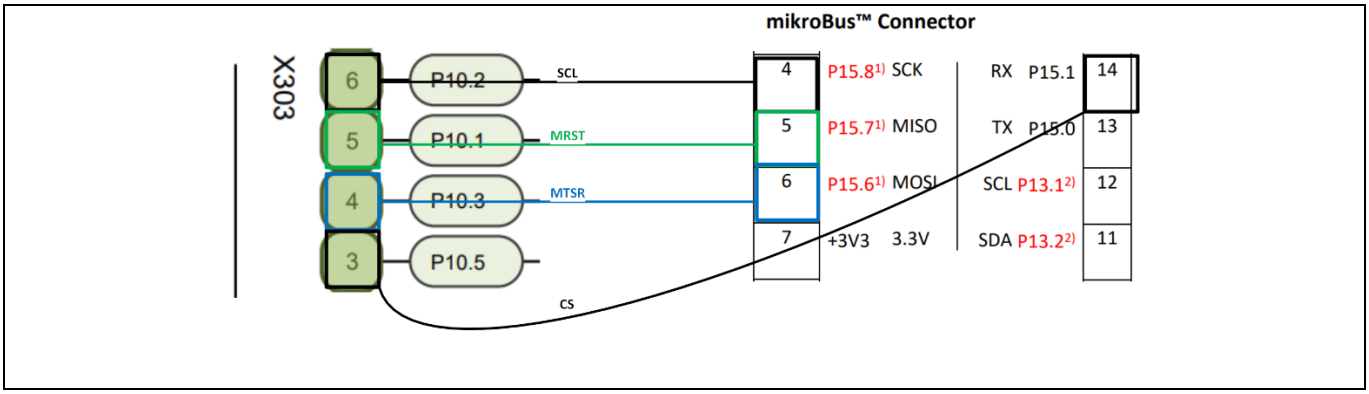

Figure 87 QSPI pin connection

The frame size for this implementation is seven 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 value. The CRC value is calculated based on the payload data. To calculate the CRC value, the TriCore™ CPU instruction CRC32 is used, and the run time CRC calculation is performed on both master and slave side.

The master device runs on CPU0 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 *Slk_Main.c*).

On the slave side, running on CPU1, a slave Rx interrupt occurs each time when a 32-bit data is recieved. When the complete frame sent by the master is received, then the communication safety measures (data and cycle time validation) related to the frame are executed. In case of any errors, such as mismatching CRC or frame counter (FC) and frame cycle time, the system raises an alarm (see function `validateSlavedataAndCycleTime in Slk_Slave_QSPI_Safe_Communication.c` *file*). Then the slave is configured to receive the next frame.

The detection of incorrect timeout is managed by another independent safety measure that is triggered every 100 ms (see the `timeoutSlaveQspiSafeCommunication` function in `Slk_Main.c`).

5.8.1.2 OneEye interface

Once the connection is done, in the OneEye GUI, QSPI safe communication alarm need to be enabled (play icon) to check if there is any mismatch occur for any of the SMs (see Figure 88). By default, the alarms are not enabled, so no alarm is triggered in case of no connection or mismatch.

An error can be injected either by disconnecting the cable connection or via the OneEye GUI by selecting either early frame or late frame error, as shown in red highlighted in Figure 88.

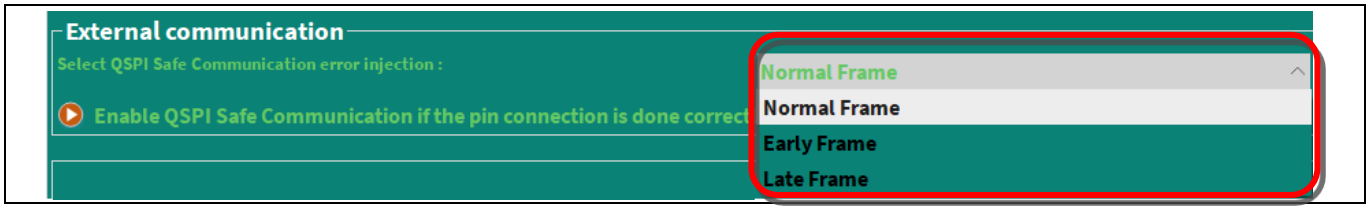

Figure 88 Enable alarm for QSPI safe communication

Figure 89 shows different alarm messages in case any error has occurred.

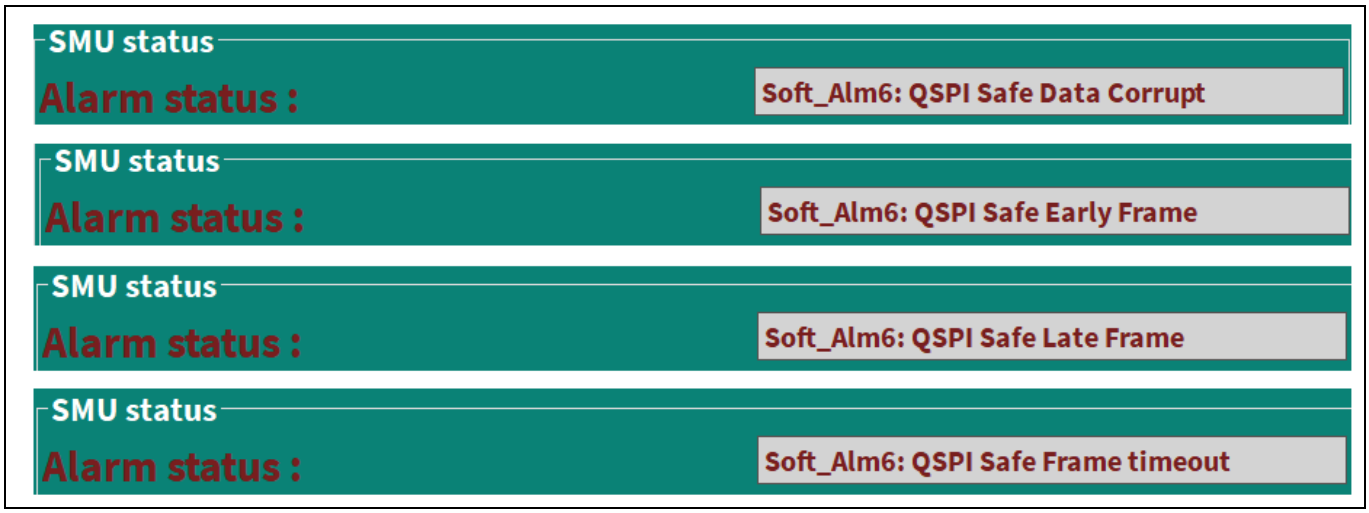

Figure 89 QSPI safe communication different alarm

5.9 Coexistence of HW/SW elements

5.9.1 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 more details on this module, see Chapter 16 of the AURIX™ TC3xx User Manual [1].

5.9.1.1 Implementation

Safety Lite Kit 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 the plausibility of this interrupt, i.e., if an expected interrupt is missing or unintended interrupts occur.

5.9.1.2 OneEye interface

There is neither a pin connection needed nor any possibility to inject the error for this SM as it is available. The error can be injected once you are in a debugging state. For this, apply a break point anywhere in the source code and then run the code. Once the break point is hit, resume the code, and the following alarm will popup (see Figure 90) as there will be a mismatch in interrupt occurrence.



Figure 90 ISR alarm

References

Contact Infineon Support to obtain these documents:

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

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

[3] Infineon Technologies AG: *AURIX™ TC37x-User's Manual Appendix V2.0.0*; Available online

[4] Infineon Technologies AG: *AURIX™ TC3xx Watchdog_1 for KIT_AURIX_TC375_LK V1.0.1*; Available online

[5] Infineon Technologies AG: *KIT_A2G_TC375_LITE*; Available online

Revision history

Document revision	Date	Description of changes
V 1.0	2025-03-17	Initial release

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

Edition 2025-03-17

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2025 Infineon Technologies AG. All Rights Reserved.

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

AN0029

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.

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.