Freescale Semiconductor
Safety Manual

Document Number: MPC5675KSM
Rev. 1, 12/2012

Safety Manual for Qorivva MPC567xK

Devices Supported:

MPC5673K
MPC5674K
MPC5675K

Table of Contents

List of Figures

List of Tables

1 Preface

This document discusses requirements for the use of the MPC567xK Microcontroller Unit (MCU) in functional safety relevant applications requiring high functional safety integrity levels.

It is intended to support system and software engineers using the MPC567xK available features as well as achieving additional diagnostic coverage by software measures.

This document shall not imply conformance of the MPC567xK according to ISO 26262 or IEC 61508. Although the MPC567xK architecture is in large part based on an MCU which was certified, there are architectural differences, and the development of the MPC567xK was not assessed according to ISO 26262 or IEC 61508.

Several measures are prescribed as safety requirements whereby the measure described was assumed to be in place when analyzing the functional safety of this Microcontroller Unit (MCU). In this sense, requirements in the Safety Manual (SM) are driven by assumptions concerning the functional safety of the system that will integrate the MPC567xK in accordance with the section “Safety Element out of Context” in ISO 26262-10.

Example: **Safety requirement:** Before executing any safety function, the FMPLLs requires configuration to use the external oscillator (XOSC) as their source clock.

Example: **Safety requirement under certain preconditions:** If the system requires robustness regarding catastrophic CMFs (such as, issues with the MPC567xK external power supply) and robustness regarding systematic faults, system level measures are to be implemented.

NOTE

Requirements (or requirements under certain preconditions) are marked by a tag of the form “SM_*nnn*” at the beginning of the requirement, and are terminated with an “end”. Both of these tags are enclosed within square brackets for easy recognition. These tags could be used to allow importing the requirements into safety traceability management tools.

For the use of the MCU this means that if a specific safety manual requirement is not fulfilled, it has to be rationalized that an alternative implementation is at least similarly efficient concerning the functional safety requirement in question (for example, provides same coverage, reduces the likelihood of Common Mode Failure (CMF) similarly well, and so on) or the estimation of an increased failure rate (λ_{SPF}, λ_{RF}, λ_{MPF}, λ_{DU} …) and reduced metrics (SFF: Safe Failure Fraction, SPFM: Single-Point Fault Metrics, LFM: Latent Fault Metric) due to the deviation has be specified.

This document also contains guidelines on how to configure and operate the MPC567xK for functional safety relevant applications requiring high functional safety integrity levels. These guidelines are preceded by one of the following text statements:

- Implementation hint
- Recommended
- Example
- Safety requirement
- Safety requirement under certain preconditions

- Rationale

These guidelines are considered to be useful approaches for the specific topics under discussion. The user will need to use discretion in deciding whether these measures are appropriate for their applications.

This document is valid only under the assumption that the MCU is used in functional safety applications requiring a fail-silent or a fail-indicate MCU. A fail-operational mode of the MPC567xK is not described.

This document is targeting high functional safety integrity levels. For functional safety goals which do not require high functional safety integrity levels, system integrators will need to tailor the requirements for their specific application.

It is assumed, that the user of this document is in general familiar with the MPC567xK device, ISO 26262, and IEC 61508 standards.

1.1 Related documents

This sections lists all the documentation mentioned in this Safety Manual:

- IEC 61508: IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems, international standard, ed. 2.0, April 2010
- ISO 26262: ISO 26262 Road vehicles - Functional safety, November 2011
- *Qorivva MPC5675K Microcontroller Reference Manual* (Document Number: MPC5675KRM, Rev. 7, December 2011)
- *Qorivva MPC5675K Microcontroller Data Sheet* (Document Number: MPC5675K, Rev. 6, February 2012)
- MPC567XK Mask Set Errata for Mask 0N72D (Document Number: MPC567XK_0N72D, Rev. 12, April 2012)
- Enhanced Signal Processing Extension and Embedded Floating-Point Version 2 Auxiliary Processing Units Programming Interface Manual (Document Number: SPE2PIM, Rev. 1.0-1: Based on Specifications SPE2rev 1.0 and EFP2rev 1.3, October 2011)
- Application note, *MPC5675K Test and Shadow Flash Parameters for ADC Self Test, MBIST, and LBIST* (Document Number: AN4422, Rev. 0, February 2012)
- *e200z760n3 Power Architecture Core Reference Manual* (Document Number: e200z760RM, Rev. 0, June 2010)
- FMEDA
  - Core_FMEDA
  - Clock_FMEDA
  - Flash_FMEDA
  - SRAM_FMEDA
  - Power_Supply_FMEDA
  - Peripheral_Failurerates
- Addressing the Challenges of Functional Safety in the Automotive and Industrial Markets, White Paper, October 2011

1.2 Vocabulary

For the purposes of this document, the vocabulary defined in ISO 26262-1 and IEC 61508-4 apply to this document.

Specifically, the following terms apply.

- **System:** functional safety-related system that both implements the required functional safety goals necessary to achieve or maintain a Safe state$_{system}$ for the equipment under control (control system) and is intended to achieve, on its own or with other Electrical/Electronic/Programmable Electronic functional safety-related systems and other risk reduction measures, the necessary functional safety integrity for the required safety functions.
- **System integrator:** person who is responsible for the system integration.
- **Element:** part of a subsystem comprising a single component or any group of components (for example, hardware, software, hardware parts, software units) that performs one or more element safety functions (functional safety requirements).
- **Trip time:** the maximum time of operation of the MCU without switching to power down state.
- **Functional safety requirement:** system level environmental requirement relevant to achieve functional safety in the specific application under consideration (condition of use).

2 General information

The MPC567xK is designed to be used in automotive or industrial applications which need to fulfill functional safety requirements as defined by functional safety integrity levels (for example, ASIL D of ISO 26262 or SIL 3 of IEC 61508).

The MPC567xK is considered a Type B subsystem (“complex”, see IEC 61508-2, section 7.4.4.1.3) with HFT = 0 (Hardware Fault Tolerance), and may be used in any mode of operation (see IEC 61508-4, section 3.5.16).

2.1 Assumed conditions of operation

Safety requirement: [SM_001] This document is only valid if the recommended operating conditions given in the *Qorivva MPC5675K Microcontroller Data Sheet* are maintained. [end]

Safety requirement: [SM_002] This document is only valid if the recommended production conditions given in the MPC567xK quality agreement are maintained. [end]

Safety requirement: [SM_003] The latest device errata shall be taken into account during system design, implementation, and maintenance. For a functional safety-related device such as the MPC567xK, this also concerns functional safety-related activities such as system functional safety concept development. [end]

2.2 Safety function

Given the application independent nature of the MPC567xK, no general safety function can be specified. Therefore, this document specifies a safety function being application independent for the majority of

applications. This application independent safety function would have to be integrated into a complete (application dependent) item. Application independent safety functions include:

- Read Instructions out of flash memory, buffer these within instruction cache, execute instructions, read data from RAM or flash memory, buffer these in data cache, process data, and write back result data into RAM.
- (Optional) Writing to flash memory EEPROM is in general not part of the safety function, as it is performed in a safe and controlled maintenance environment and the flash memory EEPROM content validated multiple times before bringing back to operation. Only intermittent fault (non DC Fault) for example, due to weak programming might cause a safety thread but is a maintenance issue in principle and not a random fault field issue.
- (Optional) Reading data from SRAM or flash memory EEPROM, buffering data in DMA FIFO, and writing data to SRAM or flash memory EEPROM by DMA engine.
- (Optional) Receiving external interrupt signals, branching instruction execution to interrupt service routine.

2.3 Safe state

A Safe state of the system is named Safe $state_{system}$ whereas a Safe state of the MPC567xK is named Safe $state_{MCU}$. A Safe $state_{system}$ of a system is an operating mode without an unreasonable probability of occurrence of physical injury or damage to the health of persons. A Safe $state_{system}$ may be the intended operating mode or a mode where it has been disabled.

Likewise, a Safe $state_{MCU}$ of the MPC567xK is by definition one of following operation modes (see Figure 1):

a) Operating correctly
b) Explicitly indicating an internal error (FCCU_F[0:1])
c) Reset
d) Completely unpowered
e) Safe Mode (functional safety relevant outputs of the MCU are forced to a high impedance state due to MC_ME_SAFE_MC[PDO] = 1, no active output for example, tristate).

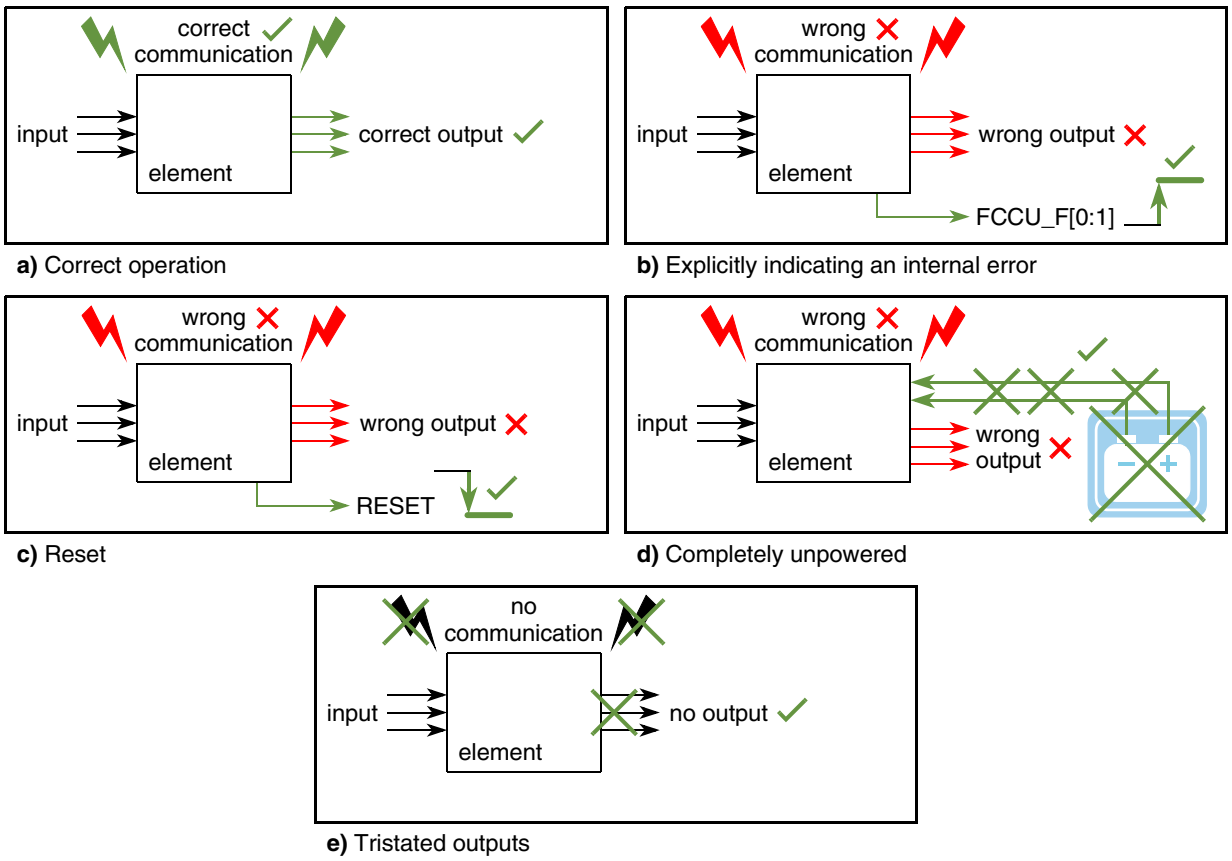


Figure 1. Safe state$_{MCU}$ of the MPC567xK

Safety requirement: [SM_004] The system transitions itself to a Safe state$_{system}$ when the MPC567xK explicitly indicates an internal error (FCCU_F[0:1]). [end]

Safety requirement: [SM_005] The system transitions itself to a Safe state$_{system}$ when the MPC567xK is in reset state. [end]

Safety requirement: [SM_006] The system transitions itself to a Safe state$_{system}$ when the MPC567xK is completely unpowered. [end]

Safety requirement: [SM_007] The system transitions itself to a Safe state$_{system}$ when the MPC567xK has no active output (for example, tristate). [end]

2.4 Single-point Fault Tolerant Time Interval and Process Safety Time

The single-point Fault Tolerant Time Interval (FTTI)/Process Safety Time (PST) is the time span between a failure that has the potential to give rise to a hazardous event and the time by which counteraction has to be completed to prevent the hazardous event occurring. It is used to define the sum of worst case fault indication time and time for execution of corresponding countermeasures (reaction). Figure 2 shows the FTTI for a single-point fault occurring in the MCU (Figure 2a) with an appropriate functional safety

mechanism to handle the fault (Figure 2b). Without any suitable functional safety mechanism, a hazard may appear after the FTTI elapsed (Figure 2c).

PST in IEC 61508 is the equivalent of FTTI in ISO 26262. Whenever single-point fault tolerant time interval or FTTI is mentioned in this document, it shall be read as PST for IEC 61508 applications.

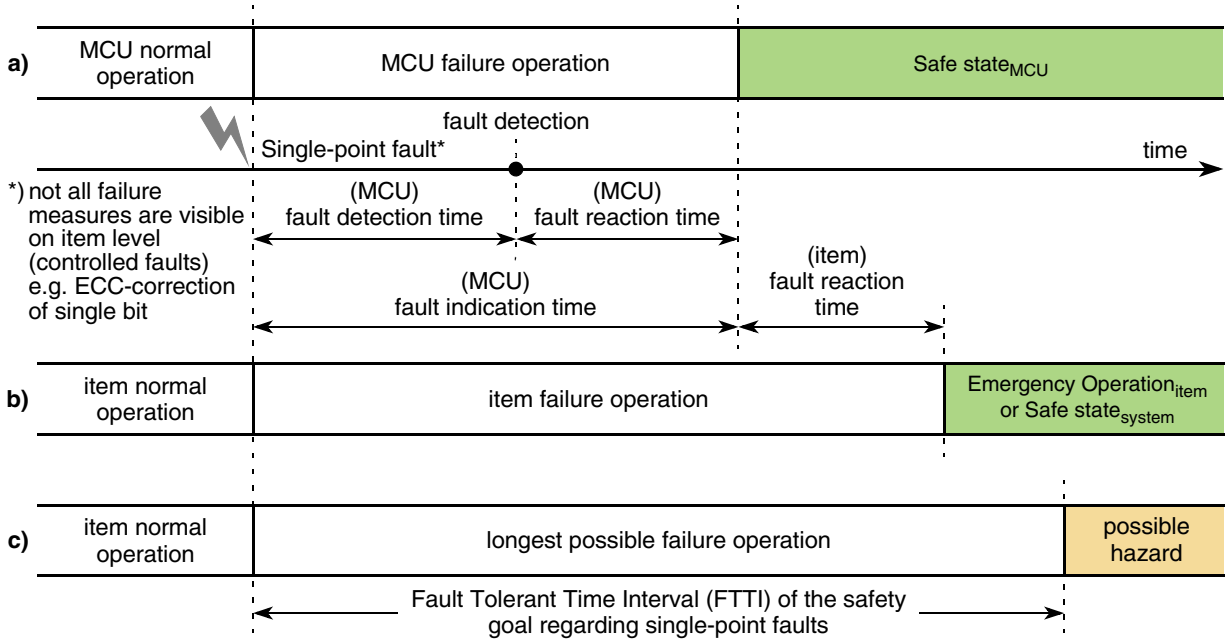


Figure 2. Fault tolerant time interval for single-point faults

Fault indication time is the time it takes from the occurrence of a fault to switching into Safe state$_{MCU}$ (for example, indication of that failure by driving the error out pins, forcing outputs of the MCU to a high impedance state, or by assertion of reset).

Fault indication time has five components, two of which are influenced by configuration settings: recognition + internal processing + external indication + software cycle + software execution.

Each component of fault indication time is described as follows:

- **Diagnostic test interval:** is the interval between on-line tests (for example, software based self test) to detect faults in a functional safety-related system. This time depends closely on the system implementation (for example, software).
  - **Software cycle time** of software based functional safety mechanisms. This time depends closely on the software implementation.
- **Fault detection time** is the maximum of the detection time of all involved functional safety mechanisms. The three mechanisms with the longest time are:
  - ADC[1] recognition time is a very demanding hardware test in terms of timing. The self-test requires the ADC conversion to complete a full test. A single full test takes at least 70 µs[2].

1.ADC recognition time is relevant only if ADC is used by the safety function.
2.This value takes into account the steps needed to run the three ADC hardware self-tests.

  - Recognition time related to the FMPLL loss of clock: it depends on how the FMPLL is configured. It is approximately 20 µs.
  - **Software execution time** of software based functional safety mechanisms. This time depends closely on the software implementation.
- **Fault reaction time** is the maximum of the reaction time of all involved functional safety mechanisms consisting of internal processing time and external indication time:
  - **Internal processing time** to communicate the fault to the FCCU lasts maximum 10 RC clock cycles (RC is the internal safe clock with nominal frequency of 16 MHz).
  - **External indication time** to notify an observer about the failure external to the MCU. This time depends on the indication protocol configured in the Fault Collection and Control Unit (FCCU):
    - Dual Rail protocol and time switching protocol
      - **FCCU configured as “fast switching mode”:** indication delay is maximum 64 µs. As soon as FCCU receives a fault signal, FCCU reports the failure to the system.
      - **FCCU configured as “slow switching mode”:** an indication delay could occur. The maximum delay is equal to period of the error out signal (FCCU_CFG.FOP). This parameter requires to be configured equal to its minimum which is 128 µs.
    - **Bi-stable protocol:** indication delay is maximum 64 µs. As soon as the FCCU receives a fault signal, it reports the failure to the system.

If the configured reaction to a fault is an interrupt an additional delay (interrupt latency) can occur until the interrupt handler is able to start executing (for example, higher priority IRQs, XBAR contention, register saving, and so on).

If the configured reaction to a fault is the forcing outputs of the MCU to a high impedance state (Safe Mode) an additional delay (Safe Mode Request Timer) may occur until the outputs transit into Safe Mode (for example, tristate).

The sum of the MCU fault indication time and system fault reaction time shall be less than the FTTI of the functional safety goal.

2.5 Latent-FTTI for latent faults

The Latent Fault Tolerant Time Interval (L-FTTI) is the time span between a latent fault that has the potential to coincidently show up with other latent faults and give rise to a hazardous multiple-point event and the time by which counteraction has to be completed to prevent the hazardous event occurring. It is used to define the sum of respective worst case fault indication time and time for execution of corresponding countermeasure. Within this time frame the safety element out of context (SEooC) shall be considered as unsafe. Figure 3 shows the L-FTTI for multiple-point faults in a system.

There is no equivalent to L-FTTI in IEC 61508.

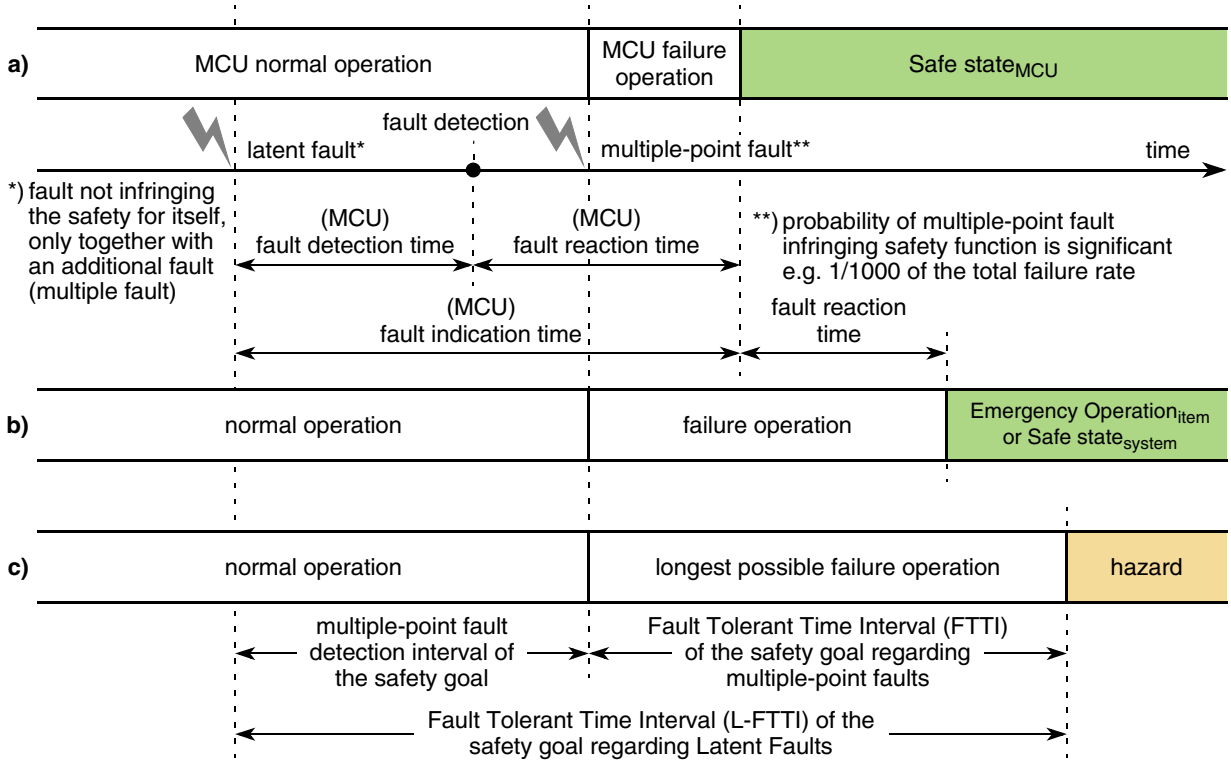


Figure 3. Fault Tolerant Time Interval for latent faults

Latent fault indication time is the time it takes from the occurrence of a multiple-point failure to when the indication of that failure is visible by driving the error out signals, forcing outputs of the MCU to a high impedance state (Safe Mode), or by assertion of reset.

Fault indication time has five components, two of which are influenced by configuration settings: recognition + internal processing + external indication + software cycle + software execution.

Each component of fault indication time is described as follows:

- **Diagnostic test interval:** is the interval between on-line tests (for example, software based self test) to detect faults in a functional safety-related system that has a specified diagnostic coverage. This time depends closely on the system level implementation (for example, software).
  - — **Software cycle time** of software based functional safety mechanisms. This time depends closely on the software implementation.
- **Fault detection time** is the maximum of the detection time of all involved functional safety mechanisms. The mechanisms with the longest time are:
  - — Single bit corrected permanent hardware SRAM fault – This fault is only controlled (corrected) it is not reported (not detected) to the operator of the system. Therefore, it is a latent triple fault scenario, as ECC has a reduced capability to detect triple bit faults. The L-FTTI is in the range of $1\times10^{9}h^{-1} \approx 2\times10^{5}$ years for a permanent single bit fault, or ≈ 20 years continuous operation for 10000 faults.
  - — **Software execution time** of software based functional safety mechanisms. This time depends closely on the software implementation.

- **Fault reaction time**
  - **Internal processing time** to communicate the fault to the RCCU lasts maximum 10 RC clock cycles (RC is the internal safe clock with nominal frequency of 16 MHz).
  - **External indication time** to notify an observer about the failure external to the MCU. This time depends on the indication protocol configured in the Fault Collection and Control Unit (FCCU):
    - Dual Rail protocol and time switching protocol
      - **FCCU configured as "fast switching mode":** indication delay is maximum 64 µs. As soon as FCCU receives a fault signal, FCCU reports the failure to the system.
      - **FCCU configured as "slow switching mode":** an indication delay could occur. The maximum delay is equal to period of the error out signal (FCCU_CFG.FOP). This parameter configured to be configured equal to its minimum which is 128 µs.
    - **Bi-stable protocol:** indication delay is maximum 64 µs. As soon as the FCCU receives a fault signal, it reports the failure to the system.

In general, internal processing time, indication time, and execution time are negligible for multiple-point failures since the L-FTTI is significantly larger than typical processing, indication, and execution times.

The sum of the MPC567xK latent fault indication and system latent and multiple-point fault reaction times shall be less than the L-FTTI of the functional safety goal.

2.6 Failure handling

Failure handling can be split into two categories:

- Handling of failures before enabling the system level safety function (for example, during/following the MCU initialization). These errors are required to be handled before the system enables the safety function, or in a time shorter than the respective FTTI or L-FTTI after enabling the safety function.
- Handling of failures during runtime with repetitive supervision while the safety function is enabled. These errors are to be handled in a time shorter than the respective FTTI or L-FTTI.

Safety requirement: [SM_008] Single-point and latent fault diagnostic measures shall complete operations (including fault reaction) in a time shorter than the respective FTTI or L-FTTI or alternatively single-point and latent fault diagnostic measures shall complete operations (including fault reaction) before enabling system level safety function. [end]

Recommendation: It is recommended to identify startup failures before enabling system level safety functions.

A typical failure reaction regarding power-up/start-up diagnostic measures is not to initialize and start the safety function and instead provide failure indication to the operator/user.

3 Functional safety concept

Failures are the main impairment to functional safety:

- A systematic failure is manifested in a deterministic way to a certain cause (systematic fault), that can only be eliminated by a change of the design process, manufacturing process, operational procedures, documentation, or other relevant factors. Thus measures against systematic faults are reduction of systematic faults for example, implementing and following adequate processes.
- A random hardware failure can occur unpredictably during the lifetime of a hardware element and follows a probability distribution. Thus, measures reducing the likelihood of random hardware faults is either the detection and control of the faults during the lifetime, or reduction of failure rates. A random hardware failure is caused by either a permanent fault (for example, physical damage), an intermittent fault, or a transient fault. Permanent faults are unrecoverable. Intermittent faults are for example, faults linked to specific operating conditions or noise. Transient faults are for example, particles (alpha, neutron) or EMI-radiation. An affected configuration register can be recovered by setting the desired value or by a power cycle. Due to a transient fault an element may be switched into a self destructive state (for example, single event latch up) and therefore may cause permanent destruction.

3.1 Faults

The following random faults may generate failures, which may lead to the violation of a functional safety goal. Citations are according to ISO 26262-1. Random hardware faults occur at a random time, which results from one or more of the possible degradation mechanisms in the hardware.

- **Single-Point Fault (SPF):**
  A SPF is “a fault in an element that is not covered by a safety mechanism” and that results to a single-point failure “which leads directly to the violation of a safety goal”. Figure 4a shows a SPF inside an element, which generates a wrong output. The equivalent in IEC 61508 to Single-Point Fault is named **Random Fault**. Whenever a SPF is mentioned in this document, it is to be read as a random fault for IEC 61508 applications.
- **Latent Fault (LF):**
  A LF is a “multiple-point fault whose presence is not detected by a safety mechanism nor perceived by the driver”. A LF is a fault which does not violate the functional safety goal(s) itself, but it leads in combination with at least one additional independent fault to a dual- or multiple-point failure, which then leads directly to the violation of a functional safety goal. Figure 4b shows a LF inside an element, which still generates a correct output. No equivalent in IEC 61508 to LF is named.
- **Residual Fault (RF):**
  A RF is a “portion of a fault that by itself leads to the violation of a safety goal”, “where the portion of the fault is not covered by a functional safety mechanism”. Figure 4c) shows a RF inside an element, which - although a functional safety mechanism is set in place - generates a wrong output, as this particular fault is not covered by the functional safety mechanism.
- **Dual-point fault (DPF):**
  A DPF is an “individual fault that, in combination with another independent fault, leads to a dual-point failure” which leads directly to the violation to a goal. Figure 4d shows two LF inside an element, which generates a wrong output.

- **Multiple-point fault (MPF):**
  A MPF is an “individual fault that, in combination with other independent faults, leads to a multiple-point failure” which leads directly to the violation of a functional safety goal. Unless otherwise stated multiple-point faults are considered as safe faults and are not covered in functional safety concept of the MPC567xK.
- **Safe Fault (SF):**
  A SF is a “fault whose occurrence will not significantly increase the probability of violation of a safety goal”. Safe faults are not covered in this document. Single-point faults, residual faults or dual-point faults are not safe faults.

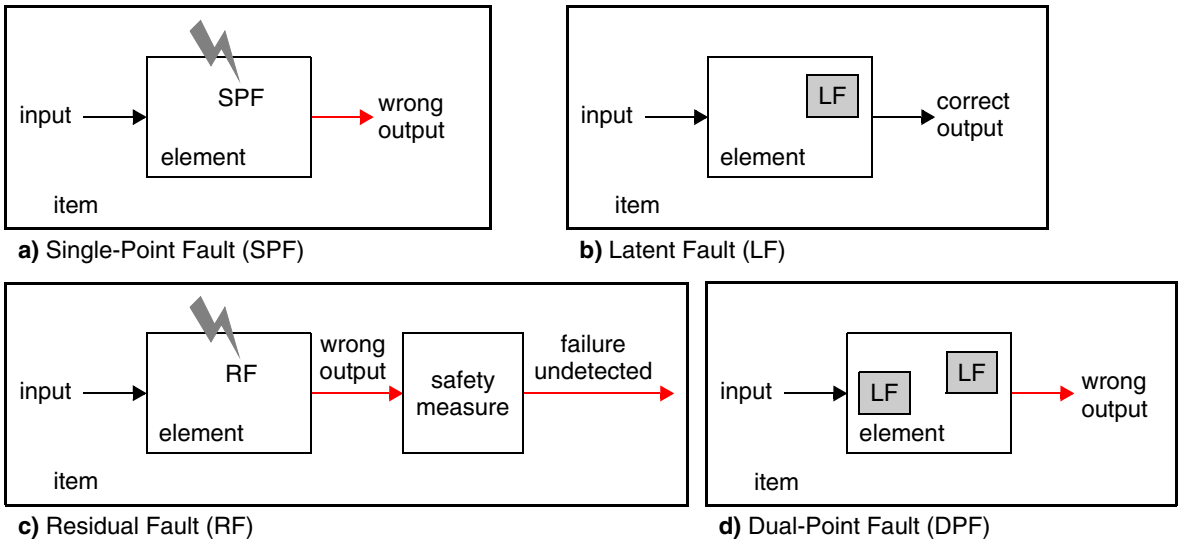


Figure 4. Faults

SPFs shall be detected within the FTTI. Latent Faults (dual-point faults) shall be detected within the L-FTTI. In automotive applications L-FTTI is in general accepted to be once per typical automotive trip time (T_{trip}) by test routines (for example, BIST after power-up). This reduces the accumulation time of latent faults from life-time of the product T_{life} to T_{trip}.

3.2 Failures

- **Common Cause Failure (CCF):**
  CCF is a coincidence of random failure states of two or more elements in separate channels of a redundancy element leading to the defined element failing to perform its intended safety function resulting from a single event or root cause (chance cause, non-assignable cause, noise, Natural pattern, …). Common Cause Failure causes the probability of multiple channels (N) having a failure rate to be larger than $\lambda_{\text{single channel}}{}^{N}$ ($\lambda_{\text{redundant element}} > \lambda_{\text{single channel}}{}^{N}$).

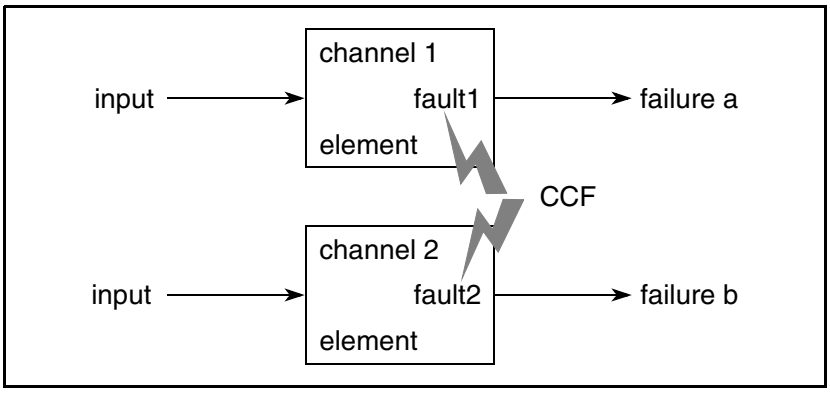


Figure 5. Common Cause Failures

- **Common Mode Failure (CMF):**
  CMF is a subset of CCF. A a single root cause leads to similar coincidental erroneous behavior (with respect to the safety function) of two or more (not necessarily identical) elements in redundant channels, resulting in the inability to detect the failures.
  Figure 6 shows three elements within two redundant channels. One single root cause (CMF A or CMF B) leads to undetected failures in the primary channel and in one of the elements of the redundant channel.

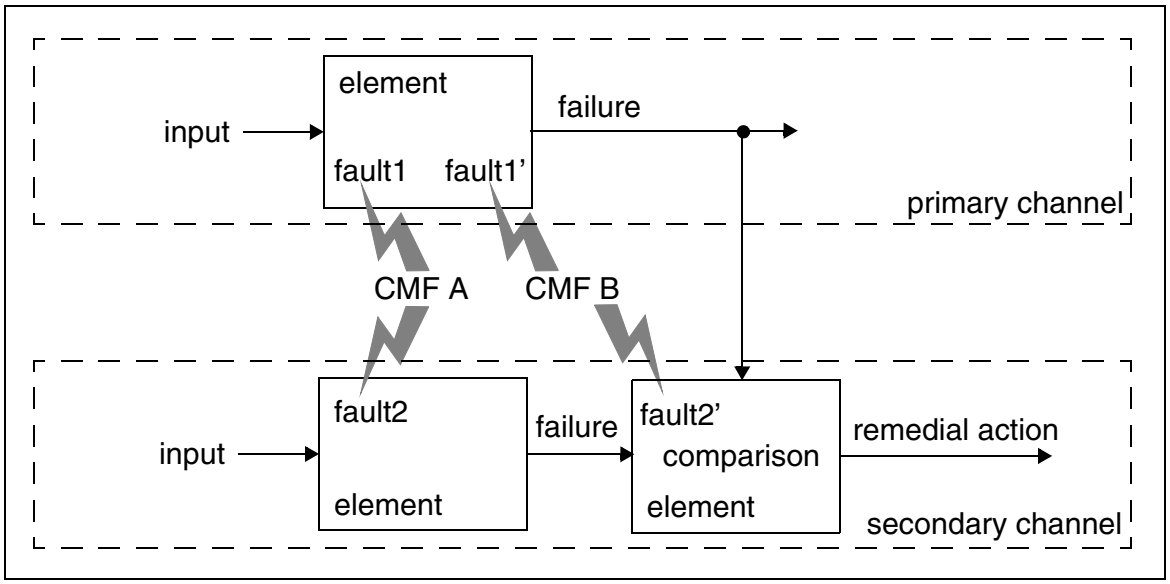


Figure 6. Common Mode Failures

- **Cascading Failure (CF):**
  CFs occur when local faults of an element in a system ripple through interconnected elements causing another element or elements of the same system and within the same channel to fail. Cascading failures are dependent failures that are not common cause failures. Figure 7 shows two elements within a single channel, to which a single root cause leads to a fault (fault 1) in one

element resulting in a failure (failure a) causing a second fault (fault 2) within the second element (failure b).

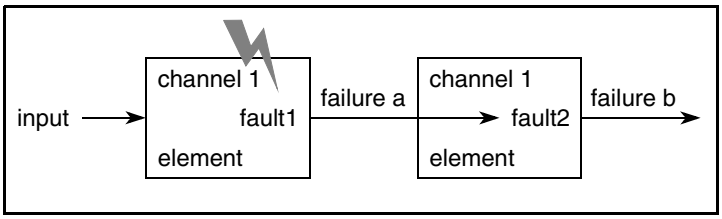
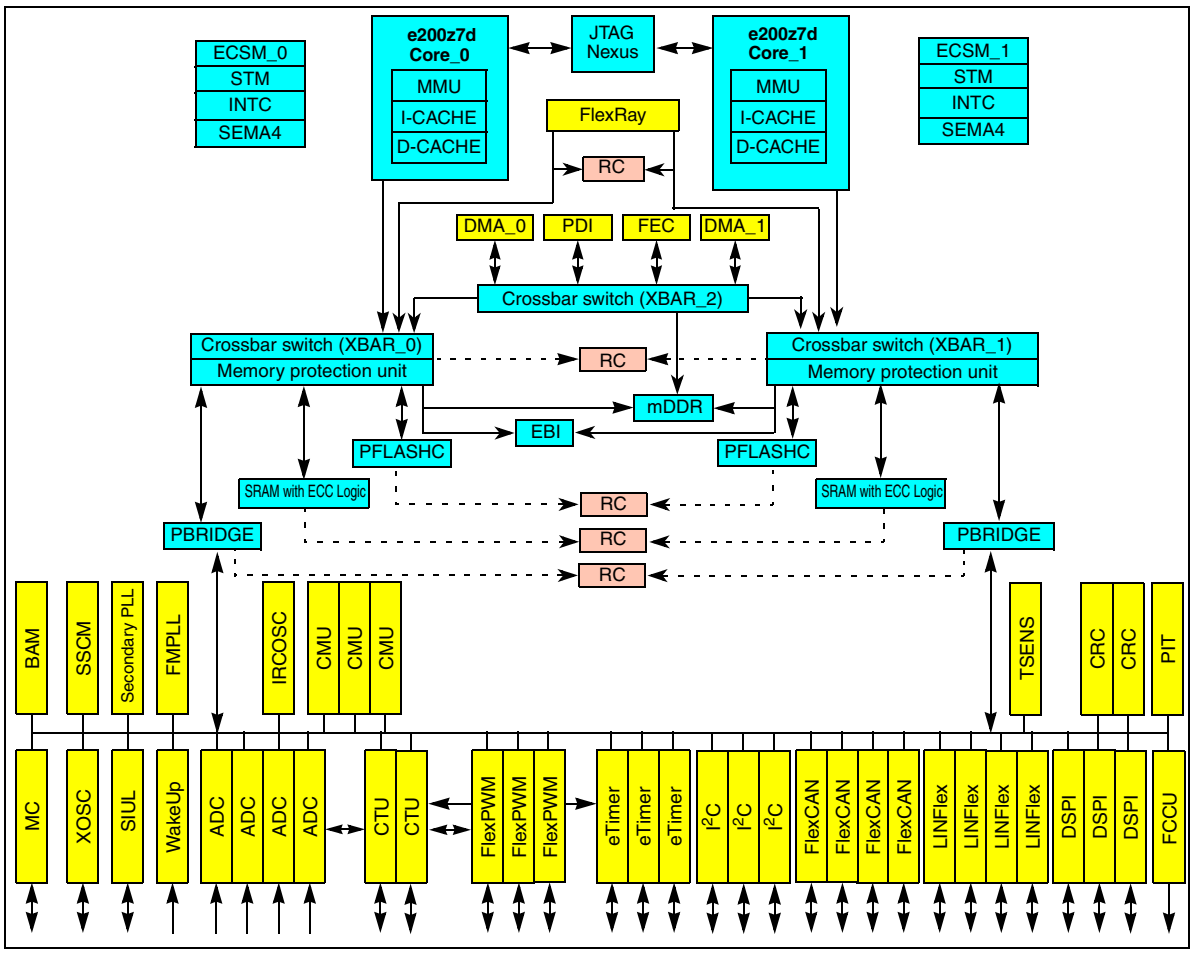


Figure 7. Cascading Failures

3.3 General functional safety concept

Figure 8 shows the block diagram of the MPC567xK.



ADC – Analog-to-digital converter
BAM – Boot assist module
CMU – Clock monitoring unit
CRC – Cyclic redundancy check unit
CTU – Cross triggering unit
DSPI – Deserial serial peripheral interface
EBI – External bus interface
ECC – Error correction code
ECSM – Error correction status module
eDMA – Enhanced direct memory access controller
FCCU – Fault collection and control unit
FEC – Fast Ethernet controller
FlexCAN – Controller area network controller
FlexPWM – Pulse width modulator module
FMPLL – Frequency-modulated phase-locked loop
I²C – Inter-integrated circuit controller
INTC – Interrupt controller
IRCOSC – Internal RC oscillator
JTAG – Joint Test Action Group interface
MC – Mode entry, clock, reset, & power modules
mDDR – Mobile double data rate dynamic RAM
PBRIDGE – Peripheral bridge
PDI – Parallel data interface
PIT – Periodic interrupt timer
PMU – Power management unit
RC – Redundancy checker
RTC – Real time clock
SEMA4 – Semaphore unit
SIUL – System integration unit Lite
SSCM – System status and configuration module
STM – System timer module
SWT – Software watchdog timer
TSENS – Temperature sensor
XOSC – Crystal oscillator

Figure 8. MPC567xK block diagram

Functional Safety integrity measures are as follows:

- Replication of IP: A dual core architecture reduces the need for component duplication at the system level, and lowers overall system level complexity.
- For the dual cores and their closely related periphery, functional safety is improved by a lockstep approach. Any deviation in the output of the two cores is detected by hardware and signaled as a possible failure.
- Error correction or detection, or both, for flash memory and SRAM to reduce the effect of transient faults (data and address domain) and permanent faults in integrated volatile and non volatile memory.
- Safety measure to validate error correction circuitry of ECC modules in real time.
- Error correction or detection for cache and cache TAGS to control the majority of transient faults and permanent faults.
- The generation and distribution of clock and power are supervised by dedicated monitors.
- Dedicated March built-in self-test circuitry for every SRAM module (48 MBIST modules) detecting address decoding faults, stuck-at faults, and transition faults with a high level of coverage. Reduced coverage is provided for coupling faults (for example, shorts and bridging faults), delay faults (such as, serial resistance), and stuck open faults (such as, resistive) of word line and bit line select drivers.
- Dedicated CRC built-in self-test circuitry for flash memory modules detecting address decoding faults, stuck-at faults, transition faults, coupling faults (for example, shorts and bridging) and delay faults (such as, serial resistance). Stuck-open faults (such as, resistive) of word line and bit line select drivers have a lower level of coverage.
- Dedicated test circuitry to identify weak faults (such as, drift of memory cells) in flash memory.
- Programmer interface to the scan chain Design For Test measure (Logical Built-In Self-Test (LBIST)). This LBIST allows setting and observing every flip-flop in an integrated circuit. Pattern and signature generators are included to allow for flexible vector count and diagnostic coverage.
- An integrated state machine is used to test the internal structural elements of the analog to digital converter (ADC). It follows the principal that internally applies twelve different voltage levels using the internal digital to analog converter (DAC).
- The ADC has integrated watchdogs to detect timing faults in the ADC.
- Pre-sampling circuitry enabling to detect open faults on ADC inputs and multiplexers.
- Hardware FIFO to enable time redundancy (oversampling) for ADC inputs.
- Multiple instantiations of CAN protocol controller (FlexCAN) supporting listen only mode that enables validation of messages received and transmitted using the multiplexed CAN bus.
- Diverse instantiations of timer and PWM modules (eTimer, FlexPWM, respectively) supported by hardware Cross Trigger Unit (CTU) synchronization circuits.
- Hierarchical memory protection (system MPU, core MMU) to reduce the effects of interference in the address domain with respect to hardware faults and software encapsulations.
- Diverse instantiations of clock generation (quartz based PLL clock and RC based oscillator).
- Safety Measures granting Boot ROM (BAM) not inferring to the functional safety.
- Safety measures granting debug modules not inferring to functional safety.

- Multiple instantiations of high speed core buses, peripheral buses and bridges allowing detection of time, control, and data domains.
- Multiple instantiations of DMA to allowing safe DMA operation.
- Multiple instantiations of OS timer to enable safe OS operations.
- Protection of critical registers to support coverage of software and hardware faults.
- Frequency metering to improve the precision of the Internal RC Oscillator (IRCOSC) to improve diagnostic coverage of clock supervision.
- Individual slew rate control for pads. This allows for stronger I/O drive for safety critical systems and weaker drive for non-safety critical signals.
- Multiple watchdogs that have independent time-bases and time-windows to monitor proper program execution.
- Integrated measures to statistically monitor the type and number of instructions executed (logical control flow supervision - performance monitor).
- Multiple instances of an interrupt module that detects faults in the control and time domains.
- Multiple instances of hardware CRC modules supporting application level signature measures (for example, safety protocol for serial communication protocols).
- The Fault Collection and Control Unit (FCCU) is responsible for collecting and reacting to failure notifications.
- The Reset Generation Module (MC_RGM) replicates the reaction to failure notification for a set of critical failures.
- Risk of CMFs are reduced by:
  - spatial separation of diagnostic and mission channels (core domain, I/O domain).
  - buffering critical signals that physically cross non-critical channels.
  - redundant monitoring of supply voltages and clocks.
  - different synthesize constraints of mission and diagnostic channel elements.
  - redundantly comparing replicated modules.
  - separating redundant clock sources in the time domain (different frequencies).
- The functional safety of the periphery is ensured by application-level (system-level) measures (such as connecting one sensor to different I/O modules, sensor validation by sensor fusion, etc.).
- Safety measures to detect erroneous test mode activation.
- Usage of internal (and external) watchdogs or timeout measures.
- Hardware semaphore unit that supports software encapsulation and improve data consistency.
- Dedicated mechanisms are provided to check the functionality of each error reaction path (such as by application controlled fault injection).
- Redundant fault signal (time domain, data domain, and/or spatial domain).
- Signature

Both cores can operate in either one of two distinct operating modes: Lock-Step Mode (LSM) or Decoupled Parallel Mode (DPM). In DPM, the two channels of the MCU work independently. Automatic hardware checks for equal operation between the two channels are disabled in DPM. When in DPM,

system level software measures are needed to achieve adequate functional safety integrity (for example, by implementing reciprocal comparisons).

The operating mode (LSM or DPM) on the MPC567xK is determined by the LSM_DPM user option bit in the shadow block of the flash memory, and is configured to the appropriate mode for the system level functional safety concept (see “Selecting LSM or DPM” section of the “Operating Modes” chapter in the *Qorivva MPC5675K Microcontroller Reference Manual*).

As LSM is transparent to the system level (for example, to application software) specific requirements must be fulfilled to improve functional safety integrity in case the device is intended to operate in LSM (see Section 5.2.22, System Status and Configuration Module (SSCM)).

The MPC567xK microcontroller support only static configuration at power-on (either LSM or DPM).

3.3.1 Sphere of Replication - Lockstep Mode (LSM)

The Sphere of Replication (SoR) contains all hardware elements which are replicated for functional safety reasons The replication is to detect permanent, dormant, latent and transient faults. The following modules are included in the SoR:

- e200z7 Core (including Memory Management Unit)
- Enhanced Direct Memory Access (eDMA)
- Interrupt Controller (INTC)
- Crossbar Switch (XBAR)
- Memory Protection Unit (MPU)
- Flash Memory Controller (PFlashC)
- Static RAM Controller (SRAMC)
- System Timer Module (STM)
- Software Watchdog Timer (SWT)
- Peripheral Bridge (PBRIDGE)

In LSM mode each member of such a pair executes the same operations or transactions as its partner resulting in lockstep behavior, where both cores and their corresponding peripherals are in synchronicity. The test for equal execution is checked on the boundary of the SoR by the redundancy Control Checker Units (RCCU).

Thus the RCCUs implement a modified fault isolation in a way that they detect but not prevent the propagation of a non-common mode failure at the point where the two redundant channels are merged into a single actuator or recipient.

Isolation of the overall system is then achieved by the Fault Collection and Control Unit (FCCU) signaling an error, thereby allowing the device or application to react appropriately.

In order to simplify application software, the software executes transparently on both cores of the MPC567xK and application sees only one logical core.

3.3.2 SoR – DPM

The SoR contains all hardware elements which are replicated for functional safety reasons The replication is to detect permanent, dormant, latent and transient faults. The following modules are included in the SoR:

- e200z7 Core
- Enhanced Direct Memory Access (eDMA)
- Interrupt Controller (INTC)
- Crossbar Switch (XBAR)
- Memory Protection Unit (MPU)
- Flash memory controller (PFlashC)
- Static RAM Controller (SRAMC)
- System Timer Module (STM)
- Software Watchdog Timer (SWT)
- Peripheral Bridge (PBRIDGE)
- Static Random Access Memory (SRAM)
- Semaphore Unit (SEMA4)

In this mode, each CPU core executes code independently. This mode of operation configures the chip into a symmetric multi-core processor. When in this mode, the redundancy checkers are disabled, and the replicated peripherals are available at a different set of addresses. The SRAM is split in half and relocated. In DMP mode, the hardware Semaphore module becomes available. Functional safety integrity has to be achieved by system level measures. system level measures requires implementing three different classes of measures:

- Avoidance of shared resources (reduce dependability): system should not use both replications of replicated resources in a single channel for the safety function. A possible measure would be exclusive usage of one of the replicated modules in each channel (for example, SRAM_0 is only used by CPU_0 and SRAM_1 is only used by CPU 1.
- Comparing the results: system should implement the compare of the different channel data. A possible measure application may to implementing reciprocal comparison by application software. Semaphore Unit (SEMA4) may be used to control access to reciprocal data.
- Separation of resources (reduce interference): system should configure MPU, MMU or other hardware measures to disable accesses to replicated resources of different replication sphere. Only exception may be a small SRAM section to read reciprocal data (only read access) from other channel (for example, I/O input, I/O output, calculation results). The Semaphore Unit (SEMA4) may be used to control access to reciprocal data set.

4 Hardware requirements on system level

This section lists necessary or recommended measures on the system level for the MPC567xK to achieve the functional safety goal(s).

The MPC567xK offers an integrated functional safety architecture using dual-core lockstep CPU (LSM), a variety of replicated function blocks, several self-test units and other elements to detect faults. By these

means, SPFs and LFs can be detected with a high diagnostic coverage. However, not all common mode failures may be detected. In order to detect failures, which may not be detected by the MPC567xK itself, it is assumed that separate circuitry is used to bring the system into Safe state$_{system}$ in such cases.

Figure 9 depicts a simplified application schematic for a functional safety relevant application in conjunction with a separate IC (only functional safety-related elements shown). The MPC567xK is supplied with its required supply voltages (1.2 V, 3.3 V and opt. 5 V). Although for most applications the 1.2 V for digital core supply is generated by an external ballast transistor from 3.3 V supply, internal ballast transistor of the MPC567xK can be used as well. Voltages generated within the separate IC need to be monitored for over voltage (over voltage supervision).

The separate integrated circuit also monitors the state of the error out signals FCCU_F[*n*] (error monitor). Through a communication interface (for example, SPI), the MPC567xK repetitively triggers the watchdog of the separate IC. In case of a failure (for example, watchdog not serviced correctly), reset output is asserted LOW to reset the MPC567xK. A fail-safe output is available to control or deactivate any fail-safe circuitry (for example, power switch).

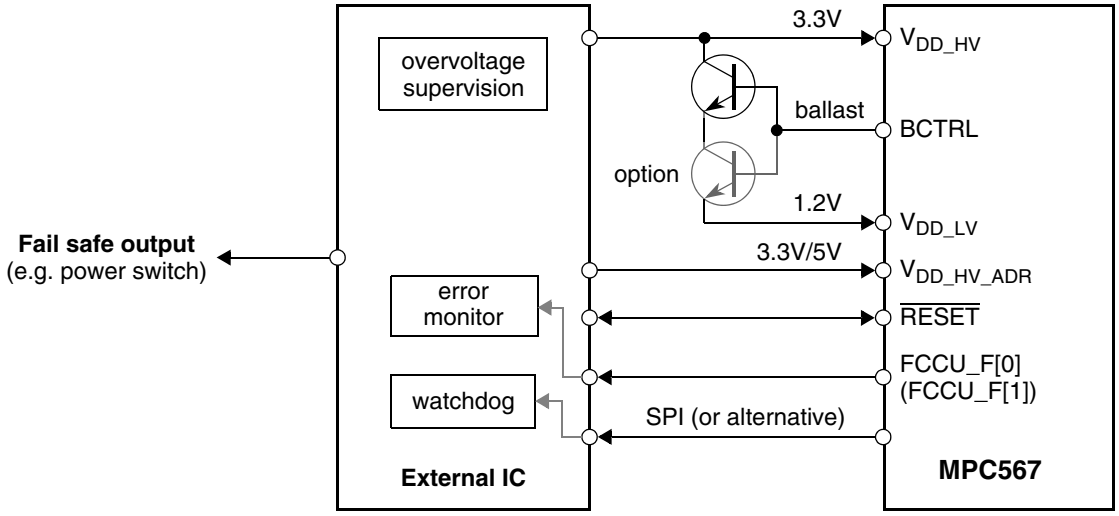


Figure 9. Functional safety-related connection to external circuitry

4.1 Assumed functions by separate circuitry

This section describes separate components supporting the usage of the MPC567xK for application requiring high functional safety integrity levels.

Failure rates of external services are only included for specific circuitries (clock, 1.2 V supply) in the FMEDA of the MPC567xK and have to be included in the system FMEDA by the system integrator.

4.1.1 High impedance outputs

Rationale: In order to bring the functional safety-critical outputs to such a level, that a Safe state$_{system}$ is achieved.

Implementation hint: If the Safe $state_{MCU}$ is “Completely unpowered” and “No active output (tristate)” is not compliant to system-level Safe $state_{system}$, a possible system-level countermeasure may be to place pull-up or pull-down resistors to match the two sets of Safe states.

4.1.2 External Watchdog (EXWD)

Implementation hint: If the Safe $state_{MCU}$ (completely unpowered and no active output (tristate)) is not compliant to system level Safe $state_{system}$, possible system level countermeasures can be implemented using an external timeout to match the two sets of Safe states. A timeout may be used to switch to Safe $state_{system}$ in case the Safe $state_{MCU}$ completely disables proper MCU operation. Section 7.3.3, External timeout function describes this measure in details.

Safety requirement under certain preconditions: [SM_031] If the system requires robustness regarding catastrophic CMFs (such as, issues with the MPC567xK external power supply) and robustness regarding systematic faults, system level measures are to be implemented. [end]

Recommendation: Implement using separate element (separate silicon substrate) measures to prevent or detect catastrophic CMFs (such as, issues with the MPC567xK external power supply on system).

Implementation hint: An external timeout (EXWD) may be a measure detecting catastrophic CMFs, such as failure of the MPC567xK external power supply and systematic failures. If a failure is detected, the external timeout (watchdog) function (EXWD) switches the system to a Safe $state_{system}$ within the FTTI.

The timeout (watchdog) may be triggered periodically by the MPC567xK within the functional safety relevant software. The trigger may be discrete signal(s) or message object(s). If within a timeout period not triggered, a failure is detected by the external timeout (watchdog) function which switches the whole system to a Safe $state_{system}$ within the FTTI (for example, the EXWD disconnects the MPC567xK from the power supply, or the communication messages are invalidated by disabling the physical layer driver).

The implementation of the communication between the MPC567xK and the separate device can be chosen as desired. The timeout (watchdog) can be triggered by communication via (examples):

- serial link (SPI),
- toggling I/O (GPIO),
- periodic message frames (FlexCAN, FlexRay, Fast Ethernet),
- toggling FCCU_F[0], FCCU_F[1] error out signals from the FCCU

4.1.3 Power Supply Monitor (PSM)

Supply voltage above the specified operational range might cause permanent damage to the MPC567xK even if kept in reset. Therefore it is either required in case of over voltage to de-energize the MPC567xK or to decommission/replace the MPC567xK after an over voltage event (continuos disable safety function).

Safety requirement under certain preconditions: [SM_032] Measures maintaining system level the Safe $state_{system}$ during and after any supply voltage above the specified operational range are required. The *Qorivva MPC5675K Microcontroller Data Sheet* provides the specified operating voltage range to be maintained. [end]

Recommendation: On the system level, in order to avoid a situation where an over-voltage will be supplied to the MPC567xK, permanently disable (Safe $state_{system}$) the system when an over-voltage is recognized.

Implementation hint: A separate and independent device may provide an over voltage monitor for the MPC567xK external 3.3 V supplies. If the power supply is above the recommended operating voltage range of the MPC567xK, the MPC567xK is to be kept powerless and the power supply monitor switches the system to a Safe $state_{system}$ within the FTTI and maintain it in Safe $state_{system}$ (Overvoltage protection with functional safety shut-off or a switch-over to a second power supply unit).

In case that over voltages can be completely inhibited by design of the power supply, over voltage monitoring is dispensable.

Over voltage on 1.2 V core supply may be detected by the MPC567xK itself. But system level measure may be required to maintain the Safe $state_{system}$ in case an over voltage cause destructive damages within the MCU.

4.1.4 Error Out Monitor (ERRM)

If the MPC567xK signals an internal failure via its error out signals (FCCU_F[0] and optionally FCCU_F[1]), the system cannot rely on the integrity of the MPC567xK outputs for safety functions. If an error out is indicated, the system must transition to and remain in Safe $state_{system}$. Depending on its functionality, the system might disable or reset the device as a reaction to the indicated error out (see the **Safety requirements** in Section 2.3, Safe state).

The system integrator can choose between two different methods to interface to the FCCU:

- Both FCCU signals connected to the separate device
- Only a single FCCU signal connected to the separate device

Both FCCU configurations work properly with all the supported error out protocols. Refer to the *Qorivva MPC5675K Microcontroller Reference Manual* for a list of supported protocols.

Recommendation: Using a correctly configured FCCU to report failure detection is recommended to fulfill the system level requirements regarding FTTI.

Rationale: To monitor the error out signals for correct functionality of the device.

The system (for example, ECU) may not rely on any output I/O other than FCCU_F[0] and FCCU_F[1], when those signals indicate an error.

4.1.4.1 Both FCCU signals connected to separate device

In this configuration the separate device continuously monitors the output of the FCCU. Thus it can detect if the FCCU does not work properly.

This configuration may not require any dedicated software support.

Rationale: To check the integrity of the FCCU and FCCU signal routing on the system level.

Implementation hint: If both error out signals (FCCU_F[0] and FCCU_F[1]) are connected to a separate device, the separate device may check both signals, taking into account that FCCU_F[0] = $\overline{\text{FCCU\_F[1]}}$.

Monitoring the error output signals through an asynchronous combinatorial logic (for example, XOR gate) can generate some glitches. Synchronous sampling or asynchronous oversampling these signals reduces the likelihood of glitches.

4.1.4.2 Single FCCU signal connected to separate device

A single signal, FCCU_F[0] (or FCCU_F[1]), is connected to the separate device.

If a fault occurs, the FCCU communicates it to the separate device through the FCCU_F[0] (or FCCU_F[1]) signal.

The functionality of FCCU_F[0] (or FCCU_F[1]) can at least be checked in the following manner:

- FCCU_F[0] (or FCCU_F[1]) output read back (internal connection) in case of voltage domain coding.
- FCCU_F[0] (or FCCU_F[1]) output connected externally to a normal GPIO in case of voltage domain coding.
- FCCU_F[0] (or FCCU_F[1]) output uses a time domain coding (for example, is active for a deterministic time interval once per deterministic time period).

The system integrator is asked to choose which solution fits the system level functional safety requirement.

The advantage of a single FCCU_F[*n*] signal being used instead of using both FCCU_F[*n*] signals as in the previous section, is the lack of necessity for a separate device to be used for comparing the FCCU_F[*n*] signals.

4.1.4.2.1 Single FCCU signal connected to separate device using voltage domain coding

Recommendation: If a single signal, FCCU_F[0] or FCCU_F[1], is connected to a separate device not applying a time domain coding, the correct operation of this signal is to be checked before executing any safety function.

Rationale: To check the integrity of the FCCU error out I/O.

To verify the functionality of an FCCU_F[*n*] signal, a fault may be injected and the behavior of the pin may be checked by the other error out signal, or GPIO. It's possible to change the polarity of the error out signal by configuring the FCCU_CFG[FCCU_CFG.PS] bit. Other methods for checking the functionality of FCCU_F[0] (or FCCU_F[1]) may be implemented.

Since FCCU is monitoring the system, it is sufficient to check FCCU_F[0] (or FCCU_F[1]) within the L-FTTI (for example, at power-up) in order to reduce the risks of latent faults. It is recommended that FCCU_F[*n*] be checked once before the system begins performing the safety relevant function.

If the system is using the MPC567xK in a single error output signal mode, the application software configures the signals and pads neighboring the FCCU_F[0] (or FCCU_F[1]) to use a lower drive strength.

Using a lower drive strength on the GPIO near FCCU_F[0] (or FCCU_F[1]) will result in the higher current strength of FCCU_F[*n*] to effect the logic level of the neighboring GPIO in the event of a short circuit. Software may configure the slew rate for the relevant GPIO in the Pad Configuration Register (PCR).

4.1.4.2.2 Single FCCU signal line connected to separate device using time domain coding

Rationale: Decode the time domain coding.

Implementation hint: If a single signal line FCCU_F[0], or FCCU_F[1], is connected to a separate device applying a time domain coding (for example, a decoder), a window timeout or windowed watchdog function, is good practice.

A time domain coding of FCCU_F[0] (or FCCU_F[1]) signal may be implemented by periodically injecting a fault within a determined time interval. An external window timeout (window watchdog) is triggered by this time domain coding. In case of an unintended faults, the window watchdog is triggered not within its window and a respective unintended fault signaled.

Since FCCU is a monitor, it is sufficient to implement a time domain interval in the range of L-FTTI. It is recommended to toggle FCCU_F[0] (or FCCU_F[1]) signal once before the system begins performing the safety relevant function.

4.2 Optional hardware measures on system level

As I/O operations are highly application dependant, functional safety assessments are not effective on the MPC567xK level. Functional safety of I/O modules and peripherals may be assessed on system level. The following sections provide examples of possible functional safety mechanisms regarding some I/O operations.

Safety requirement under certain preconditions: [SM_033] When data communication is used in the implementation of a safety function, then system level functional safety mechanisms are required to achieve the necessary functional safety integrity of communication processes. [end]

Recommendation: System level measures to detect or avoid transmission errors, transmission repetitions, message deletion, message insertion, message resequencing, message corruption, communication delay and message masquerade improves the robustness of communication channels.

4.2.1 PWM output monitor (PWMA)

Parts of the integrated FlexPWM and eTimer do not provide the functional safety integrity IEC 61508 series and ISO 26262 requires for high functional safety integrity targets.

Safety requirement under certain preconditions: [SM_034] When FlexPWM outputs are used in the implementation of a safety function suitable system level functional safety integrity measures may be required to monitor these signals. [end]

Recommendation: System level measures to detect or avoid erroneous PWM output signals improve the safety integrity of the PWM channels.

Implementation hint: If the FlexPWM outputs are used for a safety critical function, the PWM output may be monitored by system level measures (software, hardware).

Monitoring can be implemented explicitly by monitoring the PWM signal directly by a separate device. An alternative approach monitors implicitly the PWM signal, by implementing an indirect PWM feedback loop, for example, measuring average current flow of full bridge driver. This approach may use diverse implemented input modules for example, the analogue to digital converter.

The distinctive PWM features that are to be managed by the system level measures are:

- Dead-time may need to be always positive and greater than the maximum value between TON and TOFF of the inverter switches.
- Open GPIO and short to supply or ground may need to be detected. This can be detected for example, by an MCU external feedback loop to a timer module of the MPC567xK capable to perform input capture functionality (for example, eTimer).

The system must be switched to Safe $state_{system}$ if the MPC567xK detects an error.

To reduce the likelihood of erroneous control (for example, a motor control application with dead-time requirements to reduce the likelihood of short circuits destroying the motor) in functional safety application using I/O to control an actuator with short FTTI, the functional safety requires system level supervision if the maximum fault indication time and fault reaction time of the MPC567xK exceeds the FTTI of the actuators.

If the PWM signals drive switches of a power stage (for example, bridge driver), the eTimer may not be fast enough to detect a dead-time fault because its fault indication time is often greater than the time required to avoid destruction of the power stage.

5 Software requirements on system level

This section lists required or recommended measures when using the individual components of the MPC567xK.

Given the application independent nature of the MPC567xK, no general safety function can be specified. To define a specific safety function the MPC567xK would have to be integrated into a complete (application dependent) system. Nevertheless, it is possible to define abstract element safety functions and safety integrity functions:

- An element safety function is used to implement (or control) a functional safety means with available hardware.
- A safety integrity function (often reductively called diagnostic measures) is to improve the probability of successful execution of a functional safety means.

It is nevertheless possible to ignore the required measures if equivalent measures to manage the same failures are included instead.

The modules covered by the SoR reach a very high diagnostic coverage (DC) without additional dedicated measures at application or system level.

5.1 Disabled modes of operation

The system level and application software must ensure that the functions described in this section are not activated while running functional safety-relevant operations.

5.1.1 Debug mode

The debugging facilities of the MCU pose a possible source of failures in case they activate during the operation of functional safety-relevant applications. They can halt the cores, cause breakpoints to hit, write to core registers and the address space, and activate boundary scan. The MCU may not enter debug mode to reduce the likelihood of interference with the normal operation of the application software. The state of the JCOMP signal determines whether the system is being debugged or whether the system operates in normal operating mode. When JCOMP is logic low, the JTAGC TAP controller is kept in reset for normal operating mode. When it is logic high, the JTAGC TAP controller is enabled to enter debug mode. On the system level, measures must be taken to ensure that JCOMP is not be asserted by external sources to avoid entering debug mode. The activation of debug mode is supervised by the FCCU, and it signals a fault condition when debug mode is entered. If the FCCU recognizes erroneous activation of debug mode, JTAG signals will no longer recognize any input as being legal debug commands.

Safety requirement under certain preconditions: [SM_067] If modules like SWT, STM, DSPI, PIT, FLEXRAY, FlexCAN, are functional safety relevant, it is required that application software configures the respective module to continue execution in debug mode and to not 'freeze' operation when in debug mode. [end]

Rationale: To improve robustness regarding erroneous activation of Debug mode.

Implementation hint: in Debug mode, the FRZ bit in the STM_CR register controls operation of the Software Watchdog Timer (SWT). If the FRZ bit is cleared ('0'), the SWT counter continues to run in Debug mode.

In Debug mode STM_CR[FRZ] controls operation of the System Timer Module (STM) counter. If the STM_CR[FRZ] = 0, the counter continues to run in Debug mode.

The DSPI_MCR[FRZ] controls Deserial Serial Peripheral Interface (DSPI) behavior in the debug mode. If DSPI_MCR[FRZ] = 0, the DSPI continues all active serial transfers when the device in the debug mode.

The FlexCAN_ MCR[FRZ] bit controls FlexCAN Module behavior in the debug mode. If the FRZ bit is cleared ('0'), the FlexCAN Module continues communication (not affected by debug mode) when the device in the debug mode.

The FR_ MCR[FRZ] bit controls FlexRay Communication Controller (FLEXRAY) behavior in the debug mode. If the FRZ bit is cleared ('0'), the FLEXRAY Module continues communication (not affected by debug mode) when the device in the debug mode.

In Debug mode, PITMCR[FRZ] controls operation of the Periodic Interrupt Timer (PIT) counter. If the PITMCR[FRZ] = 0, the counter continues to run in Debug mode.

When the MCU is in debug mode, the External Bus Interface (EBI) behavior is unaffected and remains operating as EBI module was configured. No specific action is required by application software.

The Interrupt Controller (INTC) operation in debug mode is identical to its operation in normal mode. No specific action is required by application software.

In Debug mode, DMACR[EDBG] controls operation of the Enhanced Direct Memory Access (eDMA). If DMACR[EDBG] = 0, the ipg_debug input is ignored by the eDMA, and eDMA continues normal operation in Debug mode.

When eTimer_CTRL3[DBGEN] = 00, the Enhanced Motor Control Timer (eTimer) continues normal operation while the device is in debug mode.

When FlexPWM_CTRL2[DBGEN] = 1, the Motor Control Pulse Width Modulator Module (FlexPWM) continues to run while the device is in debug mode.

5.1.2 Test mode

Several mechanisms of the MCU can be circumvented in test mode which endangers the functional safety integrity.

Safety requirement: [SM_010] Test mode is used for comprehensive factory testing and is not validated for normal operational usage. Test mode may not be used in normal operating mode without an explicit agreement by Freescale. [end]

Recommendation: Disable test mode by system level software measures.

Implementation hint: The VPP_TEST pin is for testing purposes only, and has to be tied to GND in normal operating mode. From a system level point of view, measures must ensure that the VPP_TEST pin is not asserted to V_{DD} during boot to avoid entering test mode. The activation of test mode is supervised by the FCCU and it signals a fault condition when test mode is entered.

5.2 MPC567xK modules

Recommendation: It is recommended to periodically check the contents of configuration registers (more than 10 registers) of modules attached to PBRIDGE_*n* by application measures to detect faults in the PBRIDGE.

5.2.1 Fault Collection and Control Unit (FCCU)

The FCCU offers a hardware fail safe channel to collect faults and to bring the device into a Safe $state_{MCU}$ when a failure has occurred.

All faults detected by hardware measures are reported to the central Fault Collection and Control Unit (FCCU). It monitors critical control signals and collects all errors. Depending on the particular fault, the FCCU puts the device into the accordingly configured Safe $state_{MCU}$. This prevents fault propagation (cascading faults) to system level. Only hardware configuration of the FCCU may be required by application software. No CPU intervention is required for collection and control operation.

The FCCU offers a systematic approach to fault collection and control. It is possible to configure the reaction for each fault source separately. The distinctive features of the module are:

- Collection of redundant hardware checker results (for example, the RCCU. See Section 5.2.6, Redundancy Control Checking Unit (RCCU))
- Collection of error information from modules whose behavior is essential with respect to the functional safety goal
- Configurable and graded fault control:
  - Internal reactions
    - No reset reaction
    - IRQ
    - Functional Reset
    - MPC567xK Safe Mode entered
  - External reaction (failure is reported to the outside world via output signal(s) FCCU_F[*n*])

Two classes of faults are identified based on the criticality and the related reactions:

- critical faults
- non-critical faults

Table 1 lists sources for critical faults to be signalled to the FCCU and the type of issued reset.

Table 1. FCCU mapping of critical faults

Critical Fault	Source	Supported Mode	Signal description	Functional reset
CF[0]	RCCU0[0]	LSM	Cores out of lock	long
CF[1]	RCCU1[0]	LSM	Cores out of lock	long
CF[2]	RCCU0[1]	LSM	DMA Multiplexer out of lock	long
CF[3]	RCCU1[1]	LSM	DMA Multiplexer out of lock	long
CF[4]	RCCU0[2]	LSM	PBRIDGEs out of lock	long
CF[5]	RCCU1[2]	LSM	PBRIDGEs out of lock	long
CF[6]	RCCU0[3]	LSM	Bus master interface of FlexRay communication controller crossbar switch (XBAR) out of lock	long
CF[7]	RCCU1[3]	LSM	Bus master interface of FlexRay communication controller crossbar switch (XBAR) out of lock	long
CF[8]	RCCU0[4]	LSM	SRAM arrays out of lock	long
CF[9]	RCCU1[4]	LSM	SRAM arrays out of lock	long
CF[10]	RCCU0[5]	LSM, DPM	PFLASHC out of lock	long
CF[11]	RCCU1[5]	LSM, DPM	PFLASHC out of lock	long
CF[12]	RCCU0[6]	LSM	Bus slave interface of external bus interface (EBI) and DRAM controller (DRAMC) crossbar switch (XBAR) out of lock	long
CF[13]	RCCU1[6]	LSM	Bus slave interface of external bus interface (EBI) and DRAM controller (DRAMC) crossbar switch (XBAR) out of lock	long

Table 1. FCCU mapping of critical faults (continued)

Critical Fault	Source	Supported Mode	Signal description	Functional reset
CF[14]	SWT_0	LSM, DPM	Software watchdog timer not triggered appropriately	long
CF[15]	SWT_1	LSM, DPM	Software watchdog timer not triggered appropriately	long
CF[16]	ECSM_NCE_0	LSM, DPM	Flash memory/SRAM ECC non-correctable error (dual and multiple bit fault)	long
CF[17]	ECSM_NCE_1	LSM, DPM	Flash memory/SRAM ECC not able to correct fault (not correctable error due to dual and multiple bit fault)	long
CF[18]	ADC_CF_0	LSM, DPM	Analogue to digital converter Internal self-test delivers incorrect result (if configured as critical faults)	—
CF[19]	ADC_CF_1	LSM, DPM	Analogue to digital converter Internal self-test delivers incorrect result (if configured as critical faults)	—
CF[20]	STCU	LSM, DPM	Build in Self-test (LBIST, MBIST) delivers incorrect result (if configured as critical faults)	—
CF[21]	Reserved			
CF[22]	SSCM_XFER_ERR	LSM, DPM	SSCM transfer error occurred during STCU configuration loading phase	—
CF[23]	LSM_DPM_ERR_0	LSM, DPM	MCU mode switched between LPM and DPM during runtime (application active)	long
CF[24]	LSM_DPM_ERR_1	LSM, DPM	MCU mode switched between LPM and DPM during runtime (application active)	long
CF[25]	RCCU0[7]	LSM	Bus master interface of crossbar switch 2 (XBAR 2) to crossbar switch (XBAR) out of lock	long
CF[26]	RCCU1[7]	LSM	Bus master interface of crossbar switch 2 (XBAR 2) to crossbar switch (XBAR) out of lock	long
CF[27]	STCU	LSM, DPM	Wrong configuration of STCU as Self-Test Control Unit is active during application	long
CF[28]	DFT_0	LSM, DPM	Reserved for internal test logic monitoring Combination of functional safety critical signals from Test Control Unit (TCU)	long
CF[29]	DFT_1			long
CF[30]	DFT_2			long
CF[31]	DFT_3			long
CF[32]	CFlash_0	LSM, DPM	Code flash memory detected fault during initialization	—
CF[33]	CFlash_1	LSM, DPM	Code flash memory detected fault during initialization	—
CF[34]	DFlash_0	LSM, DPM	Data flash memory detected fault during initialization	—
CF[35]	ADC_CF_2	LSM, DPM	Analogue to digital converter Internal self-test delivers incorrect result (if configured as critical faults)	—
CF[36]	ADC_CF_3	LSM, DPM	Analogue to digital converter Internal self-test delivers incorrect result (if configured as critical faults)	—
CF[37]	JTAG/NEXUS	LSM, DPM	Combination of functional safety critical signals from JTAG and NEXUS Port Controller	long

Table 2 lists all sources for non-critical faults to be signalled to the FCCU and the type of issued reset.

Table 2. FCCU mapping of non-critical faults

Non-critical fault	Source	Signal description	Functional reset
NCF[0]	Core_0 watchdog	Core_0 watchdog prewarning status (p_wrs_core0[0])	long
NCF[1]	Core_1 watchdog	Core_1 watchdog prewarning status (p_wrs_core1[0])	long
NCF[2]	FM_PLL_0	Frequency modulated phase lock loop 0 (FMPLL_0) detected loss of lock	long
NCF[3]	FM_PLL_1	Frequency modulated phase lock loop 1 (FMPLL_1) detected loss of lock	long
NCF[4]	CMU_0	Clock monitoring unit detected a loss of external oscillator (XOSC) clock	long
NCF[5]	CMU_0	Clock monitoring unit detected a system clock (Sysclk) frequency being out of range	long
NCF[6]	CMU_1	Clock monitoring unit detected a motor control clock (MOTC_CLK) frequency out of range	long
NCF[7]	CMU_2	Clock monitoring unit detected a FlexRay clock (FRPE_CLK) frequency out of range	long
NCF[8]	ECSM_ECN_0	ECC 1-bit error correction notification	—
NCF[9]	ECSM_ECN_1	ECC 1-bit error correction notification	—
NCF[10]	ADC_NCF_0	ADC_NCF_0: Analogue to digital converter Internal self-test delivers incorrect result (if configured as non critical faults)	—
NCF[11]	ADC_NCF_1	ADC_NCF_1: Analogue to digital converter Internal self-test delivers incorrect result (if configured as non critical faults)	—
NCF[12]	STCU_NCF	MBIST or LBIST delivers incorrect result (if configured as non critical faults)	—
NCF[13]	Reserved		
NCF[14]			
NCF[15]			
NCF[16]			
NCF[17]			
NCF[18]			
NCF[19]	FLEXR_ECN	ECC 1-bit error correction notification from static RAM (LRAM, DRAM) memory array of FlexRay protocol controller	—
NCF[20]	FLEXR_NCE	ECC not correctable error notification from static RAM (LRAM, DRAM) memory array of FlexRay protocol controller (combination of LRAM and DRAM ECC errors)	—
NCF[21]	MC_ME	Software device reset (reset request originated by MC_ME)	—

Table 2. FCCU mapping of non-critical faults (continued)

Non-critical fault	Source	Signal description	Functional reset
NCF[22]	Reserved		
NCF[23]			
NCF[24]			
NCF[25]	ADC_NCF_0	ADC_NCF_2: Analogue to digital converter Internal self-test delivers incorrect result (if configured as non critical faults)	—
NCF[26]	ADC_NCF_1	ADC_NCF_3: Analogue to digital converter Internal self-test delivers incorrect result (if configured as non critical faults)	—
NCF[27]	Reserved		
NCF[28]			
NCF[29]			
NCF[30]			
NCF[31]			

The FCCU has two external signals, FCCU_F[0] and FCCU_F[1]. In Safe mode, critical errors are reported on these signals. When the device is in reset or unpowered, these outputs are tristated.

FCCU_F[*n*] are intended to be connected to an independent device which continuously monitors these signals. If a failure is detected, the separate device switches to and maintains the system to a Safe $state_{system}$ condition within the FTTI. (for example, the separate device disconnects the MPC567xK device from the power supply).

5.2.1.1 Initial checks and configurations

Besides the possible initial configuration, no CPU intervention is necessary for fault collection and fault reaction.

Safety requirement: [SM_011] System level measures may configure the FCCU to enable all reactions related to faults of peripherals used by the item safety function. [end]

Rationale: Maintain the device in the Safe $state_{system}$ in case of failure.

Implementation hint: The FCCU fault path is enabled by configuring FCCU registers (for example, FCCU_CF_CFG0, FCCU_NCF_CFG0, FCCU_CFS_CFG0, FCCU_NCFS_CFG0, FCCU_NCF_TOE0, and so on).

When a Clock Monitoring Unit (CMU) monitors a FMPLL that is not used, or is not used for functional safety critical modules, error masking and limited internal reaction can be tolerated.

External reaction of the FCCU is always enabled and can not be disabled.

Safety requirement under certain preconditions: [SM_035] If the outputs of the system I/O's need to be forced to a high impedance state upon entering safe mode, MC_ME_SAFE_MC[PDO] is to be set to 1. [end]

If the MPC567xK signals an internal failure via its error out signals (FCCU_F[0:1]), the system can no longer trust the MPC567xK outputs used within the safety function. If an error is indicated, the system has to be able to remain in Safe $state_{system}$ without any additional action from the MPC567xK. Depending on its functionality, the system might disable or reset the MPC567xK as a reaction to the indicated error out.

5.2.1.2 Runtime checks

Safety requirement under certain preconditions: [SM_036] In case MCU switching continuously between a standard operation state and the reset state or Fault State without any device shutdown does not meet the Safe $state_{system}$, item level measures must be implemented. [end]

Implementation hint: Software may be implemented to reduce the likelihood of cycling between a functional and a fault state. For example, in case of periodic non critical faults, the software could clean the respective status and periodically move the device from fault state to normal state. This looping may be avoided.

To prevent permanent cycling between a functional state and a fault state, software will keep track of cleaned faults, stop cleaning and stay in a Safe $state_{MCU}$ instead in case of inacceptable high frequency of necessary fault cleaning. The limit for the number and frequency of clearances is application dependent. This may only be relevant in case continuous switching between a standard operation state and reset state as a failure condition is not a Safe $state_{system}$.

5.2.2 Reset Generation Module (MC_RGM)

A redundant fault notification path is achieved through the use of the Reset Generation Module (MC_RGM) and the Fault Collection and Control Unit (FCCU).

Detected critical errors are forwarded independently to Reset Generation Module (MC_RGM) and Fault Collection and Control Unit (FCCU). Additionally, the state of the MC_RGM is forwarded to the FCCU and the FCCU forward an additional reset request to the MC_RGM. This decreases the likelihood of common mode failures on the functional safety path and it ensures reaction to failures in all cases. Even if FCCU would fail, a reset would be generated by MC_RGM to enter Safe $state_{MCU}$.

NOTE

Safe mode and reset use the MC_RGM as a common shared resource and therefore prone to cascading faults and are not independent channels to communicate an fault.

5.2.2.1 Initial checks and configurations

Safety requirement: [SM_012] It is good practice to configure a second failure notification channel to communicate critical application faults redundantly. [end]

Recommendation: To have a redundant notification path, both MC_RGM and FCCU may be configured to react to critical application faults.

Rationale: To have two notification paths in case of an error.

Implementation hint: To enable critical events to trigger a reset sequence, MC_RGM_FERD shall be set to zero. If particular events are excluded, MC_RGM_FEAR shall be configured to generate an alternate request in these cases.

To trigger a reset of the device by software, the MC_ME_MCTL[TARGET_MODE] shall be used. Writing MC_ME_MCTL[TARGET_MODE] = 0000b causes a functional reset, and writing MC_ME_MCTL[TARGET_MODE] = 1111b causes a destructive reset.

Recommendation: It is recommended that the peripheral access control in the PBRIDGE_*n* be configured by application software to prohibit access to the MC_RGM_PRST[*n*] (individual module reset programming model).

5.2.3 Self Test Control Unit (STCU)

The STCU executes built in self test (LBIST, MBIST) and gives reaction to detected faults by triggering a Non-Critical Fault (NCF) to the FCCU (see Chapter 50 “Self-Test Control Unit (STCU)” in the *Qorivva MPC5675K Microcontroller Reference Manual* for details).

5.2.3.1 Initial checks and configurations

The STCU does not require any configuration performed by application software.

Safety requirement under certain preconditions: [SM_037] When built in self test (for example, LBIST, MBIST, ADBIST) circuits of the MPC567xK are used as functional safety integrity measure (for example, to detect random faults, latent fault detection, and single-point fault detection) in a functional safety system, functional safety integrity measures on system level shall be implemented ensuring STCU integrity during/after STCU initialization but before executing a safety function. [end]

Rationale: The STCU’s correct behavior shall be verified by checking the expected results by software.

Implementation hint: System (application) level software shall carry out checking of STCU for ensuring STCU integrity. See the section 50.1.2 “Integrity software operations” within Chapter 50 “Self-Test Control Unit (STCU)” chapter in the *Qorivva MPC5675K Microcontroller Reference Manual* for details.

Implementation hint: The Integrity software shall confirm that all MBISTs and LBISTs finished successfully with no additional errors flagged.

This software confirmation prevents a fault within the STCU itself from incorrectly indicating that the built in self-test passed.

This is an additional functional safety layer since the STCU propagates the LBIST/MBIST and internal faults using the CF signals of the FCCU. So, reading STCU_LBS, STCU_LBE, STCU_MBSL, STCU_MBSH, STCU_MBEL, STCU_MBEH and STCU_ERR registers helps to increase the STCU auto-test coverage.

Implementation hint: The STCU shall be configured (for example, in test flash memory) to execute the LBIST and MBIST before activating the application safety function (see section 50.4.3.2 “STCU Configuration Register (STCU_CFG)” in Chapter 50 “Self-Test Control Unit (STCU)” and section 8.2.5 “Test sector” in Chapter 8 “Flash Memory” of the *Qorivva MPC5675K Microcontroller Reference Manual*). Also, the application note *MPC5675K Test and Shadow Flash Parameters for ADC Self Test, MBIST, and LBIST* (Document Number: AN4422) describes the test flash memory addresses to enable LBIST and MBIST during power-up (destructive reset).

5.2.4 Temperature Sensor (TSENS)

Recommendation: To reduce the likelihood of common mode failure(s), the effects of increasing temperature, for example, due to random hardware fault(s), may be controlled on system level.

Recommendation: The potential for over-temperature operating conditions need to be reduced by appropriate system level measures. Possible measures could include:

- Actuation of the functional safety shut-off via thermal fuse.
- Several levels of over-temperature sensing and alarm triggering.
- Connection of forced-air cooling and status indication.

Implementation hint: A temperature sensor monitors the substrate temperature to detect over-temperature conditions before they cause common mode failure (for example, faults due to over-temperature causing identical erroneous results from both cores). The maximum operating junction temperature is specified in the device data-sheet. The sensor output is forwarded to the analog acquisition channel for measurement (temperature sensor mapped to channel 15 of ADC_0). As no redundant temperature sensor is implemented, respective item level measures may be required to verify the integrity of the measured temperatures to detect possible malfunctions of the sensor itself.

5.2.4.1 Initial checks and configurations

Recommendation: If using the temperature sensor as common mode fault measure, during or after initialization but before executing any safety function the temperature sensor shall be read by software and rationalize if temperature is plausible and within operational temperature range.

However, nothing prohibits reading the temperature sensor during execution of the safety function (application run time).

Rationale: A means of assessing functionality of the temperature sensor

Implementation hint: Temperature sensor reading can be validated by implementing an additional temperature sensor on item level. Reading the temperature value of both sensors and comparing these two if being in the same temperature range would enable detection of latent faults of temperature sensor.

Safety requirement under certain preconditions: [SM_038] If using the internal temperature sensor and an external temperature sensor as common mode fault measure, to improve common mode failure(s) robustness, the temperature reading should not use the same analog to digital converter (ADC) to reduce common resources shared for the two temperate measurement channels. [end]

5.2.4.2 Runtime checks

Rationale: To detect over-temperature potentially causing common mode faults

Implementation hint: The temperature should be acquired by software within FTTI during the safety function being active (application run time) to improve the functional safety integrity regarding common mode faults (over temperature). In case of a temperature not within the operational range, application software shall switch the item to a Safe $state_{system}$.

To set a proper threshold the system integrator shall consider the temperature sensor accuracy (see the *Qorivva MPC5675K Microcontroller Data Sheet*, and the *Qorivva MPC5675K Microcontroller Reference Manual* for the on TSENS_*n* implementation in relation to the ADC).

5.2.5 Software Watchdog Timer (SWT)

The Software Watchdog Timer (SWT) is a peripheral module that can prevent system lockup in situations such as software getting trapped in a loop or if a bus transaction fails to terminate. The objective of the SWT is to detect an erroneous program sequence. The refresh of the software watchdog timer occurs within a specified timeout period. If not, according to SWT configuration, a reset can be generated immediately or the SWT can first generate an interrupt and re-initialize the SWT with the timeout period. Only if the service sequence is not written before the second consecutive timeout, the SWT generates a reset.

The SWT down counter is always driven by the IRCOSC clock.

Two service procedures are available:

- a fix service sequence represented by a write of two fix values (0xA602, 0xB480) to the SWT service register. Writing the service sequence reloads the internal down counter with the timeout period.
- The second is based on a pseudo-random key computed by the SWT every time it is serviced and which is written by the software on the successive write to the service register. The watchdog can be refreshed only if the key calculated in hardware by the watchdog is equal to the key provided by software which may calculate the key in one or more procedure/tasks (so called signature watchdog).

5.2.5.1 Runtime checks

Rationale: To detect an erroneous program sequence

Implementation hint: Control flow monitoring can be implemented using SWT. However, other control flow monitoring approaches that do not used the SWT may also be used. In case using SWT, it shall be enabled and configuration registers shall be hard-locked against manipulation by application software. The SWT time window settings shall be set to a value less than the FTTI. Detection latency shall be smaller than FTTI. During/after initialization but before executing any safety function any safety function, the software shall check that the SWT is enabled by checking the SWT control register (SWT_CR).

To enable the SWT and to hard-lock the configuration register, the WEN and HLK flags of the SWT control register (SWT_CR) shall be asserted. The timeout register (SWT_TO) shall contain a 32-bit value

that represents a timeout less than the FTTI. If Windowed mode and Keyed Service mode (two pseudorandom key values used to service the watchdog) are enabled, it is possible to reach a temporal and logical flow monitoring.

5.2.6 Redundancy Control Checking Unit (RCCU)

The task of the RCCU unit is to perform a cycle-by-cycle comparison of the outputs of the modules included in the SoR. The SoR is the logical part of the device that contains all the modules that are replicated for functional safety reasons. The RCCU is able to detect any mismatch between the outputs of two replicated modules. The error information is forwarded to the Reset Generation Module (MC_RGM) and to the Fault Collection and Control Unit (FCCU). The RCCUs are automatically enabled when MPC567xK is in the LSM mode.

5.2.6.1 Initial checks and configurations

The use of the RCCU is indispensable. This is automatically managed by the MPC567xK device. RCCU cannot be disable by application software.

5.2.7 Cyclic Redundancy Checker Unit (CRC)

The CRC module offloads the CPU in computing a CRC checksum. The CRC has the capability to process two interleaved CRC calculations. The CRC module may be used to detect erroneous corruption of data during transmission or storage. The CRC takes as its input a data stream of any length and calculates a 32-bit output value (signature). The contents of the configuration registers of the functional safety-related modules shall be checked within the FTTI.

5.2.7.1 Runtime checks

Parts of the MCU configuration registers do not provide the functional safety integrity IEC 61508 series and ISO 26262 requires for high functional safety integrity targets. This relates to systematic faults, for example, in the application software, as also regarding random faults, for example, single event upsets.

Safety requirement: [SM_014] System level measures verifies the content of the MCU configuration registers of the modules involved with the safety function to detect erroneous corruption of the content. [end]

The CRC module offloads the CPU in computing a CRC checksum. The CRC has the capability to process two different CRC calculations at the same time.

Rationale: To check the integrity of the module configuration.

Implementation hint: The CRC module offloads the CPU in computing a CRC checksum. The CRC has the capability to process two different CRC calculations at the same time. To verify the content of the MCU configuration registers of the modules involved with the safety function, the CRC module may be used to calculate a signature of the content of the registers and compare this signature with a value calculated during development.

Alternatively, the CPU could be used instead of the CRC module to check that the value of the configuration registers have not changed. However, using the CRC module is more effective.

Implementation hint: The CRC module could be used to detect data corruption during transmission or storage. The CRC takes as its input a data stream of any length and calculates a 32-bit signature value.

Implementation hint: The expected CRC of the configuration registers of the modules involved with the safety function shall be calculated offline. When the safety function is active (application run time), the same CRC value shall be calculated by the CRC module within the FTTI. To unload the CPU, the eDMA module can be used to support the data transfer from the registers under check to the CRC module. The result of the runtime computation is then compared to the predetermined value.

The application shall include detection, or protection measures, against possible faults of the CRC module only if the CRC module is used as safety integrity measure or within the safety function.

Implementation hint: An alternative approach would use the DMA to reinitiation the content of the configuration registers of the modules involved with the safety function within the respective FTTI when the safety function is active (application run time). This approach may require additional measures to detect permanent failures (not fixed by reinitiation).

5.2.7.1.1 Implementation details

The eDMA and CRC modules should be used to implement these safety integrity measures to unload the CPU.

NOTE

Caution: The signature of the configuration registers is computed in a correct way only if these registers do not contain any volatile status bit.

5.2.7.1.1.1 <*module*>_SWTEST_REGCRC

The following safety integrity functions for register configuration checks are used in this document:

- ETIMER0_SWTEST_REGCRC

  The eTimer_0 configuration registers are read and a CRC checksum is computed. The checksum is compared with the expected value.
- ETIMER1_SWTEST_REGCRC

  The eTimer_1 configuration registers are read and a CRC checksum is computed. The checksum is compared with the expected value.
- ETIMER2_SWTEST_REGCRC

  The eTimer_2 configuration registers are read and a CRC checksum is computed. The checksum is compared with the expected value.
- SIUL_SWTEST_REGCRC

  The configuration registers of the SIUL are read and a CRC checksum is computed. The checksum is compared with the expected value.
- FLEXPWM0_SWTEST_REGCRC

The FlexPWM_0 configuration registers are read and a CRC checksum is computed. The checksum is compared to the expected value.

- FLEXPWM1_SWTEST_REGCRC

  The FlexPWM_1 configuration registers are read and a CRC checksum is computed. The checksum is compared to the expected value.

- FLEXPWM2_SWTEST_REGCRC

  The FlexPWM_2 configuration registers are read and a CRC checksum is computed. The checksum is compared to the expected value.

- ADC0_SWTEST_REGCRC

  The ADC_0 configuration registers are read and a CRC checksum is computed. The checksum is compared to the expected value.

- ADC1_SWTEST_REGCRC

  The ADC_1 configuration registers are read and a CRC checksum is computed. The checksum is compared to the expected value.

- ADC2_SWTEST_REGCRC

  The ADC_2 configuration registers are read and a CRC checksum is computed. The checksum is compared to the expected value.

- ADC3_SWTEST_REGCRC

  The ADC_3 configuration registers are read and a CRC checksum is computed. The checksum is compared to the expected value.

5.2.8 Internal RC Oscillator (IRCOSC)

The IRCOSC has a nominal frequency of 16 MHz, but a frequency accuracy of ± 6 % (after trimming) over the full voltage and temperature range has to be taken into account. It does not require any external crystal. Functional safety-related modules which use the clock generated by the internal RC oscillator are: FCCU, CMU, and SWT. In rare case of RC clock failure, these modules stop working.

5.2.8.1 Initial checks and configurations

The frequency meter of the CMU_0 shall be exploited to check the availability and frequency of the internal IRCOSC. This feature allows to measure the IRCOSC frequency using the external oscillator clock as known one (IRC_SW_CHECK).

Safety requirement: [SM_015] The IRCOSC frequency is measured and compared to the expected frequency of 16 MHz (± 6 % accuracy). This test is performed after power-up before executing any safety function. CMU_CSR[FSM] flag is checked and is reset to 0 when enabling the safety function. [end]

Rationale: To check the integrity of the IRCOSC

Please refer to section “Frequency meter” in the *Qorivva MPC5675K Microcontroller Reference Manual*.

NOTE

If the IRCOSC is not operating due to a fault, the measurement of the IRCOSC frequency will never complete and the CMU_CSR[FSM] flag will remain set. The application may need to manage detecting this condition. For example, implementing a software watchdog which monitors the CMU_CSR[FSM] flag status.

5.2.8.2 Runtime checks

Recommendation: To increase the fault detection, this functional safety integrity measure can be executed once per FTTI.

5.2.9 Frequency-Modulated PLL (FMPLL)

MPC567xK consists of two Frequency-Modulated Phase-Locked-Loops (FMPLL) to generate high speed clocks. Each FMPLL provides a loss of lock error indication that is routed to the MC_RGM and the FCCU (NCF[2], NCF[3]). In case of no lock, the “system clock” can be driven by the RC oscillator, a FMPLL fault is considered as a Non-Critical Fault. Glitches which may appear on the crystal clock are filtered (low-pass filter) by the FMPLL. The FMPLL dedicated to the system clock is a modulated PLL to reduce EMI and it’s clock is distributed to the processing hardware elements. The auxiliary clock from the second FMPLL is instead distributed to the peripherals that require precise timing (FlexRay, eTimer, FlexPWM) and it’s clock is not modulated.

Since in case of fault the “system clock” can be driven by the IRCOSC, a FMPLL fault is considered as a Non-Critical Fault (NCF).

Implementation hint: The pll_fail output are measured (FMPLL_0_CR[PLL_FAIL_MASK] = 0 and FMPLL_1_CR[PLL_FAIL_MASK] = 0). To enable the MC_RGM input related to FMPLL loss of clock, the registers MC_RGM_FERD and MC_RGM_FEAR shall be configured.

5.2.9.1 Initial checks and configurations

After system reset, the external crystal oscillator is powered down and the FMPLL deactivated. Software shall enable the oscillator.The MPC567xK uses after system reset the internal RC oscillator clock (IRCOSC) as clock source (see “Oscillators” chapter in the *Qorivva MPC5675K Microcontroller Reference Manual* and Section 5.2.8, Internal RC Oscillator (IRCOSC)).

Safety requirement: [SM_016] Before executing any safety function, a high quality clock (low noise, low likelihood for glitches) based on an external clock source shall be configured as the system clock of the MPC567xK. [end]

Rationale: Since the IRCOSC is used by the CMUs as reference to monitor the output of the two PLLs, it cannot be used as input of these PLLs.

Implementation hint: The FMPLLs shall be configured to be used the external oscillator (XOSC) as a clock reference or an external provided clock reference. In general MC_CGM_AC3_SC[SELCTL] and MC_CGM_AC4_SC[SELCTL] shall be set to ‘1’.

Safety requirement under certain preconditions: [SM_039] When clock glitches endanger the system level functional safety integrity measure respective functional safety relevant modules shall be clocked with a FMPLL generated clock signal, as the PLL includes respective filters to reduce the likelihood of clock glitches due to external disturbances. Alternatively a high quality external clock having low noise and low likelihood of clock glitches shall be used. [end]

Rationale: To reduce the impact of glitches stemming from the external crystal and its hardware connection to the MCU.

Implementation hint: This requirement is fulfilled by appropriately programming the Clock Generation Module (MC_CGM) and Mode Entry Module (MC_ME).

During/after initialization but before executing any safety function, application software has to check that the MPC567xK uses the FMPLL clock as “system clock” (`PLL_SW_CHECK`).

Implementation hint: Application software can check the current “system clock” by checking the MC_ME_GS[S_SYSCLK] flag. MC_ME_GS[S_SYSCLK] = 4 indicates that the FMPLL clock is being used as the system clock.

5.2.10 Clock Monitor Unit (CMU)

The main task of the CMU is to supervise the integrity of various clock sources. The following clocks in the MPC567xK are supervised by three CMUs:

- system FMPLL
- secondary FMPLL
- 16 MHz internal RC oscillator (IRCOSC)
- external crystal oscillator (XOSC)

All three CMUs use the IRCOSC (16 MHz internal oscillator) as the reference clock for independent operation from the monitored clocks. Their purpose is to check for error conditions due to:

- loss of clock from external crystal (XOSC)
- loss of reference (IRCOSC)
- PLL clock out of a programmable frequency range (frequency too high or too low)
- loss of PLL clock

The three CMUs supervise the frequency range of various clock sources. In case of abnormal behavior, the information is forwarded to the FCCU as non-critical faults:

- CMU_0 monitors the clock signal of the SoR (NCF[5]) and the clock from the crystal oscillator (NCF[4])
- CMU_1 monitors the clock signal used by the Motor Control related peripherals (for example, eTimer, FlexPWM, CTU and ADC) (NCF[6])
- CMU_2 monitors the clock signal for the protocol engine of the FlexRay module, FlexCAN and other parts requiring non-modulated frequency (NCF[7])

5.2.10.1 Initial checks and configurations

Safety requirement: [SM_017] The following supervisor functions are required: [end]

- Loss of external crystal oscillator clock
- FMPLL frequency higher than the (programmable) upper frequency reference
- FMPLL frequency lower than the (programmable) lower frequency reference

Rationale: To monitor the integrity of the clock signals

Recommendation: CMU may be used for each clock that is used by a functional safety relevant module. Application software shall check that the CMUs are enabled and their faults managed by the FCCU.

Implementation hint: In general, the following two application-dependent configurations shall be executed before CMU monitoring can be enabled.

- The first configuration is related to the crystal oscillator clock (XOSC_CLK) monitor of CMU_0. Software configures CMU_0_CSR[RCDIV] to select an IRCOSC divider. The divided IRCOSC frequency is compared with the XOSC_CLK.
- The second configuration is related to other clock signals being monitored. The high frequency reference (CMU_*n*_HFREFR_A[HFREF_A]) and low frequency reference (CMU_*n*_LFREFR_A[LFREF_A]) is configured depending on the SoR (CMU_0), motor control related peripherals (CMU_1) and FlexRay (CMU_2) clock frequencies.

Once the CMUs are configured, clock monitoring will be enabled when software writes CMU_*n*_CSR[CME_A] = 1.

5.2.11 Mode Entry (MC_ME)

Safety requirement under certain preconditions: [SM_065] If application uses Low Power (LP) mode, it is required to monitor the duration in LP mode. If the system does not wake up within a specified period, the system will be reset by the monitoring circuitry. [end]

Implementation hint: The SWT may provide the time monitoring.

Safety requirement under certain preconditions: [SM_066] If application uses Low Power (LP) mode, it is required that application software performs a test of entry and exit to and from LP mode at startup. [end]

Rationale: To overcome faults in the wakeup and interrupt inputs to the MC_ME if the application uses Low Power mode.

5.2.12 Power Management Controller (PMC)

The Power Management Controller (PMC) manages the supply voltages for all modules on the device. This unit includes the internal regulator and ballast for the logic power supply (1.2 V) and a set of voltage monitors. Particularly, it embeds low voltage detectors (LVD) and high voltage detectors (HVD). If one of the monitored voltages goes below (LVD) or above (HVD) a given threshold, a destructive reset is initiated to control erroneous voltages before these cause a common mode failure.

The supplies monitored by the PMC are summarized in Table 3.

- Flash memory (VDDFLASH) – LVD_VDDFLASH[1] (LVD_FLASH)
- I/O (VDDIO) – LVD_VDDIO[1]
- Analogue to digital converter (VDDADC) - LVD_VDDDADC[1]
- VREG (VDDREG) – LVD_VDDREG[1]
- 1.2 V digital core supply (VDD)
  - LVD_VDD[1]
  - HVD_VDD[1]

Table 3. PMC Monitored Supplies

Detector Type	Detector Name	Voltage Monitored	Comments
Flash memory LVD	LVD_VDDFLASH	3.3 V flash memory supply	—
I/O LVD	LVD_VDDIO	3.3 V I/O supply	—
VREG LVD	LVD_VDDREG	3.3 V VREG supply	—
Core main LVD	LVD_VDD	1.2 V core supply	—
Core main HVD	HVD_VDD	1.2 V core supply	—

Over voltage of any 3.3 V supply shall be monitored externally being described in Section 4.1.3, Power Supply Monitor (PSM).

5.2.12.1 1.2 V supply supervision

Voltage detectors LVD_VDD and HVD_VDD monitor the digital (1.2 V) core supply voltage for over and under voltage in relation to a reference voltage. Figure 10 depicts the logic scheme of the voltage detectors. In case the core main voltage detector detects over or under voltage during normal operation of the MPC567xK, a destructive reset is triggered.

Figure 10. Logic scheme of the core voltage detectors

By this means, a failing external ballast transistor (stuck-open, stuck-closed) is also detected.

Safety requirement under certain preconditions: [SM_040] When the system requires respective robustness regarding 1.2 V over voltage failures, the external VREG mode is preferably selected. The internal VREG mode uses a single pass transistor and, therefore, over voltage can not be shut off redundantly. [end]

1. Abbreviations used in the Chapter 44 “Power Management Controller (PMC)” section of the *Qorivva MPC5675K Microcontroller Reference Manual*.

Rationale: To enable system level measures to detect or shut down the supply voltage in case of an destructive (multiple-point faults) 1.2 V over voltage incident.

Implementation hint: To reduce the likelihood of destructive damage due to a stuck-closed external ballast transistor (item/system level component), it may be necessary to implement two ballast transistors sequential as a system level functional safety integrity measure. This will load the regulator with two ballast transistors. In order to use two ballast transistor a ~ 30 % Cg (or smaller, transistor gate capacity) should be selected. Alternative the digital (1.2 V) core supply voltage may be monitored externally and the power supply shut-down in case of an over voltage. Alternatively an external 1.2 V HVD may detect over voltage and shut down the 3.3 V supply voltage.

Safety requirement: [SM_019] Parts of the voltage detectors LVD_VDD and HVD_VDD monitoring the digital (1.2 V) core supply voltage for over and under voltage may not provide the functional safety integrity IEC 61508 series and ISO 26262 requires for high functional safety integrity targets as no functional safety integrity measures to detect latent faults within voltage detectors LVD_VDD and HVD_VDD is integrated. System level functional safety assessments have to validate this is compliant to the item level functional safety requirements and respectively if required add additional item level functional safety integrity measures. [end]

5.2.12.2 3.3 V supply supervision

Voltage detectors LVD_VDDIO, LVD_VDDVREG, and LVD_VDDFLASH monitor the 3.3 V supply for under voltage in relation to a reference voltage. Figure 11 depicts the logic scheme of the voltage detectors. In case a single LVD detects under voltage during normal operation of the MPC567xK, a destructive reset is triggered.

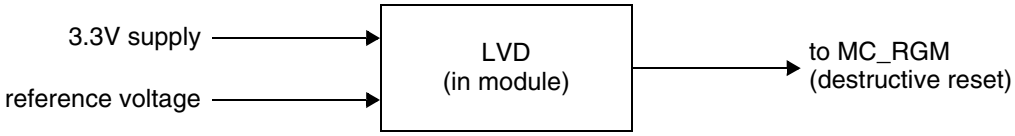


Figure 11. Logic scheme of the 3.3 V voltage detectors

Safety requirement: [SM_020] Parts of the voltage detectors LLVD_VDDIO, LVD_VDDVREG, and LVD_VDDFLASH monitoring the supply voltage for under voltage may not provide the functional safety integrity IEC 61508 series and ISO 26262 requires for high functional safety integrity targets as no functional safety integrity measures to detect latent faults within voltage detectors LVD_VDD and HVD_VDD are integrated. System level functional safety assessments have to validate this is compliant to the item level functional safety requirements and respectively if required additional item level functional safety integrity measures may be implemented. [end]

5.2.13 Memory Protection Unit (MPU)

The Memory Protection Unit (MPU) provides hardware access control for all memory references generated in a device. Using preprogrammed region descriptors that define memory spaces and their associated access rights for each XBAR bus master (cores, DMAs, FlexRay protocol controller, Fast Ethernet protocol controller), the MPU concurrently monitors all system bus transactions (including those

initiated by the eDMA, fast Ethernet protocol controller, or FlexRay protocol controller) and evaluates the appropriateness of each transfer.

Memory references that have sufficient access control rights are allowed to complete, while references that are not mapped to any region descriptor or have insufficient rights are terminated with a protection error response. The MPU implements a set of program-visible region descriptors that monitor all system bus addresses. The result is a hardware structure with a two-dimensional connection matrix, where the region descriptors represent one dimension and the individual system bus addresses and attributes represent the second dimension.

5.2.13.1 Initial checks and configurations

Safety requirement under certain preconditions: [SM_041] If DPM is selected, system level functional safety integrity measures will cover replicated bus masters, modifying the replicated resources may be required. [end]

Safety requirement under certain preconditions: [SM_042] If non-replicated bus masters (for example, FlexRay and FEC) are used, system level functional safety integrity measures must cover bus operations to reduce the likelihood of replicated resources being erroneously modified. [end]

Rationale: Access restriction is protection against unwanted read/write accesses to some predefined memory mapped address locations.

Implementation hint: The MPU shall be used to ensure that only authorized software routines can configure modules and all other bus masters (eDMA, core, FlexRay protocol controller, Ethernet protocol controller) can access only their allocated resources according to their access rights. For the non-replicated master FlexRay, a correct MPU setup is highly recommended.

5.2.14 Memory Management Unit (MMU)

The software managed first level (L1) unified Memory Management Unit (MMU) provides the virtual to physical address translation. The Power ISA embedded category architecture defines that a process ID (PID) value is associated with each effective address (instruction or data) generated by the processor. At the Power ISA embedded category level, a single PID register within each core (replicated) is defined as a 32-bit register, and it maintains the value of the PID for the current process. This PID value is included as part of the virtual address in the translation process (PID0). For the e200z7 MMU, the PID is 8 bits in length. The most-significant 24 bits are not implemented and read as 0. The p_pid0[0:7] interface signals indicate the current process ID. An operating system may restrict access of a process (PID) to virtual pages by selectively granting permissions for user mode read, write, and execute, and supervisor mode read, write, and execute on a per page basis. These permissions can be set up for a particular system (for example, program code might be execute-only, data structures may be mapped as read/write/no-execute) and can also be changed by the operating system based on application requests and functional safety policies. The MMU provides a hardware access control for all memory references generated in a core. Using preprogrammed region descriptors that define memory spaces and their associated access rights for each process identifier or task identifier (PID), the MMU monitors all core transactions (excluding those initiated by the eDMA, Ethernet protocol controller, or FlexRay protocol controller) and evaluates the appropriateness of each transfer.

Memory references that have sufficient access control rights are allowed to complete, while references that are not mapped to any region descriptor or have insufficient rights are terminated with a protection error response. The MMU implements a set of program-visible region descriptors that monitor all addresses within a core. The result is a hardware structure with a two-dimensional connection matrix, where the region descriptors represent one dimension and the individual system bus addresses and attributes represent the second dimension.

The MMU may be used to encapsulate (address domain) software routines having different functional safety integrity levels. As the MMU controls accesses using the PID, routines owing different PID are controlled not to interfere in the address range (resources).

5.2.14.1 Initial checks and configurations

Access restriction at the MMU level is protection against unwanted read/write accesses to some predefined memory mapped address locations by specific software routines (processes).

Safety requirement under certain preconditions: [SM_043] Software routines developed according to the requirements of different ASIL requirements must be included in the address domain to reduce the likelihood of interference between each other. This is especially true if software with no requirement to comply with ISO 26262 (QM) is executed together with software requiring a high safety integrity. [end]

Rationale: Access restriction at the MMU level is protection against unwanted software (process) read/write accesses to some predefined memory mapped address locations.

Recommended: The MMU may be used to ensure that only authorized software routines (processes) can configure modules and access private resources. All other software routines can access only their allocated resources according to their access rights.

5.2.15 Performance Monitor Counter (PMC)

The performance monitor counter registers (PMC0, PMC1, PMC2, PMC3) are 32-bit counters that can be programmed to generate overflow event signals when they overflow. Each counter is enabled to count up to 128 core events. The performance monitor interrupt is triggered by an enabled condition or event. The enabled condition or events defined for the core may be a PMCn overflow condition occurs. Events able to be counted:

- Processor cycles: Every processor cycle not in waiting, halted, stopped states and not in a debug session.
- Instructions completed: Every completed instructions. 0, 1, 2, or 3 per cycle.
- Instruction words fetched: Fetched instruction words. 0, 1, or 2, 3, or 4 per cycle. (note that an instruction word may hold 1 or 2 instructions, or 2 partial instructions when fetching from a VLE page)
- Branch instructions completed: Completed branch instructions, includes branch and link type instructions
- Branch and link type instructions completed: Completed branch and link type instructions
- Conditional branch instructions completed: Completed conditional branch instructions

- Taken Branch instructions completed: Completed branch instructions which were taken. Includes branch and link type instructions.
- Taken Conditional Branch instructions completed: Completed conditional branch instructions which were taken.
- Load instructions completed: Completed load, load-multiple type instructions
- Store instructions completed: Completed store, store-multiple type instructions
- Integer instructions completed: Completed simple integer instructions
- Multiply instructions completed: Completed Multiply instructions
- Divide instructions completed: Completed Divide instructions
- Number of return from interrupt instructions: Includes all types of return from interrupts
- Cycles decode stalled due to no instructions available: No instruction available to decode
- Cycles issue stalled: Cycles the issue buffer is not empty but no instructions issued
- Cycles branch issue stalled: Branch held in decode awaiting resolution
- Cycles execution stalled waiting for load data: load stalls
- Interrupts taken
- External input interrupts taken
- Watchdog timer interrupts taken
- Watch point occurs: assertion of jd_watchpt0 detected
- External input interrupt latency cycles: Instances when the number of cycles between request for interrupt asserted (but possibly masked/disabled) and redirecting fetch to external interrupt vector exceeds threshold.
- …

Performance counter enable the quantification of interferences (CPU performance domain) of different processes running simultaneously on the cores. The performance counter may be used to encapsulate (core performance domain) software routines having different functional safety integrity levels. As the performance counter quantify the interference of a running task with other pseudo parallel executed task (for example, interrupts) the interference in the CPU performance domain may be monitored for example, by the operating system.

Performance counter enable to trace characteristic signature of functional safety relevant processes. This enables the implementation of a logic flow control based on instruction statistics. Deterministic counter event may for example, be used to generate the signature (pseudo random number) to trigger the on-chip watchdog.

5.2.15.1 Initial checks and configurations

Safety requirement under certain preconditions: [SM_044] Software routines developed according to the requirements of different ASIL requirements shall be encapsulated in address domain to reduce the likelihood of interference. This is specifically true if QM software is executed together with software requiring high safety integrity. [end]

Rationale: CPU performance monitoring is protection against unwanted software (process) core performance interference due to resource conflicts.

Recommendation: The performance counter may be used to ensure that functional safety relevant software routines (processes) get assigned to an appropriate CPU performance within the time slot assigned to.

Implementation hints: Performance counter may be used to monitor the correct statistical instruction count of the individual program sections. The correct statistical instruction pattern of the individual program sections is monitored using hardware (performance counter) and compared to expected values by software.

5.2.16 Built-in Hardware Self Tests (BIST)

Built-in hardware self-test (BIST) or built-in test (BIT) is a mechanism that permits circuitry to test itself. Hardware supported BIST is used to speed-up self-test and reduce the CPU load. As hardware assisted BIST is often destructive, it shall be executed ahead or after a reset (destructive reset or external reset).

Not every fault expresses itself immediately. For example, a fault may remain unnoticed if a component is not used or the context is not causing an error or the error is masked.

If faults are not detected over a long time (latent faults), they can pile up once they propagate. ISO 26262 requires 90% latent-fault metric for ASIL D, 80% for ASIL C, and 60% for ASIL B. Typically hardware assisted BIST is therefore used as safety integrity measure to detect latent faults.

The MPC567xK is equipped with a Built-in hardware self-test:

- System SRAM (MBIST, executed at boot-time, latent fault measure)
- Logic (LBIST, executed at boot-time, latent fault measure)
- ADC (PBIST, executed during boot or executed at least once per FTTI), latent fault measure and single-point failure measure)
- Flash memory array integrity self check (executed at boot-time, latent fault measure)
- Flash memory margin read (executed after every programming operation or executed at least once per FTTI, latent fault measure and single-point failure measure)
- Flash memory: ECC logic check (executed at least once per FTTI, single-point failure measure)

Boot-time test (MBIST, LBIST) are performed after the occurrence of a destructive or external reset, unless they are disabled. All boot-time tests are executed before application software enables a safety function. If failed, chip will remain in Safe state$_{MCU}$.

All tests may be performed without dedicated external test hardware.

The following safety integrity measure validates the ECC fault signalling and is executed by software to detect single-point faults, although no built-in hardware support is used.

- Flash memory: ECC Fault Report Check: Software is required to read from the flash memory a set of test patterns (provided by Freescale) to test the integrity of faults reported by the ECC logic and captured in the ECSM and FCCU (shall be performed once per FTTI).

5.2.16.1 MBIST

The SRAM BIST test (MBIST) runs during initialization (during boot), but some software actions are required (see Section 5.2.3, Self Test Control Unit (STCU)).

5.2.16.2 LBIST

The logic BIST test (LBIST) runs during initialization (during boot), but software actions are required (see Section 5.2.3, Self Test Control Unit (STCU)).

5.2.16.3 Flash memory array integrity self check

The flash memory array integrity self check runs in flash memory user test mode and is initiated by software and the result is checked by software (see Chapter 5.2.23, “Flash memory”).

5.2.16.4 Flash memory margin read

The flash memory margin reads may be activated to increase the sensitivity of the array integrity self check. It may be enabled in flash memory user test mode and is initiated by software (see Section 5.2.23.3.1, FLASH_SW_ECCREPORT).

5.2.16.5 Flash memory ECC logic check

The flash memory ECC logic check runs in flash memory user test mode. It is executed in software and supported by hardware (see Section 5.2.23.3.2, FLASH_SW_ECCTEST).

5.2.16.6 Flash memory ECC fault report check

The flash memory ECC fault report check is executed in software (refer to Chapter 5.2.23, “Flash memory”).

5.2.16.7 Peripheral Built-In Self-Test (PBIST)

The ADC BISTs run during initialization (during boot) and optionally during normal operation, but software actions are required run those tests (see Section 5.2.33, Analog to Digital Converter (ADC)).

5.2.17 Error correction (ECC, ECSM)

On MPC567xK, no dedicated ECC module exists, since ECC functionality is located in or near the different memory modules and might vary slightly depending on the needs (and size) of the storage. It is used to detect data corruption in memory and (for SRAM only) address corruption.

The ECC module can correct all single-bit errors (single-bit error correction, SEC), detects all dual-bit faults (double-bit error detection, DED), and detects several multiple bits errors (affecting more than two bits). For system SRAM, addressing information is included in the calculation and evaluation of the ECC to also detect addressing failure of the SRAM arrays. Detected single-bit addressing failures are not corrected. Instead, they are treated and reported as detected multi-bit faults.

The following ECC protection is available:

- 64 bits of flash memory are protected by 8 bits for ECC
- 32 bits of SRAM and 16 address lines are protected by 7 bits for ECC

ECC is automatically calculated during memory write accesses and is checked and faults care corrected while reading memory.

In case there is a SBE due to data corruption, the ECC module corrects the read data. Optionally, an interrupt for checking the address of last corrected data can be generated. The corresponding information is stored in the Error Correction Status Module (ECSM). The ECSM controls the ECC configuration and reporting for the platform memories (flash memory and SRAM).

If there is a double-bit (or detected multiple-bit) fault, both the FCCU and MC_RGM modules assert the error out signal(s), then reset the MPC567xK.

The ECC module may be source of single-point faults (erroneous modification of fault free data) or latent fault (no correction in case of a single-bit data fault). For this reason, SRAM ECC modules are implemented redundantly (multiple instances), for example, integrated in the redundant Static RAM Controller (SRAMC).

The flash memory module ECC algorithm supports the following features to include catastrophic fault models:

- **All ‘0’s**
  - **Error**—The All 0 error algorithm detects as a Double-bit ECC error any word in which all 72 bits (code flash memory) or 39 bits (data flash memory) are all 0.
- **All ‘1’s**
  - **No Error**—The All 1 no error algorithm detects as valid any word read on a just erased sector in which all 72 bits (code flash memory) or 39 bits (data flash memory) are all 1. This option allows performing a blank check after a sector erase operation.

NOTE

Errata e3452: If a double-bit ECC error is encountered when reading the data flash memory, a functional reset will occur if the next data flash memory access is a program (irrespective of the length of time between the data read causing the ECC error and the program attempt). If another location in the data flash memory is read (which does not generate an ECC error) before attempting to program the data flash memory, the reset does not occur. This only impacts programming operations - an erase after a double-bit ECC error will not generate a reset.

NOTE

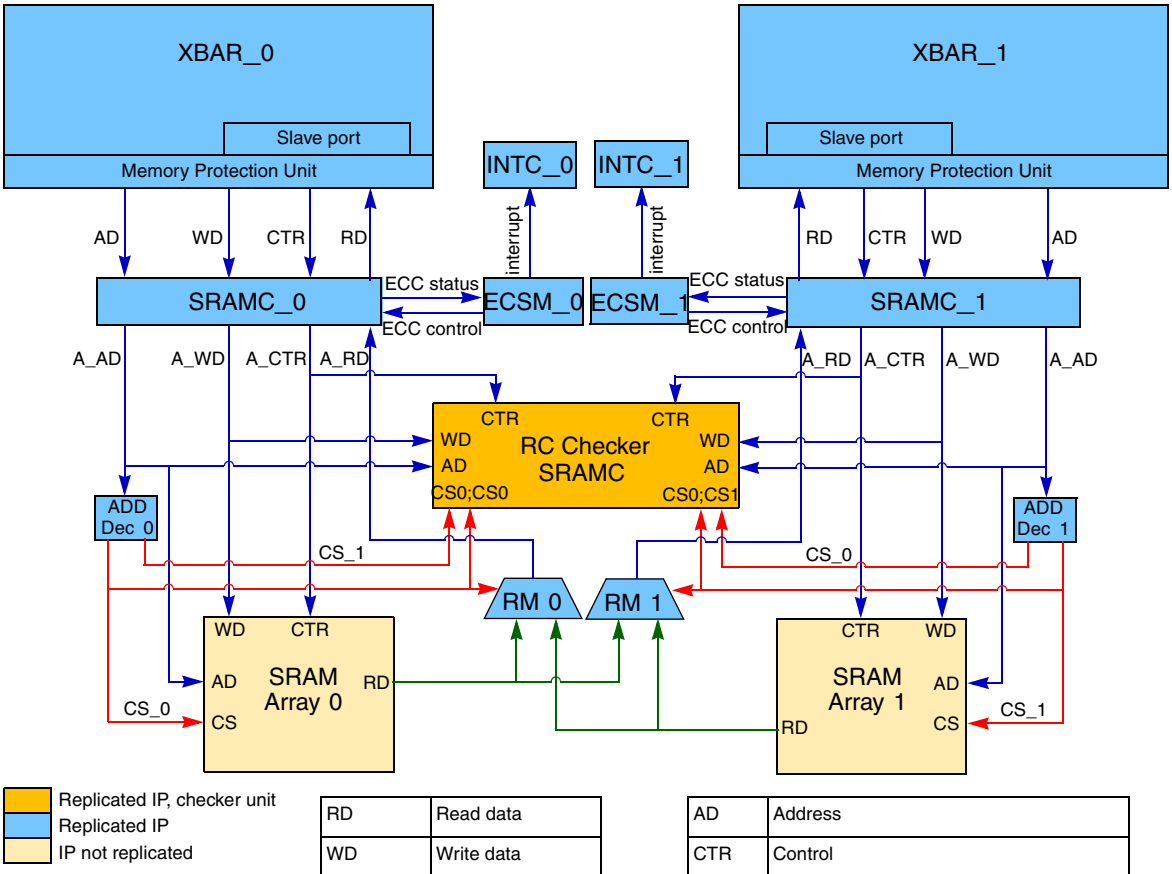
As the ECC module protecting the flash memory is not replicated. Respective functional safety mechanisms implemented in software are required to achieve an appropriate diagnostic coverage regarding single-point and latent faults.

NOTE

In LSM, SRAM accesses are received by both SRAM controllers simultaneously. Both SRAMC correct erroneous single bit and detect dual and multiple bit faults independently (dual channel).

NOTE

In DPM, in case core 0 only accesses SRAM_0 and core 1 accesses only SRAM 1, SRAM accesses of the two channels are received by different SRAM controllers independently. Both SRAMC correct erroneous single bit and detect dual and multiple bit faults independently (dual channel).



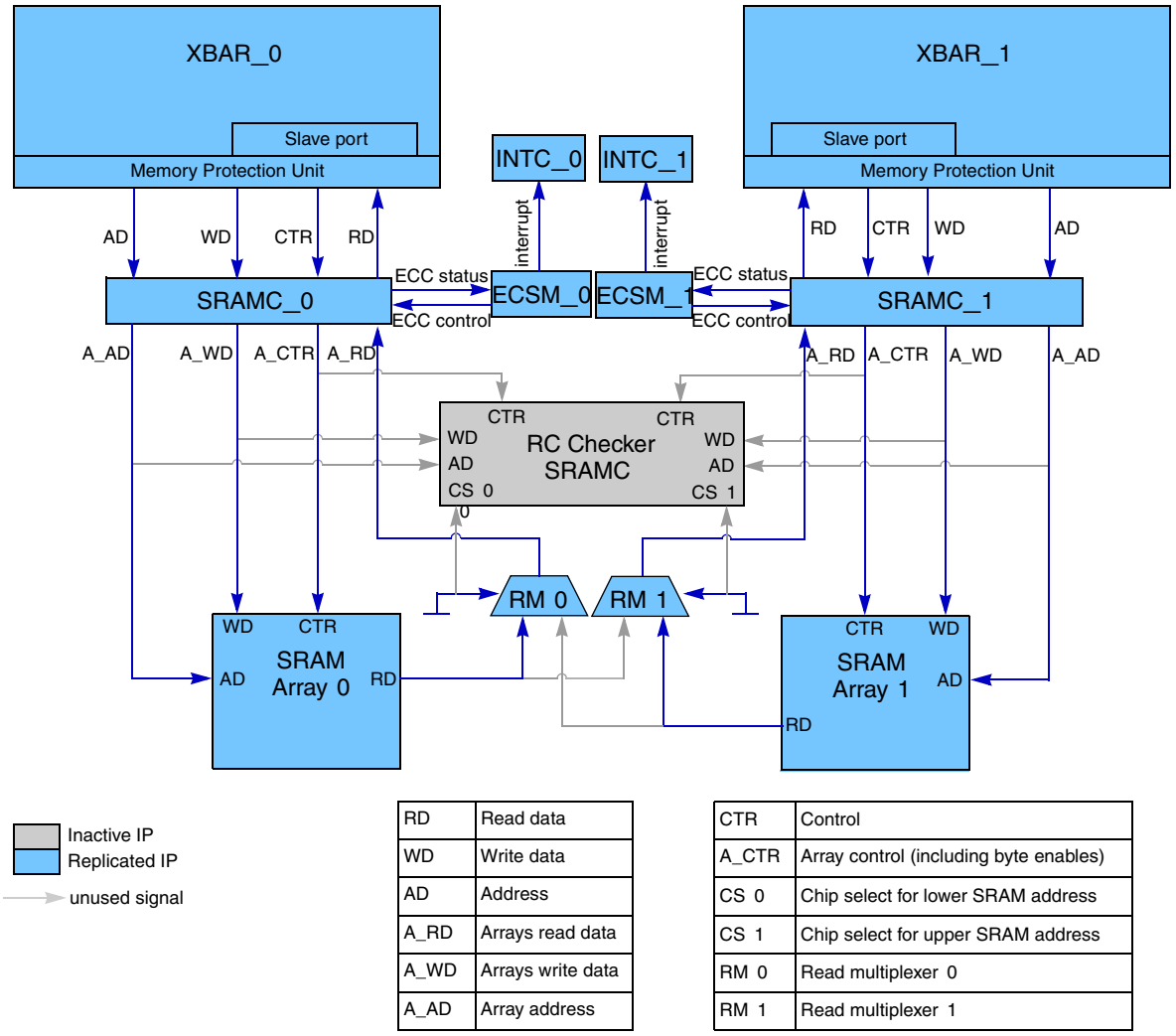
RD	Read data
WD	Write data
A_RD	Arrays read data
A_WD	Arrays write data
RM_0	Read multiplexer 0
RM_1	Read multiplexer 1

AD	Address
CTR	Control
A_CTR	Array control (including byte enables)
CS_0	Chip select for lower SRAM address range
CS_1	Chip select for upper SRAM address range
A_AD	Array address

Figure 12. SRAM integration in LSM

NOTE

In DPM, SRAM accesses are handled by completely replicated circuitries.



RD	Read data
WD	Write data
AD	Address
A_RD	Arrays read data
A_WD	Arrays write data
A_AD	Array address

CTR	Control
A_CTR	Array control (including byte enables)
CS 0	Chip select for lower SRAM address
CS 1	Chip select for upper SRAM address
RM 0	Read multiplexer 0
RM 1	Read multiplexer 1

Figure 13. SRAM integration in DPM

5.2.17.1 Runtime checks

5.2.18 Interrupt Controller (INTC)

As INTC is a replicated module, no software action is required to detect faults within this module.

No specific hardware protection is provided to reduce the likelihood of spurious or missing interrupt requests, which may be caused for example, by Electromagnetic Interface (EMI) on the interrupt lines, bit flips in the interrupt registers of the peripherals, or a fault in the peripherals.

5.2.18.1 Runtime checks

Safety requirement under certain preconditions: [SM_070] Applications that are not resilient against spurious or missing interrupt requests may need to include detection or protection measures on system level. [end]

Rationale: To manage spurious or missing interrupt requests.

Implementation hint: A possible way to detect spurious interrupts is to check corresponding interrupt status in the interrupt status register (polling) of the related peripheral before executing the Interrupt Service Routine (ISR) service code.

5.2.19 Semaphore Unit (SEMA4)

Semaphore modules are only used in DPM. Failures of the SEMA4 module may cause unwanted interrupts in LSM.

Each SEMA4 unit is connected to both replicated INTC modules. This means that even in LSM when SEMA4 units are not used, a corrupted SEMA4 could trigger continuous interrupts to both INTCs. To reduce the likelihood of this failure the INTC should have the SEMA4 interrupt masked (IRQ 247 (SEMA4_0) and 248 (SEMA4_1)).

5.2.19.1 Initial checks and configurations

Safety requirement under certain preconditions: [SM_048] To improve the strength of common mode faults, application software may mask the SEMA4 interrupts by programming the interrupt controller to reduce the likelihood of spurious interrupts if LSM is activated. [end]

5.2.20 Enhanced Direct Memory Access (eDMA)

As eDMA is a replicated module, no software action is needed to detect faults inside this module.

5.2.20.1 Runtime checks

Safety requirement under certain preconditions: [SM_049] Applications that are not resilient to spurious, or missing functional safety relevant, eDMA requests may include detection or protection measures on system level. [end]

Rationale: To manage spurious or missing eDMA transfer requests.

Implementation hint: The methodology to satisfy this requirement is application dependent. Two possible implementations which satisfy these requirements are:

- Counting the number of eDMA transfers triggered inside a control period and comparing this value with the expected one.
- If the eDMA is used to manage the analog acquisition with the Cross-Triggering Unit (CTU) and ADC, the number of the converted ADC channels is saved in the CTU FIFO together with the acquired value. The eDMA transfers this value from the CTU FIFO to a respective SRAM location. Spurious or missing transfer requests can be detected by comparing the converted channel with the expected one.

Safety requirement under certain preconditions: [SM_050] Applications that are not resilient to spurious, or missing functional safety relevant, eDMA requests can not use the PIT module to trigger functional safety-relevant eDMA transfer requests. [end]

Rationale: To reduce the likelihood of a faulty PIT (which is not redundant) from triggering an unexpected eDMA transfer.

5.2.21 Periodic Interrupt Timer (PIT)

5.2.21.1 Runtime checks

Recommendation under certain preconditions: When using PIT module, the PIT module should be used in such a way that a possible functional safety relevant failure is detected by the Software Watchdog Timer (SWT).

Rationale: To catch possible PIT failures.

Recommendation under certain preconditions: When the PIT is used in a safety function, a checksum of its configuration registers may be calculated and compared with the expected value to check that the PIT configuration is correct.

Rationale: To check that the PIT remains at its expected configuration.

5.2.22 System Status and Configuration Module (SSCM)

5.2.22.1 Initial checks and configurations

Recommendation: Since the software integrated in the BAM has not been developed in an ISO 26262 or IEC 61508 compliant development process, system level measure must be taken to ensure system integrity or disable use of the BAM.

Rationale: Since BAM code was neither developed nor qualified according to the IEC 61508-4 or ISO 26262-6, any execution of the BAM, or part of it, needs to be inhibited or validated by appropriate measures.

Implementation hint: Execution of BAM code may be supervised (inhibited) by writing SSCM_ERROR[PAE] = 1. Each access to the BAM memory area then produces an exception.

Safety requirement: [SM_022] During and after initialization but before executing any safety function, the application software needs to read SSCM_STATUS[LSM] and to confirm that the device runs in respective Mode (LSM: SSCM_STATUS[LSM] = 1, decoupled parallel Mode: SSCM_STATUS[LSM] = 0). [end]

5.2.23 Flash memory

To support the detection of latent faults in the flash memory array and addressing logic, the integrity of the logic used for flash memory programming requires integrity validation. So, the ECC logic and the array integrity self check need to be enabled by software.

This array integrity self check and the ECC logic test is based on hardware circuits integrated in the flash memory control logic. An array integrity self check calculates a MISR signature over the array content and

thus validates the content of the array as well as the decoder logic. The calculated MISR value depends on the array content and must be validated by application software.

The array integrity self check and the ECC logic check must be executed on each program flash memory block used. Additionally, the ECC logic check must be executed for the data flash memory.

The ECC logic test calculates a MISR signature over test vectors and, thus, validates the correct function of the ECC logic. The calculated MISR value must be validated by application software.

5.2.23.1 Initial checks and configurations

Safety requirement: [SM_023] Before executing any safety function, a flash memory array integrity self check should be executed. The calculated MISR value is dependent on the array content and, therefore, has to be validated by system level application software. [end]

Rationale: To check the integrity of the flash memory array content.

Implementation hint: This test may be started by application software: its result may be validated by reading the corresponding registers in the flash memory controller after it has been finished (see “Array integrity self check” section in the “Flash memory” chapter of the *Qorivva MPC5675K Microcontroller Reference Manual*).

5.2.23.2 Runtime checks

Safety requirement: [SM_024] When programming the flash memory, the corresponding software driver must validate that the flash memory was programmed correctly. [end]

Rationale: To check that the written data is coherent with the expected data.

Implementation hint: The programming of flash memory may be validated by checking the value of C90FL_MCR[PEG] (FLASH_SW_READBACK). Furthermore, the data written may be read back, then checked by software if identical to the programmed data. The data the read back may be executed in Margin Read Enable mode (C90FL_UT0[MRE] = ‘1’). This enables to validate the programmed data using read margins more sensitive to weak programme or erase status. This requires two separate checks, one with read margin sensitive to weak programming (C90FL_UT0[MRE] = ‘1’ and C90FL_UT0[MRE] = ‘0’) and another with read margin sensitive to weak erasing (C90FL_UT0[MRE] = ‘1’ and C90FL_UT0[MRE] = ‘1’).

The flash memory SEC/DED only contains data but no addresses (please refer to Chapter 5.2.17, “Error correction (ECC, ECSM)”). Therefore, a flash memory ECC logic test must be implemented by application software.

Safety requirement: [SM_025] A flash memory ECC logic test (FLASH_SW_ECCTEST) must be executed within the FTTI. This validates the (digital) logic within the flash memory that is responsible for detecting and correcting faults (ECC logic) in the data that is read. [end]

Rationale: The intention of this test is to assure that correct data is not accidently modified, and single bit errors are correctly updated.

Implementation hint: Software can check ECC logic by providing appropriate test patterns to the input of ECC logic, 32 bits of data and 7 bits of ECC parity bits. Software may validate that the ECC provides

the correct action (correction, detection). A test interface enables software to force the output of the ECC (see the “ECC logic check” chapter in the *Qorivva MPC5675K Microcontroller Reference Manual* and Section 5.2.23.3.2, FLASH_SW_ECCTEST).

NOTE

ECC logic check does not transmit any detected fault information to the RCCU, and so on.

Safety requirement: [SM_026] Flash memory ECC failure reporting has to be executed within the FTTI to validate if detected ECC faults are communicated properly to the FCCU and other necessary modules. [end]

Rationale: The intention of this test is to assure that failure detection is correctly reported.

Implementation hint: It consists of reading a set of data words from flash memory having erroneous ECC bits programmed. Respective ECSM register content may be validated by software (see Section 5.2.23.3.1, FLASH_SW_ECCREPORT).

5.2.23.3 Implementation details

5.2.23.3.1 FLASH_SW_ECCREPORT

The goal of the FLASH_SW_ECCREPORT is to ensure high coverage of the faults in ECC logic fault signalling and communication with minimum performance penalty to application software.

The MPC567xK flash memory has no hardware support to inject ECC faults (fake faults) which are reported to the FCCU. Therefore data with ECC pattern not matching to the data has to be programmed into flash memory. A suitable program flash memory location separated from any used data has to be selected, to avoid non intended reads from this ECC test location. Preferable a location within the shadow space of the flash memory is used to avoid erasing of the test pattern during update of program flash memory.

The procedure to program and test a pattern with a single-bit injected fault, then be corrected may be:

1. Write 0000_0000_0000_0001h to the program flash memory test location.
2. Write 0000_0000_0000_0000h to the same program flash memory test location without going through an erase operation. Ignore the programming error message.
3. Subsequent reads from this program flash memory test location will report single-bit error correction if appropriately enabled by C90FL_UT0[SBCE] = 1.

In order to avoid unintended reads from this ECC test location a suitable second program flash memory location, separated from any used data (including the previous single-bit test location), is to be selected. Preferable a location within the shadow space of the flash memory is used to avoid erasing of the test pattern during update of program flash memory. The procedure to program and test a pattern with a double-bit injected fault to be detected is:

1. Write 0x0000_0000_0000_0003 to the program flash memory test location.
2. Write 0000_0000_0000_0000h to the same program flash memory test location without going through an erase operation. Ignore the programming error message.

3. Subsequent reads from this program flash memory test location will report a double-bit error detection by generating a machine check exception.

5.2.23.3.2 FLASH_SW_ECCTEST

The goal of the FLASH_SW_ECCTEST is to ensure high coverage of the faults in ECC logic with minimum performance penalty to application software.

To give an estimation, the performance penalty of an implementation of this test is determined to be 176 µs, which is less than 2 % of the available processing time considering a FTTI of 10 ms.

The MPC567xK flash memory has a UTEST (user-test) mode ECC logic check feature which can be utilized for this ECC logic test. A data pattern with walking ‘0’ through data and ECC parity bits can be applied during the ECC logic check procedure to achieve high fault coverage of the ECC logic and fast execution.

5.2.23.3.2.1 Data pattern - walking ‘0’

To achieve the required diagnostic coverage data pattern with walking ‘0’ through data and ECC parity bits may be used. Table 4 shows the data vectors.

Table 4. Data pattern used by the ECC logic test[1]

Data vector number	8-bit ECC parity bits	64-bit data bits
0	0xFF	0xFFFF_FFFF_FFFF_FFFE
1	0xFF	0xFFFF_FFFF_FFFF_FFFD
2	0xFF	0xFFFF_FFFF_FFFF_FFFB
3	0xFF	0xFFFF_FFFF_FFFF_FFF7
4	0xFF	0xFFFF_FFFF_FFFF_FFEF
5	0xFF	0xFFFF_FFFF_FFFF_FFDF
6	0xFF	0xFFFF_FFFF_FFFF_FFBF
7	0xFF	0xFFFF_FFFF_FFFF_FF7F
…	…	…
62	0xFF	0xBFFF_FFFF_FFFF_FFFF
63	0xFF	0x7FFF_FFFF_FFFF_FFFF
64	0xFE	0xFFFF_FFFF_FFFF_FFFF
65	0xFD	0xFFFF_FFFF_FFFF_FFFF
…	…	…
71	0x7F	0xFFFF_FFFF_FFFF_FFFF
72	0xFF	0xFFFF_FFFF_FFFF_FFFF

NOTES:
1 Each vector is a 72-bit ECC code-word.

It is important to note that for double-word data = FFFF_FFFF_FFFF_FFFFh, the correct ECC check bits should be 0xFF. Therefore, every data vector in above data pattern, except the last one, contains a single-bit

ECC error and will result in a single-bit correction. These erroneous patterns have to be programmed into flash memory and periodically read to identify correct function of the ECC logic.

5.2.23.3.2.2 UTEST mode ECC logic check

The procedure to use the UTEST mode ECC logic check is:

1. Write F9F9_9999h to UT0 to enable UTEST mode (UT0[UTE] will be set).
2. Write UT0[SBCE] to '1' to enable single-bit error correction visibility.
3. Write UT0[EIE] to '1'.
4. Write UT0[DSI], UT1[DAI] and/or UT2[DAI] bits to provide the current data vector including the double-word data and check bit values to be read. The data and check bit values are from the chosen ECC test data pattern, i.e., walking '0' pattern shown above.
5. Write double-word address to receive the data input in step 4 into the ADR register.
6. Reads the address stored in ADR register via BIU using a CPU instruction. The expected data, and corrections or detections should be observed based on data written into the UT0[DSI], UT1[DAI] and/or UT2[DAI] registers. MCR[EER] and MCR[SBC] will be checked to evaluate the status of reads done.
7. Repeat steps 4 to 6 for all the data vectors in the proposed test data pattern.
8. Once completed, clear the UT0[EIE] bit to 0.

5.2.24 Cross Triggering Unit (CTU)

The CTU generates triggers based on input events (FlexPWMs, eTimers, and/or external GPIO).

The trigger can be caused by:

- A pulse
- An interrupt
- An ADC command (or a stream of consecutive commands)
- All of these

5.2.24.1 Runtime checks

Safety requirement: [SM_027] The CTU must be properly configured so output triggers are generated within the desired time schedule with respect to the input event(s). [end]

Rationale: To reduce the likelihood of erratic output trigger generation.

For each trigger, a set of ADC commands and pulses to be generated can be defined.

If the application safety function includes the read of inputs synchronized with events (FlexPWMs, eTimers, and external signals, or any combination), the system integrator can use the CTU module for this purpose. The required software needed is listed in Section 5.2.24.2, Synchronize sequential read input.

For a detailed description on how the CTU works (triggered and sequential mode), its configuration and use, refer to the *Qorivva MPC5675K Microcontroller Reference Manual*.

5.2.24.2 Synchronize sequential read input

5.2.24.2.1 CTU_HWSWTEST_TRIGGERNUM

If the reload signal occurs before all the triggers are generated, an overrun indication is flagged and the application software may have to handle the error indication.

Rationale: Tests if all the triggers configured within a control period have been generated and serviced.

Implementation hint: The Cross Triggering Unit Error Flag register (CTU_CTUEFR) shows information about the overrun status.

When the CTU detects an error, an interrupt is generated. In the interrupt service routine, the value of the Error Flag Register (CTUEFR) is tested for error condition. If any of the tested bits are valid (= 1, thus an error occurred), appropriate actions may be required.

5.2.24.2.2 CTU_SWTEST_TRIGGERTIME

Application software configures one eTimer channel to capture the time at which each trigger event occurs.

In triggered mode, the time instant of each trigger within one control period is captured and stored in a FIFO. Application software has to check the FIFO values against the expected ones according to CTU configuration.

In sequential mode, an eTimer channel is used to check the correct time of a single trigger with respect to the corresponding event.

Rationale: To check if triggers are generated at the correct time.

Implementation hint: Some eTimer inputs are internally connected to the CTU output. See “Enhanced Motor Control Timer (eTimer)” in the *Qorivva MPC5675K Microcontroller Reference Manual* for details.

Implementation hint: eTimer capture register implements a two entry FIFO, but in CTU triggered mode up to 8 time values need to be stored. To reduce the likelihood of FIFO overflow condition, eTimer can be configured to trigger a eDMA transfer to move the captured value to specific RAM location.

In sequential mode, an eTimer channel may be needed to check the correct time of a single trigger with respect to the corresponding event.

5.2.24.2.3 CTU_HWSWTEST_TRIGGEROVERRUN

This hardware mechanism checks if a new trigger occurs that requires an action by a subunit that is currently busy. In this case, an overrun interrupt is generated and the application software handles the error condition.

Over-run detection mechanism must be enabled by software during configuration of the CTU.

Rationale: Checks if a new trigger occurs that requires an action by a subunit (for example, ADC command generator) that is currently busy.

Implementation hint: To enable the over-run detection the CTU_CTUIR[IEE] is written with a 1. This interrupt is shared between several sources of error. The application software can determine which particular interrupt is represented by reading the CTU_CTUEFR.

5.2.24.2.4 CTU_HWSWTEST_ADCCOMMAND

The CTU stores in its internal FIFOs both the value provided by each ADC conversion and the channel number. Application software checks the ADC channel number sequence against what is expected for each FIFO. Moreover, invalid commands issued by the CTU are flagged and the corresponding error is handled by the application software (not included in example code).

Rationale: To detect if the incorrect channel has been acquired, or if the incorrect ADC result FIFO is selected.

Implementation hint: To enable detection of invalid commands, the CTU_CTUIR[IEE] flag needs to be asserted. This interrupt is shared between several sources of error. They can be discriminated by reading the CTUEFR register.

This safety integrity function is required only when reading analog signals.

5.2.24.2.5 CTU_SWTEST_ETIMERCOMMAND

Application software configures one channel of eTimer_0, eTimer_1 or eTimer_2 to count the number of eTimer commands generated within a CTU control period and checks the number against the expected one.

Rationale: To check the correctness of the number of generated commands.

Implementation hint: Some eTimer inputs are internally connected to the CTU output. (see the *Qorivva MPC5675K Microcontroller Reference Manual* for details).

5.2.24.2.6 CTU_HW_CFGINTEGRITY

This hardware mechanism ensures the consistency of the CTU configuration at the beginning of each CTU control period.

The configuration registers are all double-buffered. If the configuration is only partial when the control period starts, the previous configuration is used and an error condition is flagged, which is handled by the application software.

Rationale: Ensures the consistency of the CTU configuration.

Implementation hint: The CTU uses a safe reload mechanism. The General Reload Enable (GRE) bit in the Cross Triggering Unit Control Register (CTUCR) has to be used to detect partial or incomplete CTU update. To enable the interrupt in case of error during reload, CTU_CTUIR[IEE] = 1. This interrupt is shared between several sources of error. They can be discriminated by reading the CTUEFR register. Alternatively, repetitive reading of MRS_RE is also possible.

5.2.24.2.7 Other requirements for CTU module usage

Safety requirement: [SM_028] If the CTU is used to read an analog signal through the ADC, the software must check the Invalid Command Error flag (CTU_CTUEFR[ICR]) after programming the ADC command lists. [end]

Rationale: To check the presence of invalid commands.

5.2.25 Fault injection tests

It is possible to use fault injection (fake faults) to check the correct implementation of functional safety mechanisms. Fault injection is provided primarily for software development and validation purposes.

Fault injection is mainly implemented in the FCCU. Each possible critical or non-critical fault input of the FCCU (please refer to Table 1 and Table 2) can be triggered by software.

Additionally errors can be injected into Flexray SRAM and System SRAM to generate injected ECC errors.

FLASH_SW_ECCREPORT (see Section 5.2.23.3.1, FLASH_SW_ECCREPORT) may be used to inject ECC errors in program flash memory, and FLASH_SW_ECCTEST (see Section 5.2.23.3.2, FLASH_SW_ECCTEST) may be used to inject faults in the ECC. However, FLASH_SW_ECCTEST does not allow to forward injected faults to the FCCU.

5.2.26 SRAM

A multiple cell failure caused for example, by a neutron or alpha particle or a short circuit between cells may cause three or more bits to be corrupted in an ECC-protected word. As result, either the availability may be reduced or the ECC logic may perform an additional data corruption labeled as single bit correction. This is prevented within the design of MPC567xK by the use of bit scrambling (column multiplexing) which effects, that physically neighboring columns of the RAM array do not contain bits of the same logical word but the same bit of neighboring logical words. Thus the information is logically spread over several words causing only single bit faults in each word which can be correctly corrected by the ECC. MPC567xK has a multiplexor factor of eight for its system RAM multiplexing adjacent analog bit lines to an analog sense amplifier. It is always enabled and needs no configuration.

Safety requirement: [SM_069] It is required read at least four different addresses per RAM block within the FTTI will occur. [end]

Rationale: To provide sufficient diagnostic coverage for column repair with ECC.

5.2.27 Glitch filter

An analog glitch filter is implemented on the reset signal of the MPC567xK. A selectable (WKPU_NCR[NFE0]) analog glitch filter is implemented on the NMI-input. External interrupt sources can be configured to be used with any chip GPIO. Interrupt sources (1 to 32) can be configured to have a digital filter to reject short glitches on the inputs. These filters are used to reduce noise and transient spikes in order to reduce the likelihood of unintended activation of the reset or the interrupt inputs.

5.2.28 Register Protection module (REG_PROT)

The PowerPC architecture supports two levels of privilege for program execution: user mode and supervisor mode. Only the supervisor mode allows the access to the entire CPU register set, and the execution of a subset of instructions is limited to supervisor mode only. In user-mode, access to most registers including system control registers is denied. It is intended that most parts of the software be executed in user-mode so that the MPC567xK is protected from errant register changes made by other user-mode routines.

In addition, all peripherals, processing modules and other configurable IP is protected by a REG_PROT module, which offers a mechanism to protect address locations in a module under protection from being written (for example, to handle the concurrent operation of software tasks with different or lower functional safety integrity level). It includes the following levels of access restriction:

- A register cannot be written once Soft Lock Protection is set. The lock can be cleared by software or by a system reset.
- A register cannot be written once Hard Lock Protection is set. The lock can only be cleared by a system reset.
- If neither Soft Lock nor Hard Lock is set, the Register Protection module may restrict write accesses for a module under protection to supervisor mode only.

5.2.28.1 Runtime checks

Recommendation: All configuration registers, and registers that are not modified during application execution, are to be protected with a Hard Lock.

Rationale: Hard Lock is the last access protection against unwanted writes to some predefined memory mapped address locations.

Implementation hint: Most of the off-platform peripherals have their own Register Protection module. Register Protection address space is inside the memory space reserved for the peripherals (please, refer to the “MPC567xK registers under protection” section of the *Qorivva MPC5675K Microcontroller Reference Manual*). Each peripheral register that can be protected through the Register Protection module has a Set Soft Lock bit reserved in the Register Protection address space. This bit is asserted to enable the protection of the related peripheral registers. Moreover, the Hard Lock Bit (REG_PROT_GCR[HLB] = 1) should be set for best write protection.

Recommendation: It is recommended that only hardware related software (OS, drivers) run in supervisor mode.

5.2.29 External Bus Interface (EBI)

As parts of the external bus interface (EBI) of the MPC567xK do not provide the functional safety integrity (external bus interface is not a replicated module) IEC 61508 series and ISO 26262 requires for high functional safety integrity targets. Therefore system level measures are required.

5.2.29.1 Runtime checks

Safety requirement under certain preconditions: [SM_054] When a functional safety functions relies on the operation of the External Bus Interface, functional safety integrity measures must be implemented on system level to achieve the required functional safety integrity. These measures shall provide functional safety integrity regarding data domain (reducing the likelihood of data corruption) and control domain (reducing the likelihood of address corruption and control faults). [end]

Rationale: To achieve the integrity of data written and read through the External Bus Interface.

Implementation hint: The eDMA and CRC modules may be used periodically, within the FTTI, to calculate the CRC signature of non-volatile data stored in memory devices addressed by the EBI and compare this with a expected value. Alternatively, data shall be stored redundantly, using different coding schemes (for example, inverted bits). On every read operation redundant data are compared by the system level application software.

5.2.30 Multi-port DDR DRAM controller (MDDRC)

As parts of the Multi-port DDR DRAM Controller (mDDR) of the MPC567xK do not provide the functional safety integrity (external bus interface is not a replicated module) IEC 61508 series and ISO 26262 requires for high functional safety integrity targets. Therefore system level measures are required.

5.2.30.1 Runtime checks

Safety requirement under certain preconditions: [SM_055] When a functional safety functions relies on the operation of the MCCRC, functional safety integrity measures shall be implemented on system level to achieve the required functional safety integrity. These measures shall provide functional safety integrity regarding data domain (reducing the likelihood of data corruption) and control domain (reducing the likelihood of address corruption and control faults). [end]

Rationale: To achieve the integrity of data written and read through the MDDRC.

Implementation hint: The eDMA and CRC modules may be used to periodically within the FTTI calculate the CRC signature of non volatile data stored in memory devices addresses by the MDDRC and compare this with a expected value. Alternatively, data is stored redundantly, preferably using different coding schemes (for example, inverted bits). On every read operation the redundant data are compared.

5.2.31 Wake-Up Unit (WKPU) / External NMI

Safety requirement under certain preconditions: [SM_068] If external NMI and Wake-up are used as a safety mechanism, it is required to implement respective system level measures to detect latent faults within WKPU. [end]

Rationale: To test the analog filter of the WKPU for external NMIs and wakeup events.

Implementation hint: To test the analog filter of the WKPU for external NMIs, application software may configure the NMI during startup to cause only a critical interrupt, then trigger the external NMI and check that the critical interrupt occurred.

5.2.32 Crossbar Switch 2 (XBAR2)

As parts of the crossbar switch 2 (XBAR2) of the MPC567xK do not provide the functional safety integrity (external bus interface is not a replicated module) IEC 61508 series and ISO 26262 requires for high functional safety integrity targets. Therefore system level measures are required.

5.2.33 Analog to Digital Converter (ADC)

Parts of the Analog-to-Digital Converter (ADC) of the MPC567xK do not provide the functional safety integrity IEC 61508 series and ISO 26262 requires for high functional safety integrity targets. Therefore system level measures are required.

5.2.33.1 Initial checks and configurations

Safety requirement under certain preconditions: [SM_045] When Analog-to-Digital Converter (ADC) of the MPC567xK are used in a safety function, suitable system level functional safety integrity measures must be implemented after reset (external reset or destructive reset) before starting the respective safety function to ensure ADC integrity. [end]

Recommendation: After reset (external reset or destructive reset), but before executing any safety function, the following hardware BISTs of one or both ADC modules may be executed by the application software to detect latent faults:

- RESISTIVE-CAPACITIVE SELF-TEST
- SUPPLY SELF-TEST
- CAPACITIVE SELF-TEST

Rationale: To check the integrity of the ADC modules

These tests can be executed in either of the following modes:

- CPU mode
- CTU mode

In CPU mode, the application software takes care of the hardware self-test activation and checks the test flow and the timing.

In CTU mode, the CTU module takes care of the hardware self-test activation, flow monitoring, and timing. It is important to note that in this operating mode, the CPU does not take part in running the hardware self-test.

Hardware self-tests use analog watchdogs to check the outcome of self-test conversions. The reference thresholds of these watchdogs are saved in the flash memory test sector.

Safety requirement under certain preconditions: [SM_046] Before running the ADC hardware self-test, the system integrator must copy the reference thresholds from test flash memory into the watchdog registers (STAW*n*R). [end]

Rationale: To set the correct threshold for the self-tests.

Implementation hint: Table 5 shows mapping of the values stored in test flash memory to be copied into the watchdog registers. Depending on the reference voltage used for the ADCs, ADC*n*_CAL W4 or ADC*n*_CAL W5 is to be used for ADC_*n*_STAW0R.

Please refer to the “self-test analog watchdog” section of the “ADC” chapter and the “Test sector” section of the “Flash Memory” chapter in the *Qorivva MPC5675K Microcontroller Reference Manual* for details.

Table 5. Mapping of test flash memory values to STAWxR

Watchdog register	Name in test flash memory	Address in test flash memory	Remark
STAW3RH	ADC*n*_CAL W1	0x0010 / 0x0034	—
STAW3RL		0x0012 / 0x0036	—
STAW4RH	ADC*n*_CAL W2	0x0014 / 0x0038	—
STAW4RL		0x0016 / 0x003A	—
STAW5RH	ADC*n*_CAL W3	0x0018 / 0x003C	—
STAW5RL		0x000A / 0x003E	—
STAW0RH	ADC*n*_CAL W4	0x000C / 0x0040	to be used if $V_{DD\_HV\_ADRn}$ = 3.3 V
STAW0RL		0x000E / 0x0042	
STAW0RH	ADC*n*_CAL W5	0x0020 / 0x0044	to be used if $V_{DD\_HV\_ADRn}$ = 5 V
STAW0RL		0x0022 / 0x0046	
STAW1ARH	ADC*n*_CAL W6	0x0024 / 0x0048	—
STAW1ARL		0x0026 / 0x004A	—
STAW1BRH	ADC*n*_CAL W7	0x0028 / 0x004C	—
STAW1BRL		0x002A / 0x004E	—
STAW2R	ADC*n*_CAL W8	0x002E / 0x0050	—

Implementation hint: Since test flash memory cannot be read directly, the Test Flash Enable feature of the SSCM may be exploited. This action is performed through the following steps:

1. If code is executed out of flash memory, CPU branches into RAM and executes code out of SRAM memory.
2. Write SSCM_STCR[TFE] = 1.
3. Test sector is readable at the offset 0x0 of the flash memory address space (See “System Status and Configuration Module (SSCM)” of the *Qorivva MPC5675K Microcontroller Reference Manual*).
4. Thresholds are copied from the Test sector to the respective register.
5. Write SSCM_STCR[TFE] = 0.
6. Code can continue execution out of flash memory.

NOTE

As the BAM is not developed following an ISO compliant software process, system integrators are asked to avoid reading the test sector through the BAM access method. Please refer to Chapter 5.2.22, “System Status and Configuration Module (SSCM)” for details.

Safety requirement under certain preconditions: [SM_047] When using integrated self-test as the functional safety integrity measure, the analog watchdog timer for CPU mode and CTU mode must be enabled for the self-test. The programmable watchdog timeout is smaller than the FTTI. [end]

Rationale: To check the correct completion of the ADC self-test algorithms.

Implementation hint: Every hardware BIST is activated via a dedicated command sent to the ADC (see "self-testing" section in the "ADC" chapter of the *Qorivva MPC5675K Microcontroller Reference Manual* for details on implementing these tests).

The SUPPLY SELF-TEST is executed without interleaved conversion.

Due to its analog parts, the ADCs require additional tests implemented in software. Please refer to Section 5.3.3, Analog inputs.

5.3 I/O functions

The integrity of functional safety relevant periphery is mainly ensured by application level measures (for example, connecting one sensor to different I/O modules, sensor validation by sensor fusion).

Functional safety relevant peripherals are assumed to be used redundantly in some way. Different approaches can be used, for example, by implementing replicated input (for example, connect one sensor to two DSPIs or even connect two sensors measuring the same quantity to two ADCs) or by crosschecking some I/O operations with different operations (for example, using sensor values of different quantities to check for validity). Preferably, the replicated modules generate or receive the replicated data using different coding styles (for example, inverted in the voltage domain or using voltage and time domain coding for redundant channels). System integrators may choose the approach that best fits their needs.

Safety requirement under certain preconditions: [SM_056] No specific hardware measures have been implemented to specifically reduce common mode failure(s) regarding replicated I/O peripherals. In case system level requires specific robustness regarding common mode faults within the I/O peripheral system, respective measures are required on system level. [end]

Rationale: To improve the common mode fault robustness of the I/O.

Implementation hint: Possible measures could use different coding schemes within each redundant I/O channel (for example, inverted signals, different time periods).

Implementation hint: Possible measures could be using different replicated peripherals (for example, eTimer_0, eTimer_1, or FlexPWM) to implement multiple independent and different channels.

Safety requirement under certain preconditions: [SM_057] Peripherals (for example, SIUL, FlexPWM or the eTimer) used for the functional safety function must be configured properly before their usage. Any misconfiguration prevents them from delivering the expected functionality. [end]

Rationale: To configure peripherals used by the safety functions and to reduce the likelihood of CMFs caused by improper configuration of the peripherals.

Safety requirement under certain preconditions: [SM_058] When safety functions use digital GPIO, the pads need to be configured by writing the appropriate values to the GPIOs corresponding SIUL_PCR*n*

(see the “System Integration Unit Lite (SIUL)” chapter in the *Qorivva MPC5675K Microcontroller Reference Manual*). [end]

Rationale: To configure GPIO used by the safety functions and to reduce the likelihood of CMF caused by improper configuration of the GPIO.

All accesses of Core_0 and DMA_0 use PBRIDGE_0 for every peripheral access. All access of Core_1 and DMA_1 use PBRIDGE_1. To reduce the possibility of CMFs, redundant channels must be read by different spheres of replication (for example, by Core_0 and Core_1 or DMA_0 and DMA_1).

This is also required regarding configuration, replicated I/O functions have to be configured by replicated masters (Core, DMA).

A single peripheral bus installed may be a source causing cascading faults in both processing channels.

5.3.1 Digital inputs

Safety requirement under certain preconditions: [SM_059] When safety functions use digital input, system level functional safety mechanisms have to be implemented to achieve required functional safety integrity. [end]

5.3.1.1 Hardware

Implementation hint: Functional safety digital inputs may to be acquired redundantly. To reduce the risk of common mode failures, the redundant channels may not use GPIO adjacent to each other (refer to Section 7.1, Causes of dependent failures).

- Double read operation of a digital input is implemented by two general purpose inputs (GPI) of the SIUL unit. A comparison (by software) between the double reads detects an error (please refer to Figure 14).
- A double read PWM input is implemented by using two modules as two channels. The functional safety integrity is achieved by double reads and a software comparison. One channel is provided by eTimer_0 and the other by eTimer_1 or eTimer_2. The usage of eTimer_1 and eTimer_2 is also possible. Read PWM input means any input read related to signal transitions (rise or fall). This may also include the time that the signal was high, low or both (please refer to Figure 14).
- A double read eTimer input is implemented by using two modules as two channels. The functional safety integrity is achieved by double reads and a software comparison. One channel is provided by eTimer_0 and the other by eTimer_1 or eTimer_2. The usage of eTimer_1 and eTimer_2 is also possible. Read Encoder Input means any input read elated to signal transitions (rise or fall). This may also include signals coming from an encoder (please refer to Figure 15).

For double read eTimer input, each signal, the SIUL can provide additional channels to support interrupt-based reading for each signal.

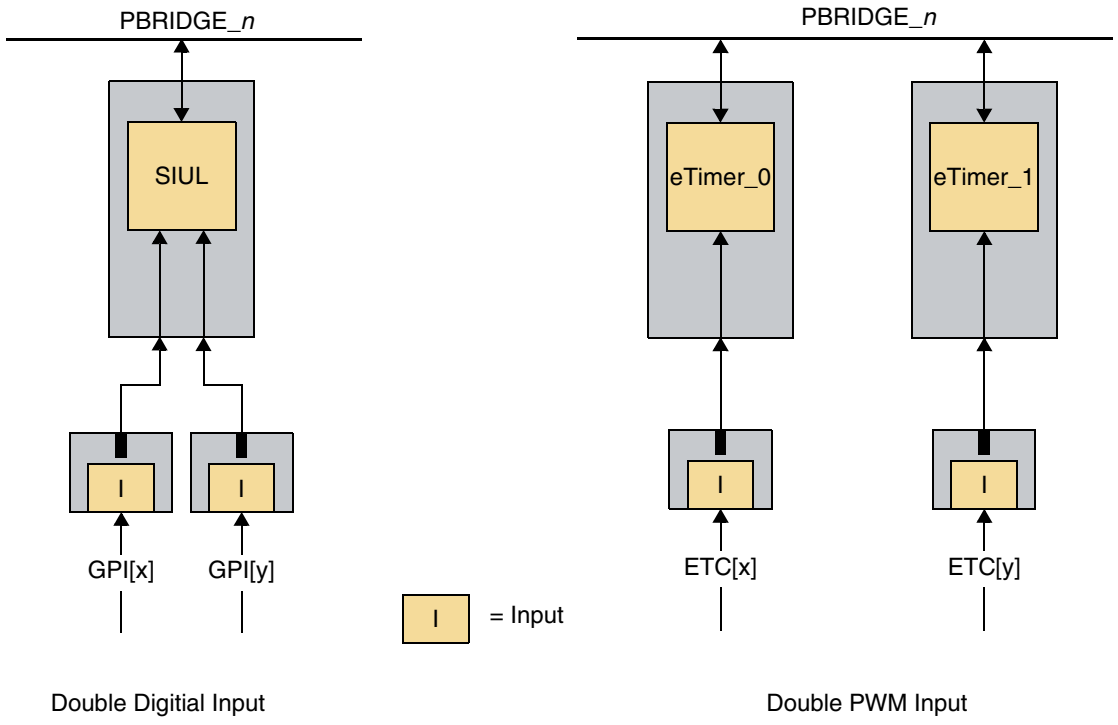


Figure 14. Double Digital Input and Double PWM input

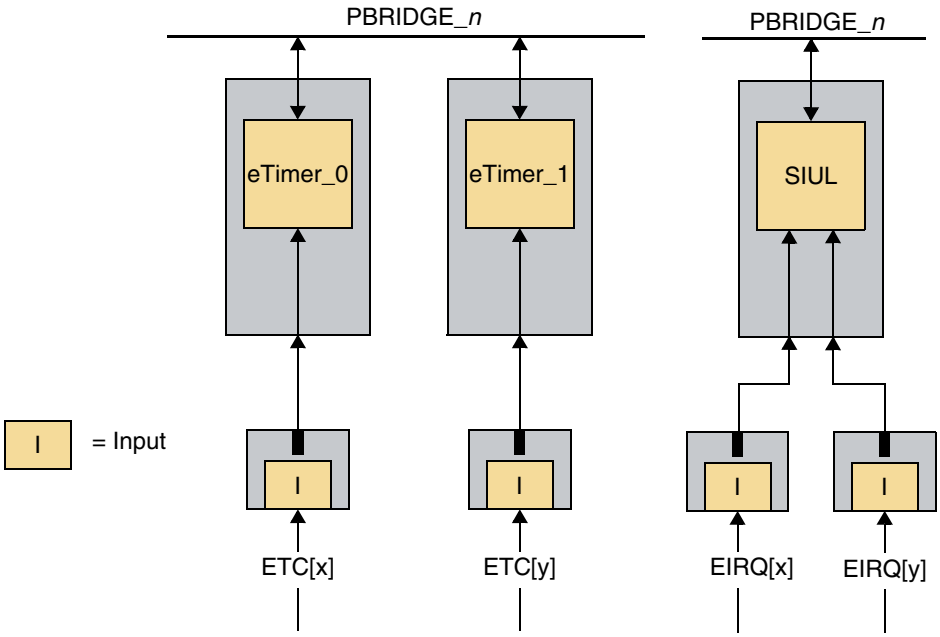


Figure 15. Double Read Encoder Input

Implementation hint: If sufficient diagnostic coverage can be obtained by a plausibility check on a single acquisition for a specific application, a plausibility check can replace a redundant acquisition.

5.3.1.2 Software

Digital inputs used for functional safety purposes are assumed to be input redundantly as described in this section. Table 6 lists three element safety functions for input, the corresponding safety integrity functions and their execution frequency. Alternative solutions with sufficient diagnostic coverage are possible.

Table 6. Digital inputs software tests

Function	Test	Frequency
Double Read Digital Inputs	SIUL_SWTEST_REGCRC	Once after programming
	GPI_SWTEST_CMP	Once for every acquisition
Double Read PWM Inputs	ETIMER0_SWTEST_REGCRC	Once after programming
	ETIMER1_SWTEST_REGCRC	Once after programming
	SIUL_SWTEST_REGCRC	Once after programming
	ETIMERI_SWTEST_CMP	Once for every acquisition
Double Read Encoder Inputs	ETIMER0_SWTEST_REGCRC	Once after programming
	ETIMER1_SWTEST_REGCRC	Once after programming
	SIUL_SWTEST_REGCRC	Once after programming
	ENCI_SWTEST_CMP	Once for every acquisition

5.3.1.2.1 Double Read Digital Inputs

Rationale: To check that the configuration of the two I/Os used correspond with the expected configuration, to reduce the likelihood of CMF caused by incorrectly configured I/Os, and to check that the two input values read are similar.

Implementation hint: Functional safety integrity is achieved by replicated reading and software comparison by the processing function. The application can implement the tests SIUL_SWTEST_REGCRC and GPI_SWTEST_CMP.

5.3.1.2.1.1 Implementation details

The only hardware element that can be used for the safety function is the general purpose I/O (GPIO).

Implementation hint: Every I/O that is not dedicated to a single function can be configured as GPIO. I/Os that are dedicated to ADC are an exception to this rule, as they can only be configured as inputs.

NOTE

Caution: Redundant GPIO should be selected in a way that their signals are not adjacent, which helps minimize the likelihood of CMFs.

5.3.1.2.1.2 SIUL_SWTEST_REGCRC

See Section 5.2.7, Cyclic Redundancy Checker Unit (CRC) for <*module*>_SWTEST_REGCRC implementation details.

5.3.1.2.1.3 GPI_SWTEST_CMP

This software test is used to execute the comparison between the double reads performed by the independent channels, and reads the outputs sequentially. This allows any GPIO to be used, but could result in a wrong result if the state of the input changes between reading the first and second inputs.

An alternative implementation would be to use the parallel data input registers (PGPDI) in the same way that the GPODW_SWAPP_WRITE uses the output equivalent of these registers. This would allow the inputs to be read at the same point in time but would restrict the GPIO that could be used.

5.3.1.2.2 Double Read PWM Inputs

The SIUL module may be configured (via the appropriate SIUL_PCR*n*) to provide configuration and input direction of the input GPIO.

Rationale: To check that the configuration of the modules used by this safety function compare to the expected configuration and to validate that the two sets of read data correlate.

Implementation hint: The software tests that the application may implement are:

- ETIMER0_SWTEST_REGCRC
- ETIMER1_SWTEST_REGCRC
- SIUL_SWTEST_REGCRC

In addition, the double reads shall be compared by the application with the implementation of the following test:

- ETIMERI_SWTEST_CMP.

5.3.1.2.2.1 Implementation details

Rationale: To reduce the risk of cascading faults due to shared resources.

Implementation hint: The following hardware elements shall be used for the safety function:

- eTimer_0 channels
- eTimer_1 channels

The system integrator may select one channel from the eTimer_0 module and another from the eTimer_1.

5.3.1.2.2.2 ETIMERx_SWTEST_REGCRC and SIUL_SWTEST_REGCRC

See Section 5.2.7, Cyclic Redundancy Checker Unit (CRC) for <*module*>_SWTEST_REGCRC implementation details.

5.3.1.2.2.3 ETIMERI_SWTEST_CMP

This test is used to execute the comparison between the double reads of PWM inputs performed by two channels of different eTimer (for example, eTimer_0 and eTimer_1). The comparison may take into account possible approximation because of different capturing of the asynchronous input signals.

5.3.1.2.3 Double Read Encoder Inputs

Rationale: To reduce the risk of cascading faults due to shared resources.

Implementation hint: One channel may be addressed by one eTimer, and the second channel by a different eTimer.

Rationale: To check that the configuration of the modules used by this safety function compare to the expected one. To reduce the likelihood of a common mode failure caused by improper configuration of the pads. To check that the two sets of read data compare properly.

Implementation hint: The SIUL is configured to forward one or two interrupt based event readings.

The application software shall implement the tests:

- ETIMER0_SWTEST_REGCRC
- ETIMER1_SWTEST_REGCRC
- SIUL_SWTEST_REGCRC

The application software shall implement the test ENCI_SWTEST_CMP, which compares signals acquired from each channel.

5.3.1.2.3.1 Implementation details

Rationale: To reduce the risk of cascading faults due to shared resources.

Implementation hint: The following hardware elements may be used for the safety function:

- eTimer_0 channels
- eTimer_1 channels
- External interrupt via GPIO

The system integrator may select one channel from eTimer_0 and one from eTimer_1. The external interrupt signals are optional.

5.3.1.2.3.2 ETIMERx_SWTEST_REGCRC and SIUL_SWTEST_REGCRC

See Section 5.2.7, Cyclic Redundancy Checker Unit (CRC) for <*module*>_SWTEST_REGCRC implementation details.

5.3.1.2.3.3 ENCI_SWTEST_CMP

The ENCI_SWTEST_CMP test is used to compare the double reads performed by the eTimer channels 0 and 1 and/or the SIUL. The comparison may take into account approximation because of different captured values of the asynchronous input signals and the execution of interrupt based event readings. Approximation required by different behavior of the encoded inputs is handled at the application level.

5.3.1.2.4 Synchronize sequential read input

The synchronize sequential read input is implemented by the CTU, which generates the trigger for events according to the triggered mode or the sequential mode.

The CTU can be used if the synchronization of the reading of some inputs with some events is required (FlexPWMs, eTimers, and external signals, or any combination).

Safety requirement under certain preconditions: [SM_053] If the CTU is part of an application safety function, the system level functional safety integrity measures must be implemented to achieve required integrity. [end]

Rationale: To validate the integrity of the CTU.

Implementation hint: The following mix of hardware mechanisms and software safety integrity measures implemented at the application level provide respective functional safety integrity:

- CTU_HWSWTEST_TRIGGERNUM
- CTU_SWTEST_TRIGGERTIME
- CTU_HWSWTEST_TRIGGEROVERRUN
- CTU_HWSWTEST_ADCCOMMAND (only if the input is an analog signal)
- CTU_SWTEST_ETIMERCOMMAND
- CTU_HW_CFGINTEGRITY

5.3.1.2.4.1 Hardware element

The synchronize sequential read input is implemented by the CTU, which generates the trigger events according to one of the two operation modes shown in Figure 16.

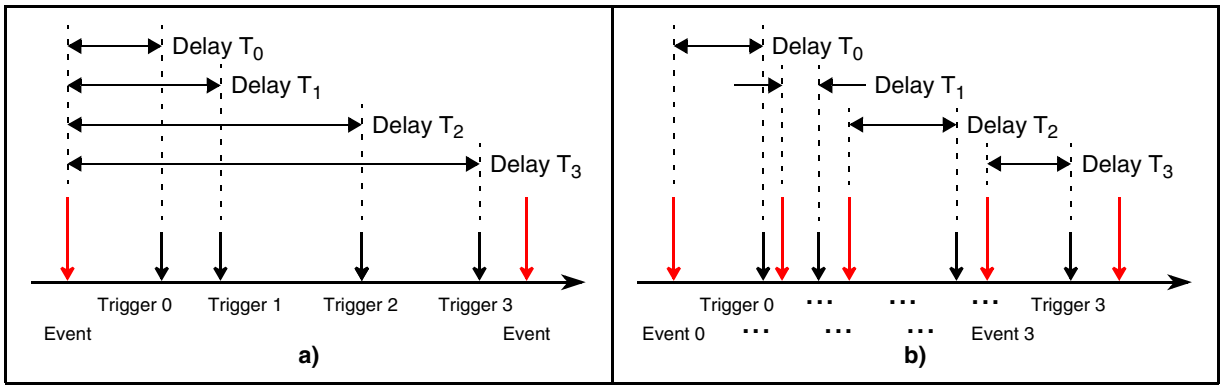


Figure 16. CTU operating modes: triggered a) and sequential b)

The CTU receives various incoming signals (Event X in Figure 16) from different sources (FlexPWMs, eTimers, or external GPIO, or any combination). These signals are then processed to generate trigger events (Trigger X in Figure 16). An event can be a rising edge, a falling edge or both edges of each incoming signal. The output trigger can be a pulse, an ADC command (or a stream of consecutive commands) or both to one or more peripherals (for example, ADC, eTimers, and so on).

In triggered mode, the input event, which can be also a combination (logical OR) of several signals, determines the reload/restart of the CTU counter and up to eight comparators are available to generate up to eight output triggers with a given delay with respect to the reload signal. In sequential mode, one comparator can be used to generate a trigger with a given delay with respect to one out of eight input events (Event 0 works as the reload event).

Implementation hint: The CTU is configured so that the output triggers are generated with the desired time schedule with respect to the input event(s).

For each trigger the set of ADC commands and pulses to be generated are defined.

Particularly, each ADC command specifies which channel is acquired by which ADC, if two ADCs perform a concurrent conversion or just one of them is operational, and in which CTU internal FIFO the result(s) will be stored. Four FIFOs are available ($2 \times 16 + 2 \times 4$). In case of a concurrent acquisition the same FIFO is used for both results. ADCs are configured to accept commands from the CTU (instead of commands provided via software). Multiple single or concurrent acquisitions can be scheduled for each trigger events (overall 24 commands per control period, for example, between two successive reload signals). The next command is sent when the ADC signals the completion of previous acquisition.

Recommendation: The CTU can be configured to generate interrupt requests when a trigger occurs (for example, to trigger READ DIGITAL INPUTS).

5.3.1.2.4.2 Implementation details

The following hardware elements may be used for the safety function:

- CTU
- One eTimer channel
- Another eTimer channel

Table 7. CTU software tests

Function	Test	Frequency
Synchronize sequential read input	CTU_HWSWTEST_TRIGGERNUM	Once for every control period (< FTTI)
	CTU_SWTEST_TRIGGERTIME	Once for every CTU control period (triggered mode) or every trigger (sequential mode)
	CTU_HWSWTEST_TRIGGEROVERRUN	Once for every trigger
	CTU_HWSWTEST_ADCCOMMAND	Once for every ADC command
	CTU_SWTEST_ETIMERCOMMAND	Once for every control period (< FTTI)
	CTU_HW_CFGINTEGRITY	Once for every control period (< FTTI)

5.3.2 Digital outputs

Functional safety digital outputs are always assumed to be written either redundantly or with read back. In case of single output with read back, the feedback loop should be as large as possible to cover faults on system level also. Figure 17 depicts the connection of two (functional safety critical) actuators connected to the MPC567xK. Actuator 1 is connected to an output peripheral, for example, a motor is connected to a PWM-output (output peripheral 3). The signal generated by the output peripheral 3 can be input to an input peripheral, for example, an eTimer. This measure is to confirm, that the generated output signal is correct. This read back may be internally of the MPC567xK (internal read back) or externally (external read back). The external read back covers more types of failures (for example, corrupt wire bonds or solder joints) than the internal read back, but still does not guarantee, that the actuator really behaves as desired. This is achieved by including the actuator and sensor into the read back loop. An alternative solution is to

redundantly output a signal. For example, the actuator 2 consists of two relays in series to switch off a functional safety relevant supply voltage. The selection of the suited output connection is part of the I/O functional safety concept on system level.

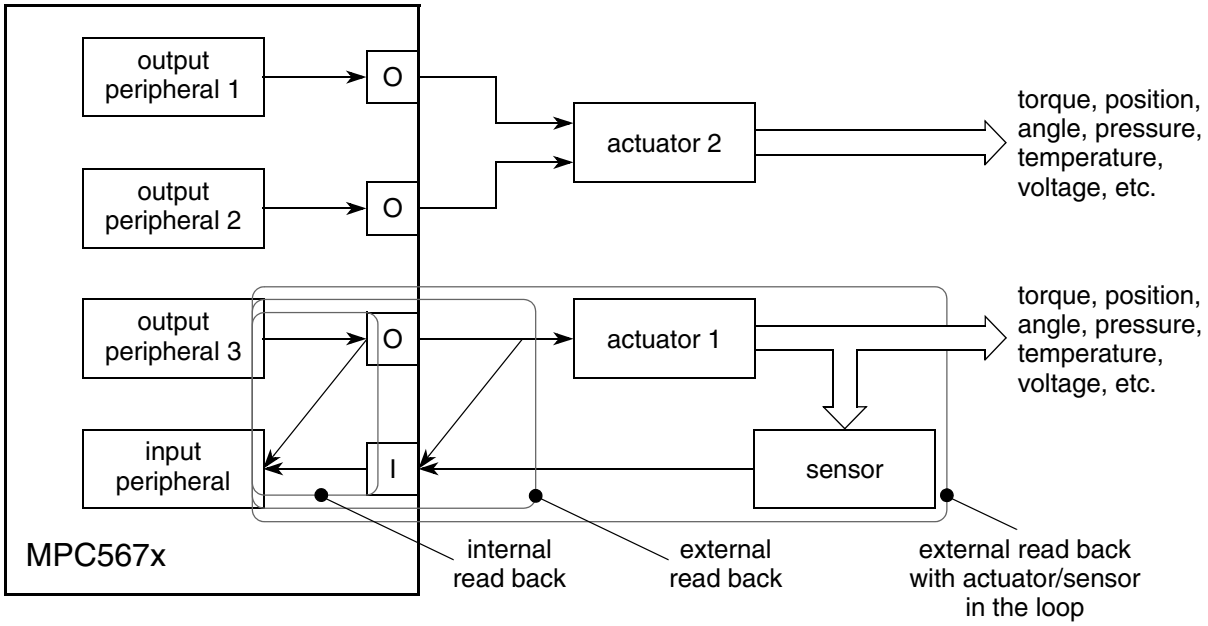


Figure 17. Digital Outputs with redundancy and read back

Implementation hint: If a sufficient diagnostic coverage can be reached by a plausibility check on a single output channel for a specific application, a plausibility check can replace a redundant write or a read back. This hint is a special case of deviating from **Safety requirements** as described in Section 1, Preface.

5.3.2.1 Hardware

5.3.2.1.1 Single Write Digital Output

- Single Write Digital Output with external read back:
  A comparison between the desired output values and the value read back via external read back configuration is done. After writing the output value, the status of the digital input is evaluated.
- Single Write Digital Output with internal read back[1]:
  A comparison between the desired output values and the value read back via internal read back configuration. After writing the output value, the internal read back status is evaluated.
- Single Write PWM Output with external read back:
  This procedure output compare the PWM read back provided by a single channel of the eTimer_0 (eTimer_1, eTimer_2) with the expected values that have been written to the external pad of the FlexPWM_1 (FlexPWM_2, FlexPWM_0) output channel.
- Single Write PWM Output with internal read back[1]:
  This procedure output compare the PWM read back provided by a single channel of the eTimer_0

1.Internal read back does not cover package faults (for example, wire bond, etc.).

(eTimer_1, eTimer_2) with the expected values that have been written to the FlexPWM_1 (FlexPWM_2, FlexPWM_0) output channel.

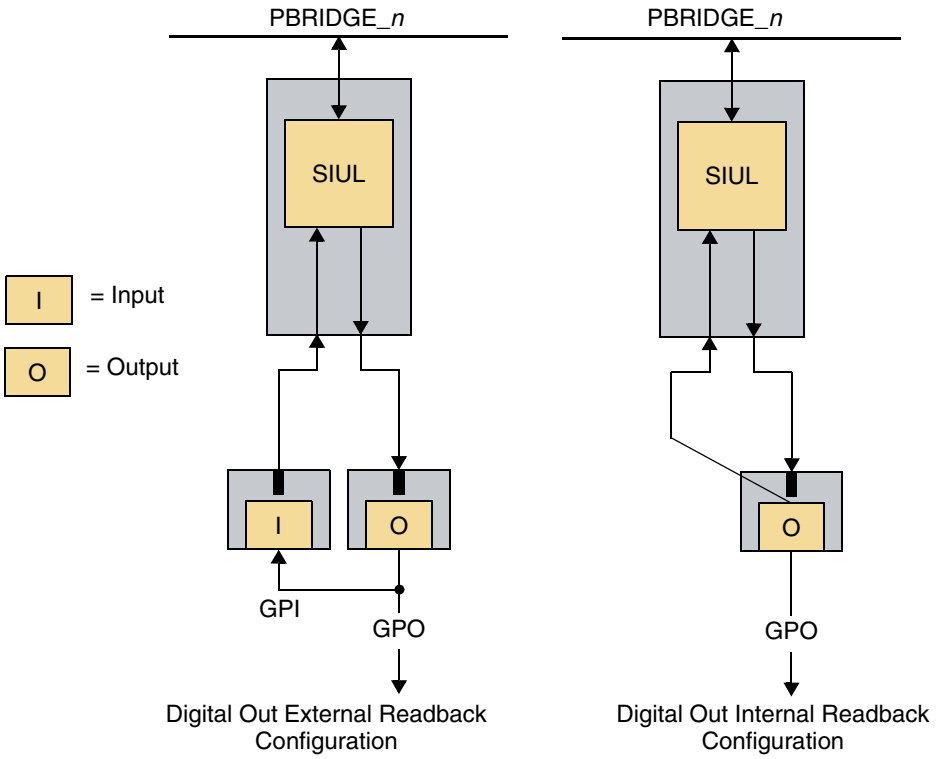
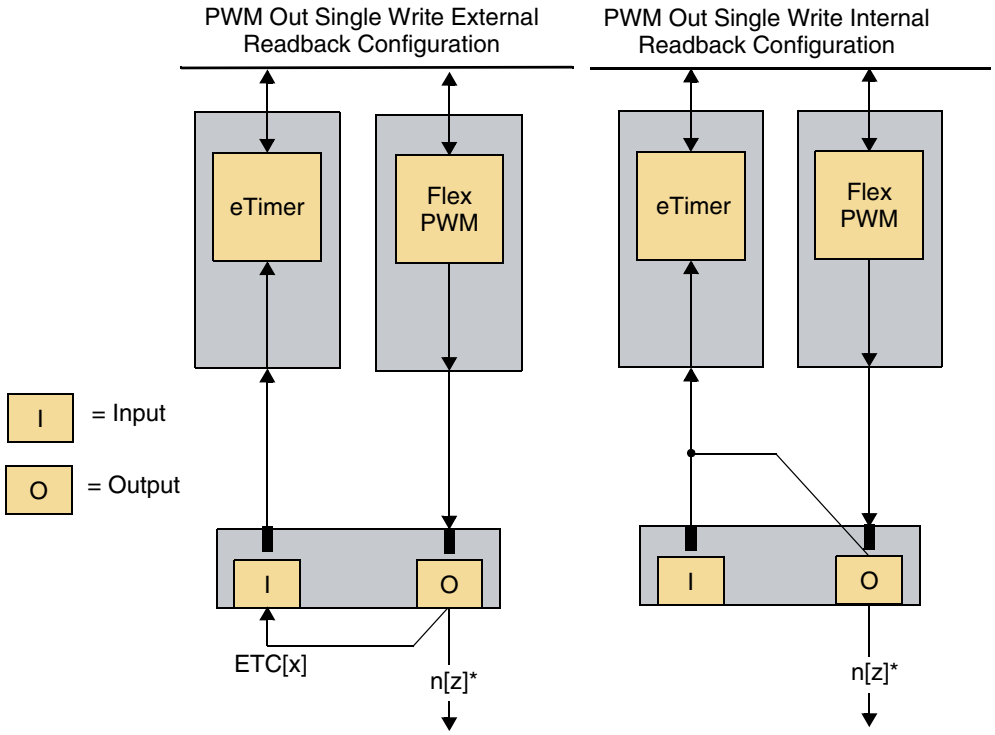


Figure 18. Single Write Digital Output With Read Back



*Note: n[z] represents any FlexPWM output (for example, PWMA[z], PWMB[z] or PWMX[z]).

Figure 19. Single Write PWM Output With Read Back

5.3.2.1.2 Double Write Digital Output

- Double Write Digital Output
  - The SIUL hardware element is used to perform a double write digital output.
- Double Write PWM Output
  - The hardware elements are used to perform a double write PWM output:
    - eTimer_0 and eTimer_1
    - eTimer_0 and eTimer_2
    - eTimer_1 and eTimer_2
    - FlexPWM_0 and FlexPWM_1
    - FlexPWM_0 and FlexPWM_2
    - FlexPWM_1 and FlexPWM_2

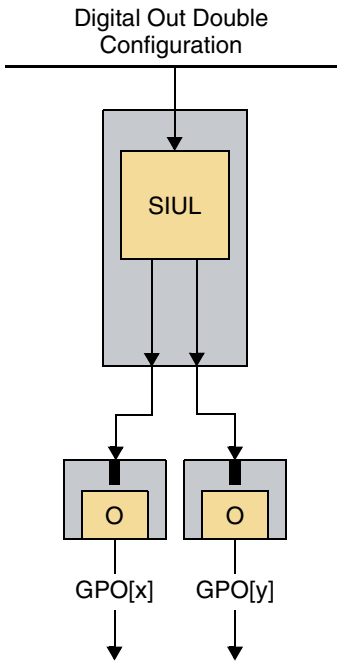
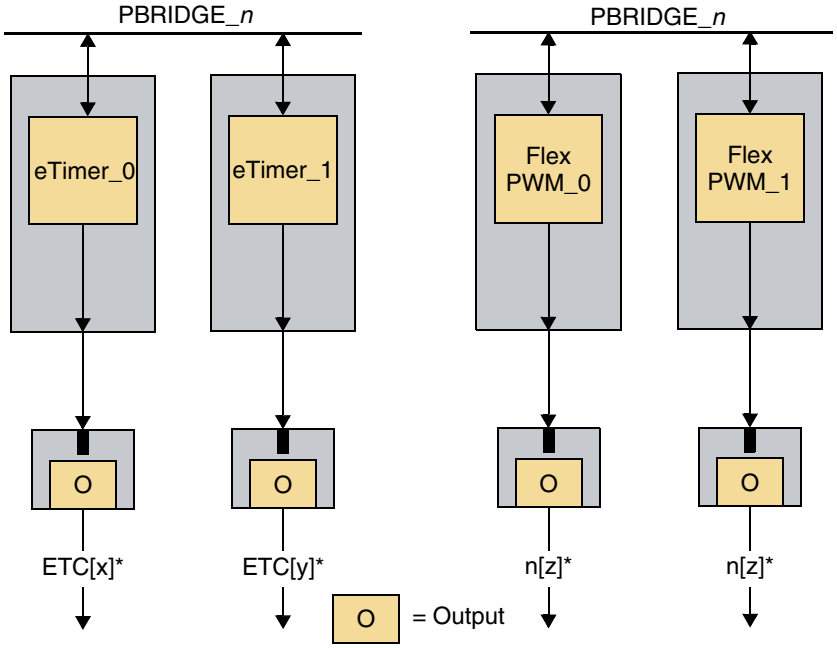


Figure 20. Double Write Digital Output



*Note: n[z] represents any FlexPWM output (for example, PWMA[z], PWMB[z] or PWMX[z]), but each output must be driven by different FlexPWM modules. The same consideration is valid for the eTimer; any eTimer output may be used, but each output must be driven by different eTimer module.

Figure 21. Double Write PWM Output

5.3.2.2 Software

Digital outputs used for functional safety purposes are assumed to be written either redundantly or with read back as described in this section. Table 8 lists four element safety functions for output, the corresponding safety integrity functions and their execution frequency. Alternative solutions with sufficient diagnostic coverage are possible.

Table 8. Digital outputs software tests

Function	Test	Frequency
Single Write Digital Outputs With Read Back	SIUL_SWTEST_REGCRC	Once after programming
	GPOERB_SWTEST_CMP	Once every write
	GPOIRB_SWTEST_CMP	Once every write
Double Write Digital Outputs	SIUL_SWTEST_REGCRC	Once after programming
	GPODW_SWAPP_WRITE	Once every write
Single Write PWM Outputs With Read Back	SIUL_SWTEST_REGCRC	Once after programming
	ETIMER0_SWTEST_REGCRC[1]	Once after programming
	ETIMER1_SWTEST_REGCRC[1]	Once after programming
	ETIMER2_SWTEST_REGCRC[1]	Once after programming
	FLEXPWM0_SWTEST_REGCRC[2]	Once after programming
	FLEXPWM1_SWTEST_REGCRC[2]	Once after programming
	FLEXPWM2_SWTEST_REGCRC[2]	Once after programming
	PWMRB_SWTEST_CMP	Once every write
Double Write PWM Outputs	SIUL_SWTEST_REGCRC	Once after programming[3]
	ETIMER0_SWTEST_REGCRC[1]	Once after programming
	ETIMER1_SWTEST_REGCRC[1]	Once after programming
	ETIMER2_SWTEST_REGCRC[1]	Once after programming
	FLEXPWM0_SWTEST_REGCRC[2]	Once after programming
	FLEXPWM1_SWTEST_REGCRC[2]	Once after programming
	FLEXPWM2_SWTEST_REGCRC[2]	Once after programming
	PWMDW_SWAPP_WRITE	Once every write

NOTES:
1 This test is required only if the eTimer channels are used for the safety function.
2 This test is required only if the FlexPWM channels are used for the safety function.
3 If a change in a single SIUL configuration register is capable of affecting both the output and the read-back paths, then SIUL_SWTEST_REGCRC may be executed every FTTI. In all other cases configuration errors are covered by the software comparison.

5.3.2.2.1 Single Write Digital Outputs With Read Back

The SIUL hardware element is used to perform a Single Write Digital Output With Read Back (see Figure 18).

Rationale: To check if written data is coherent to the expected data.

Implementation hint: The read back may be implemented either with external or with internal readback.

The SIUL element is correctly configured to provide the output write and the pad directions as follows:

- External read back – SIUL is configured to read back the signal from an additional pad, and the loopback is performed outside the device. In this configuration, only half of the available digital outputs are available as functional safety outputs.
- Internal read back – SIUL is configured to read back the pad value via an internal read path. All pads dedicated to digital I/O are capable of reading the pad digital status using the input logic.

Rationale: To reduce the likelihood of a CMF caused by incorrect configuration of pads

Implementation hint: The application software integrates software test SIUL_SWTEST_REGCRC in the application to check the correct configuration of the pads, and to compare a read back with the digital output write. GPOERB_SWTEST_CMP may be used for the external read back or GPOIRB_SWTEST_CMP for internal read back.

5.3.2.2.1.1 Implementation details

The SIUL hardware element may be used for the safety function.

NOTE

Pads that are not dedicated to a single function can be configured as GPIO. Pads dedicated to ADC are an exception to this rule, as they can only be configured as inputs.

5.3.2.2.1.2 SIUL_SWTEST_REGCRC

See Section 5.2.7, Cyclic Redundancy Checker Unit (CRC) for <*module*>_SWTEST_REGCRC implementation details.

5.3.2.2.1.3 GPOERB_SWTEST_CMP

This software test is used to execute the comparison between the desired output values and the value read back via external read back configuration. After writing the output value, the test reads the status of the digital input.

Rationale: To check if the read data equals the written data

Implementation hint: The output is externally (on system level) connected to an input I/O. After writing the value to the output signal, the input is read to check that the correct output is present.

5.3.2.2.1.4 GPOIRB_SWTEST_CMP

This software test is used to execute the comparison between the desired output values and the value read back via internal read back configuration. After writing the output value, the test reads the status.

Rationale: To check if the read data equals the written data.

5.3.2.2.2 Double Write Digital Outputs

The SIUL hardware element is used to perform a Double Write Digital Output.

Rationale: To configure pads used by this safety function and reduce the likelihood of a CMF caused by incorrect configuration of pads

Implementation hint: The SIUL is configured by application software to correctly define the configuration of the outputs used. The software performs a double write.

Rationale: To reduce the likelihood of a CMF caused by incorrect configuration of the pads

Implementation hint: To achieve the integrity of the two output channels, the application validates the SIUL configuration implementing the SIUL_SWTEST_REGCRC.

Rationale: To write a digital output by exploiting redundancy

Implementation hint: The application software implements the double output write as defined by the GPODW_SWAPP_WRITE.

5.3.2.2.2.1 Implementation details

The only hardware element that can be used for the safety function is the GPIO.

Every pad not dedicated to a single function may be configured as GPIO. Pads dedicated to ADC are an exception to this rule, as they can be configured as inputs only.

5.3.2.2.2.2 GPODW_SWAPP_WRITE

Rationale: To minimize the common mode failure of the SIUL

Implementation hint: The output write of a redundant channel may be implemented by writing the two outputs with a single instruction to the appropriate register and this register may be checked by read back.

To write two or more GPIOs with a single instruction, the Masked Parallel GPIO Pad Data Out register (SIUL_MPGPDO*n*) register can be used. The two GPIOs used must be in the same SIUL_MPGPDO*n* register.

To protect the value of the other GPIOs that belong to the same SIUL_MPGPDO*n*, the MASK field of the SIUL_MPGPDO*n* register needs to be properly configured.

When using a single write (atomic) instruction to SIUL_MPGPDO*n* register, it is good practice to read back (read after write) the register content due to the fact that a transient fault in the SIUL IPS interface can affect in principle both output channels. The readback is needed to cover this common mode of failure. An alternative implementation would be to write the two outputs separately not using the parallel register, resulting in a small delay in output change between the channels.

5.3.2.2.3 Single Write PWM Outputs With Read Back

The following combination of elements may be used to perform a Write PWM Output With Read Back:

- eTimer_0 – FlexPWM_0

- eTimer_0 – FlexPWM_1
- eTimer_0 – FlexPWM_2
- eTimer_0 – FlexPWM_3
- eTimer_1 – FlexPWM_0
- eTimer_1 – FlexPWM_1
- eTimer_1 – FlexPWM_2
- eTimer_1 – FlexPWM_3
- eTimer_2 – FlexPWM_0
- eTimer_2 – FlexPWM_1
- eTimer_2 – FlexPWM_2
- eTimer_2 – FlexPWM_3
- eTimer_3 – FlexPWM_0
- eTimer_3 – FlexPWM_1
- eTimer_3 – FlexPWM_2
- eTimer_3 – FlexPWM_3

These units are configured to implement one PWM output channel and (via internal read back) the eTimer_0 input PWM channel. The SIUL is configured to define the configuration of the output pads used. The software performs a write operation followed by a read operation. To achieve the integrity of the two output channels, the application shall tests the SIUL configuration implementing the SIUL_SWTEST_REGCRC (to reduce the likelihood of a common mode failure caused by incorrect configuration of the pads).

Rationale: To check that the configuration of the modules used by this safety function adheres to the expected configuration.

Implementation hint: A single channel of the eTimer is used with a multiplexing of the internal read back of the different output of the FlexPWM. The read back paths are limited to six signals, two for each sub-module of the FlexPWM.

The following tests validate correct configurations:

- FLEXPWM0_SWTEST_REGCRC
- FLEXPWM1_SWTEST_REGCRC
- FLEXPWM2_SWTEST_REGCRC
- ETIMER0_SWTEST_REGCRC
- ETIMER1_SWTEST_REGCRC
- ETIMER2_SWTEST_REGCRC

Rationale: To check that the written data is what is expected.

Implementation hint: The application software writes to the output port and then compare the written value via the read back (PWMRB_SWTEST_CMP).

5.3.2.2.3.1 Implementation details

The following hardware elements may be used for the safety function:

- eTimer_0 channels
- eTimer_1 channels
- eTimer_2 channels
- FlexPWM_0 channels
- FlexPWM_1 channels
- FlexPWM_2 channels

5.3.2.2.3.2 FLEXPWMx_SWTEST_REGCRC and ETIMERx_SWTEST_REGCRC

See Section 5.2.7, Cyclic Redundancy Checker Unit (CRC) for <*module*>_SWTEST_REGCRC implementation details.

5.3.2.2.3.3 PWMRB_SWTEST_CMP

This test compares the PWM read back provided by a single channel of the eTimer_1 (eTimer_0) with the expected values that have been written to the FlexPWM_0 (FlexPWM_1) output channel.

For this example, FlexPWM_0 is used to generate a PWM output and eTimer_1 is used to read back and verify the output. Another combination could be used if required in an application.

5.3.2.2.4 Double Write PWM Outputs

Rationale: The hardware elements eTimer_0, eTimer_1 and eTimer_2 or FlexPWM_0, FlexPWM_1 and FlexPWM_2 are used to perform a double Write PWM Output.

Implementation hint: These units are configured to implement two independent PWM channels. The SIUL is configured to define the configuration of the output pads used. The software performs a double write (see Section 5.3.2.2.4.3, PWMDW_SWAPP_WRITE).

Rationale: To reduce the risk of cascading faults

Implementation hint: Using adjacent pads as redundant I/O increases the likelihood of CMFs. Therefore, it is preferable to use I/O that do not share the same configuration and data registers in the SIUL.

Rationale: To reduce the likelihood of a CMF caused by incorrect configuration of the pads

Implementation hint: To improve the integrity of the two output channels, the application should test the SIUL configuration implementing the SIUL_SWTEST_REGCRC.

Rationale: To check that the configuration of the modules used by this safety function adheres to the expected configuration

Implementation hint: The application software shall implement a test for the register configuration:

- ETIMER0_SWTEST_REGCRC (for eTimer)
- ETIMER1_SWTEST_REGCRC (for eTimer)
- ETIMER2_SWTEST_REGCRC (for eTimer)

- FLEXPWM0_SWTEST_REGCRC (for FlexPWM)
- FLEXPWM1_SWTEST_REGCRC (for FlexPWM)
- FLEXPWM2_SWTEST_REGCRC (for FlexPWM)

Rationale: To reduce the possibility of cascading a failure to shared circuitries, different modules should be used.

Implementation hint: The output write of a redundant PWM channel is implemented by writing the new output values to both PWM channels. The system integrator can decide whether to use two of the three eTimers (eTimer_0, eTimer_1, eTimer_2) or two of the three FlexPWMs (FlexPWM_0, FlexPWM_1, FlexPWM_2).

5.3.2.2.4.1 Implementation details

The following hardware elements are used for the safety function:

- eTimer_0 channels
- eTimer_1 channels
- eTimer_2 channels
- FlexPWM_0 channels
- FlexPWM_1 channels
- FlexPWM_2 channels

5.3.2.2.4.2 SIUL_SWTEST_REGCRC

See Section 5.2.7, Cyclic Redundancy Checker Unit (CRC) for <*module*>_SWTEST_REGCRC implementation details.

5.3.2.2.4.3 PWMDW_SWAPP_WRITE

If the content of the PWM outputs are changed, care must be taken since the outputs can not be updated synchronously. Therefore for a short period of time both outputs could be different.

5.3.3 Analog inputs

5.3.3.1 Hardware

Two options for reading analog inputs exist:

- Single Read Analog Inputs
- Double Read Analog Inputs

Apart from these hardware BISTs, tests may be implemented in software as described in section Section 5.3.3.2.1, Single Read Analog Inputs and Section 5.3.3.2.2, Double Read Analog Inputs.

Oversampling can be used to detect transient faults affecting the ADC channel during normal operation.

It is important to note that the ADC is part of the temperature measuring safety integrity function, and it is therefore required that the ADC hardware BIST functions be executed even if the ADC is not in application use.

5.3.3.1.1 Single Read Analog Inputs

The single-read analog input uses a single-analog-input channel either of ADC_0, ADC_1, ADC_2, or ADC_3 to acquire an analog voltage signal (see Figure 22).

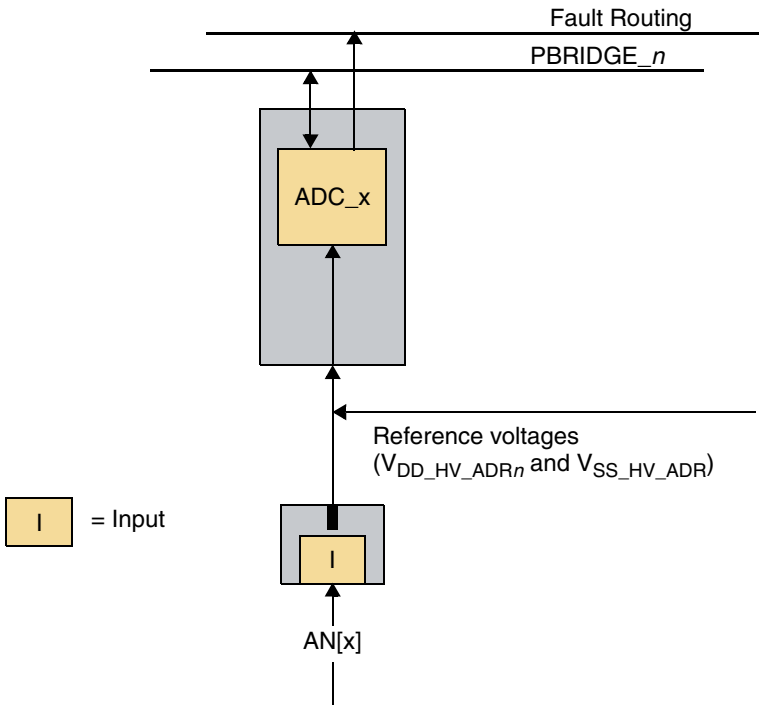


Figure 22. Single Read Analog Input configuration

5.3.3.1.2 Double Read Analog Inputs

The Double Read Analog Input uses two analog input channels to acquire a replicated analog input signal. Two ADC units acquire and digitize the two copies of a redundant analog signal connected to the inputs. In this configuration (if applied to all possible analog inputs), only half of the analog inputs are available to the applications (AN[0:8] of ADC_0 for signals, and AN[0:8] of ADC_1 for signal copies). The comparison of the results is performed by the system level application software (see Figure 23).

Rationale: ADC_0 and ADC_1 as also ADC_2 and ADC_3 share input for the channels (AN[11:14]). Using double reads is a possible source of CMFs.

Implementation hint: One shared ADC channel (AN[11:14]) may not be used for both inputs of the double read analog input function.

The usage of one input of one channel AN[11:14] in combination with another channel AN[0:8] is possible.

The functional safety integrity is achieved by replicated acquisition with separated analog input channels and software comparison by the processing function (see Figure 23).

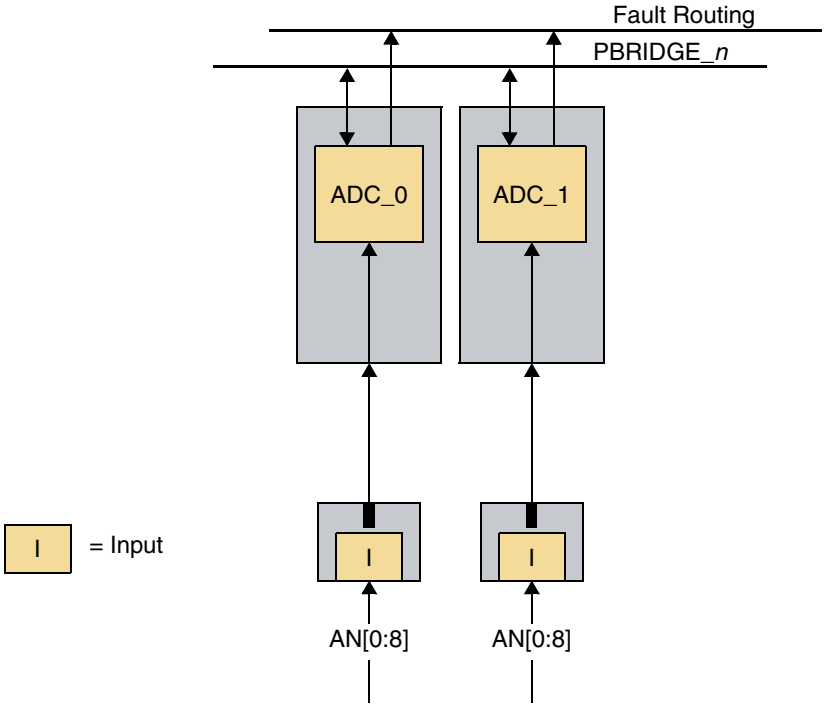


Figure 23. Double Read Analog Inputs configuration

5.3.3.2 Software

Analog inputs used for functional safety purposes are assumed to be input redundantly as described in this section. Table 9 lists two element safety functions for analog input, the corresponding safety integrity functions and their execution frequency. Alternative solutions with sufficient diagnostic coverage are possible.

It is important to note that the ADC is part of the temperature measuring safety integrity function, and it is therefore required that the hardware BIST functions be executed once after the boot even if the ADC is not in application use.

Table 9. Analog inputs software tests

Function	Test	Frequency
Single Read Analog Inputs	SUPPLY SELF-TEST	Once in the FTTI
	RESISTIVE-CAPACITIVE SELF-TEST	Once in the FTTI
	CAPACITIVE SELF-TEST	Once in the FTTI
	ADC_SWTEST_TEST1	Once in the FTTI
	ADC_SWTEST_TEST2	Once in the FTTI
	ADC_SWTEST_VALCHK	Once for every acquisition
	ADC_SWTEST_OVERSAMPLING	Once for every acquisition
	ADC0_SWTEST_REGCRC	Once in the FTTI
	ADC1_SWTEST_REGCRC	Once in the FTTI
	ADC2_SWTEST_REGCRC	Once in the FTTI
	ADC3_SWTEST_REGCRC	Once in the FTTI
	SIUL_SWTEST_REGCRC	Once in the FTTI
Double Read Analog Inputs	SUPPLY SELF-TEST	Once after boot
	RESISTIVE-CAPACITIVE SELF-TEST	Once after boot
	CAPACITIVE SELF-TEST	Once after boot
	ADC0_SWTEST_REGCRC	Once after programming
	ADC1_SWTEST_REGCRC	Once after programming
	ADC2_SWTEST_REGCRC	Once after programming
	ADC3_SWTEST_REGCRC	Once after programming
	SIUL_SWTEST_REGCRC	Once after programming
	ADC_SWTEST_CMP	Once for every acquisition

5.3.3.2.1 Single Read Analog Inputs

To support a high diagnostic coverage two known reference supply voltages are utilized by two software tests which are described in the following sections (ADC_SWTEST_TEST1 and ADC_SWTEST_TEST2).

The reference supply voltages are the following:

- $V_{DD\_HV\_ADR0}$ (ADC_0 high reference voltage)
- $V_{DD\_HV\_ADR1}$ (ADC_1 high reference voltage)
- $V_{DD\_HV\_ADR2}$ (ADC_2 high reference voltage)
- $V_{DD\_HV\_ADR3}$ (ADC_3 high reference voltage)
- $V_{SS\_HV\_ADR0}$ (ADC_0 low reference voltage)

- $V_{SS\_HV\_ADR1}$ (ADC_1 low reference voltage)
- $V_{SS\_HV\_ADR2}$ (ADC_2 low reference voltage)
- $V_{SS\_HV\_ADR3}$ (ADC_3 low reference voltage)

The SIUL unit is configured to correctly enable the ADC inputs. The pads used for analog inputs can only be configured as inputs.

Single Read Analog Inputs may be implemented using the following safety integrity functions at the application level:

- ADC_SWTEST_TEST1
- ADC_SWTEST_TEST2
- ADC_SWTEST_VALCHK
- ADC0_SWTEST_REGCRC, ADC1_SWTEST_REGCRC, ADC2_SWTEST_REGCRC, or ADC3_SWTEST_REGCRC
- SIUL_SWTEST_REGCRC
- ADC_SWTEST_OVERSAMPLING

5.3.3.2.1.1 Implementation details

The following hardware elements can be used for the safety integrity functions:

- Analog input channels AN[0:8] of ADC_0
- Analog input channels AN[11:14] of ADC_0 and ADC_1 (shared channels)
- Analog input channels AN[0:8] of ADC_1
- Analog input channels AN[0:8] of ADC_2
- Analog input channels AN[11:14] of ADC_2 and ADC_3 (shared channels)
- Analog input channels AN[0:8] of ADC_3

The system integrator may select one channel from different ADC modules, for example, ADC_0 or from ADC_1. Shared channels can be used.

5.3.3.2.1.2 SIUL_SWTEST_REGCRC

See Section 5.2.7, Cyclic Redundancy Checker Unit (CRC) for <*module*>_SWTEST_REGCRC implementation details.

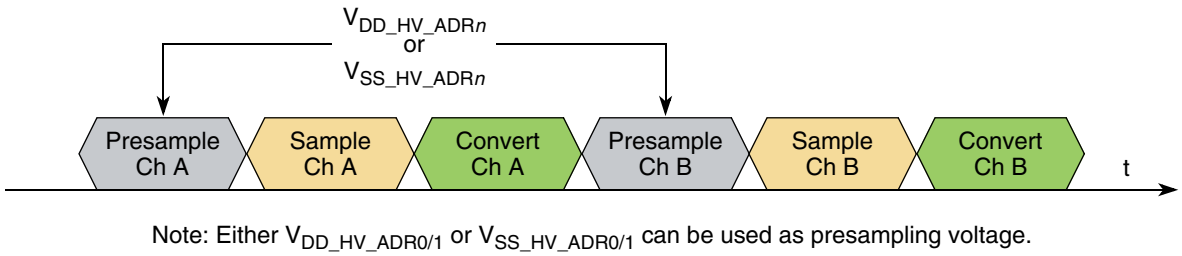
5.3.3.2.1.3 ADC*n*_SWTEST_REGCRC

If ADC_0 is used the ADC0_SWTEST_REGCRC may be used. If ADC_1 is used the ADC1_SWTEST_REGCRC may be used. If ADC_2 is used the ADC2_SWTEST_REGCRC may be used. If ADC_3 is used the ADC3_SWTEST_REGCRC may be used.

See Section 5.2.7, Cyclic Redundancy Checker Unit (CRC) for <*module*>_SWTEST_REGCRC implementation details.

5.3.3.2.1.4 ADC_SWTEST_TEST1 (open detection)

This test exploits the presampling feature of the ADC. Presampling allows to precharge or discharge of the ADC internal capacitor before it starts the sampling and conversion phases of the analog input received from the pads. During the presampling phase, the ADC samples the internally generated voltage. While in the sampling phase, the ADC samples analog input coming from the pads. In the conversion phase, the last sampled value is converted to a digital value. Figure 24 shows the normal sequence of operation for two channels (Presampling – Sampling – Conversion).



Note: Either $V_{DD\_HV\_ADR0/1}$ or $V_{SS\_HV\_ADR0/1}$ can be used as presampling voltage.

Figure 24. Implementation of ADC_SWTEST_TEST1

Reference voltages, which can be used during presampling phase, is either $V_{DD\_HV\_ADR}$ or $V_{SS\_HV\_ADR}$.

If there is an open failure in the analog multiplexing circuitry, the signal converted by the ADC is not the analog input coming from the pad, but the presampling reference voltage ($V_{DD\_HV\_ADRn}$ or $V_{SS\_HV\_ADRn}$). Figure 25 depicts the signal path in the analog multiplexing circuitry for presampling phase and conversion phase.

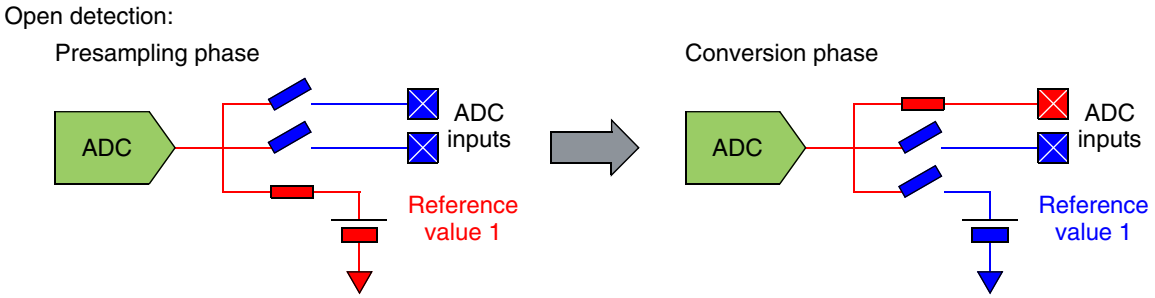


Figure 25. ADC_SWTEST_TEST1 (open detection)

Each analog input channel used by the safety function may be tested by system level measures (software).

Since the pads dedicated to analog inputs are of type INPUT, a missing enable from the SIUL results in an open failure.

Rationale: To detect open failures of the channel multiplexing circuitry (see Figure 25).

Implementation hint: Presampling can be enabled on a per channel basis through the ADC_*n*_PSR0 register. ADC_*n*_PCSR[PREVAL0] selects which reference voltage is used to precharge/discharge the ADC internal capacitor, (ADC_*n*_PSCR[PRECONV] = 0). (See “Analog-to-Digital Converter (ADC)” chapter in the *Qorivva MPC5675K Microcontroller Reference Manual* for details on the presampling feature).

NOTE

Caution! To reduce the likelihood of a false indication of an open fault in the analog multiplexor, signals connected to the ADC inputs should not be outside of the limits of the reference voltages ($V_{DD\_HV\_ADR}$, $V_{SS\_HV\_ADR}$). In case this limitation cannot be fulfilled by the application, a more complex algorithm may be necessary (for example, run the test three times with $V_{DD\_HV\_ADR}$, $V_{SS\_HV\_ADR}$, $V_{DD\_HV\_ADR}$).

5.3.3.2.1.5 ADC_SWTEST_TEST2 (short detection)

To detect short failures two different voltages are acquired by the ADC. If these values are different from the expected ones, a short failure on the multiplexed circuitry has been detected.

To implement this test a presampling feature of the ADC can be exploited. The presampling may be configured in such a way that the sampling of the channel is bypassed and the presampling reference supply voltages are converted.

During the first step the $V_{DD\_HV\_ADRn}$ is converted and compared with the expected value; then the $V_{SS\_HV\_ADRn}$ is converted and compared with the expected one (see Figure 26).

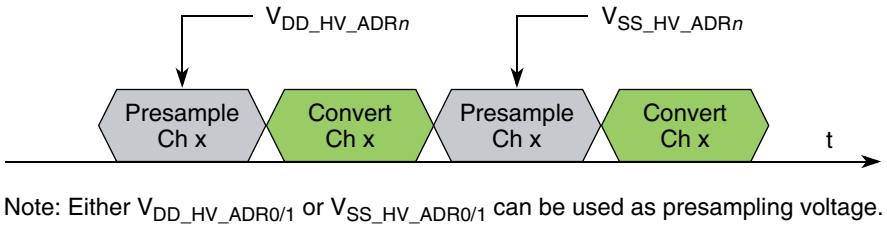


Figure 26. Implementation of ADC_SWTEST_TEST2

Rationale: To detect short failures of the channel multiplexing circuitry (see Figure 27).

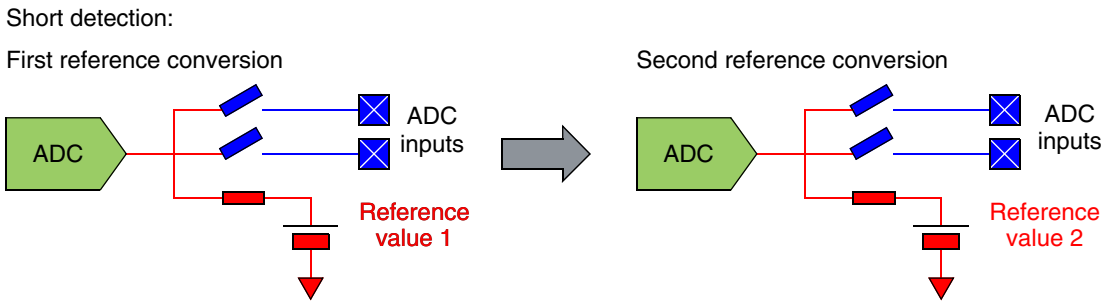


Figure 27. ADC_SWTEST_TEST2 (short detection)

Implementation hint: Presampling can be enabled on a per channel basis through the ADC_*n*_PSR0 register. ADC_*n*_PCSR[PREVAL0] selects which reference voltage is used to precharge/discharge the ADC internal capacitor. To bypass the conversion of the input channel and convert the presampled values, ADC_*n*_PCSR[PRECONV] = 1. (See “Analog-to-Digital Converter (ADC)” chapter in the *Qorivva MPC5675K Microcontroller Reference Manual* for details on the presampling feature).

5.3.3.2.1.6 ADC_SWTEST_VALCHK

When ADC conversion is triggered by the CTU, the acquired digital sample data are stored into a dual queue along with information about the channel that performed the acquisition. The checking of the expected channel provides coverage of the control logic and part of the queue logic. Checking of the expected sequence of acquired channels provides the coverage of the control logic and part of the queue logic.

Implementation hint: If ADC is configured to work in CTU mode, the conversion results are stored in CTU FIFOs (see CTU chapter in *Qorivva MPC5675K Microcontroller Reference Manual* for details). Along with the converted data, the converted channel number and ADC module are stored. CTU includes two sets of registers to read this information (FIFO Right aligned data, FRx, and FIFO Right aligned data, FLx). These registers may be read to check that the sequence of the acquired channel is what is expected.

5.3.3.2.1.7 ADC_SWTEST_OVERSAMPLING

In case of Single Read Analog Inputs the ADC_SWTEST_OVERSAMPLING may be implemented as counter measure against random fault.

ADC_SWTEST_OVERSAMPLING is an acquisition redundant in time.

It refers to sampling the signal at a rate significantly higher than the Nyquist frequency related to the input signal. If there is a fault, the acquired values will not be correlated.

This safety integrity measure compares the acquired value to check the correlation.

Against random fault, three consecutive analog values are converted for each acquisition to implement the ADC_SWTEST_OVERSAMPLING. Figure 28 shows the sampling of an analog signal at different points in time (A_1, A_2 and A_3). Every conversion is indicated by an arrow, which indicates the converted digital value by its length. The second acquisition (A_2) is faulty because the first converted value is quite different respect the other two.

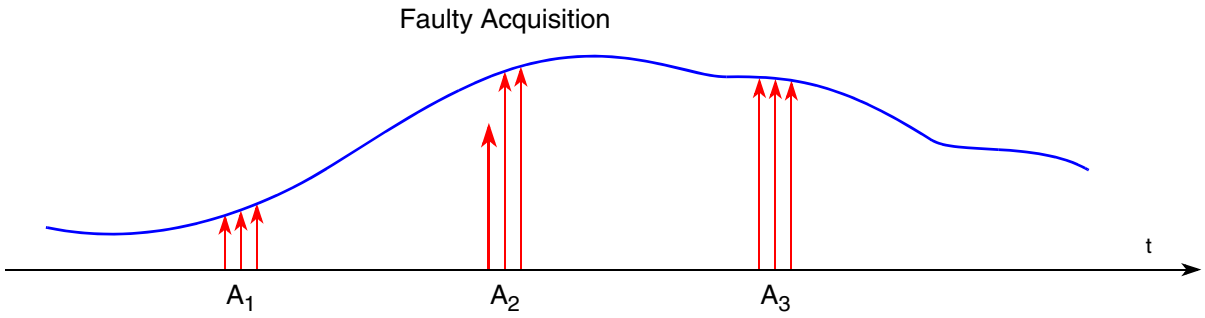


Figure 28. Series of acquired analog values

5.3.3.2.2 Double Read Analog Inputs

Rationale: To validate that the configuration of the modules used by this safety function corresponds with what is expected. To reduce the likelihood of common mode failures caused by improper configuration of the pads.

Implementation hint: Double Read Analog Inputs may be implemented using the following safety integrity functions at the application level:

- ADC0_SWTEST_REGCRC
- ADC1_SWTEST_REGCRC
- ADC2_SWTEST_REGCRC
- ADC3_SWTEST_REGCRC
- SIUL_SWTEST_REGCRC

Rationale: To validate that the two sets of read data correlate.

Implementation hint: Double Read Analog Inputs may be implemented using the software test ADC_SWTEST_CMP to compare the channel reads.

5.3.3.2.2.1 Implementation details

The following hardware elements may be used for the safety function:

- Analog input channels AN[0:8] of ADC_0
- Analog input channels AN[0:8] of ADC_1
- Analog input channels AN[0:8] of ADC_2
- Analog input channels AN[0:8] of ADC_3

One channel from different ADC modules may be used, for example, one from ADC_0 module and one from the ADC_1 module.

5.3.3.2.2.2 SIUL_SWTEST_REGCRC

See Section 5.2.7, Cyclic Redundancy Checker Unit (CRC) for <*module*>_SWTEST_REGCRC implementation details.

5.3.3.2.2.3 ADC_SWTEST_CMP

This software test is used to execute the comparison between the double reads performed by any combination of two ADC_*n* module channels. The comparison may take possible conversion tolerances into account.

5.3.4 Other requirements

Rationale: To detect missing eTimer acquisition.

Implementation hint: In the eTimer module, the capture flag (eTimer_*n*_STS[ICF*n*]) may be used.

Rationale: To detect stalled quadrature counting.

Implementation hint: When using the eTimer counter to decode a primary and secondary external input as quadrature encoded signals, the eTimer watchdog may be used (see the “Counting Modes” section of the *Qorivva MPC5675K Microcontroller Reference Manual*). eTimer watchdog is only available for channel 0.

Implementation hint:

- When an application needs to access the ADC result FIFO, a 32-bit read access enables the verification of the correct channel number on which the conversion was executed.
- All the FIFO empty interrupt flags should checked when the motor control period interrupt occurs
- In the eTimer module, the capture flag should be used to detect missing eTimer acquisition.
- If the ADC analog watchdog function is used for functional safety relevant signal, two analog watchdog channels should monitor the same signal.
- If Sine Wave Generator (SWG) is used, the ADC (eventually in conjunction with CTU) should be used to check the output signal.
- If an external temperature sensor is used to validate the accuracy of the internal temperature sensor, the external temperature sensor may not be converted by the same ADC that was used to convert the internal temperature value (ADC 0).

5.4 Communications

5.4.1 Redundant communication

Parts of the integrated DSPI, LINFlex, and I^2C communication controller do not provide the functional safety integrity IEC 61508 series and ISO 26262 requires for high functional safety integrity targets. As these communication protocols often deal with low complex slave communication nodes, higher level functional safety protocols as described in Chapter 5.4.2, “Fault-tolerant communication protocol” may not be feasible. Therefore appropriate communication channel redundancy may be required. Multiple instances of communication controller may be used to build up a single fault robust communication link.

Implementation hint: In case the communication over the following interfaces is integral part of the safety function, multiple instances of the replicated hardware communication controller are implemented redundantly, preferable using different data coding, for example, inversion, if using:

- Synchronous Serial Communication Controller (DSPI)
- LINFlex Communication Controller
- I^2C Communication Controller

DSPI, LINFlexD, and I^2C do not have special functional safety mechanisms other than what is included into them by their protocol specifications. The system level communication architecture needs to provide the functional safety mechanisms on the interface of the modules to meet functional safety requirements.

5.4.2 Fault-tolerant communication protocol

Parts of the integrated FlexRay, FlexCAN, and Fast Ethernet communication channel do not provide the functional safety integrity IEC 61508 series and ISO 26262 requires for high functional safety integrity targets.

Implementation hint: In case the communication over the following interfaces is integral part of the functional safety function, a software interface with the hardware communication channel in accordance with the IEC 61784-3 or IEC 62280 series is required for:

- FlexRay Communication Controller
- FlexCAN Communication Controller
- Fast Ethernet Communication Controller (FEC)
- Universal Asynchronous Communication Controller (LINFlex)

FlexRay, FEC, FlexCAN, and Universal Asynchronous Communication Controller (LINFlex) do not have specific functional safety mechanisms other than ECC protection of SRAM arrays and what is included into them by their protocol specifications. The application software, middleware software, or operating system needs to provide the functional safety mechanisms on the interface of the IP modules to meet functional safety requirements.

Typically mechanisms are:

- end-to-end CRC to detect data corruption
- sequence numbering to detect message repetitions, deletions, insertions, and resequencing
- an acknowledgement mechanism or time domain multiplexing to detect message delay
- sender identification to detect masquerade

As the black channel typically includes the physical layer (for example, communication line driver, wire, connector), the functional safety software protocol layer is an end-to-end functional safety mechanism from message origin to message destination.

Appropriate functional safety software protocol layer (for example, Fault Tolerant Communication Layer, FTCOM, CANopen Safety Protocol) may be necessary to ensure the failure performance of the communication process. Software protocol layer implements an software interface with the hardware communication channel in accordance with the IEC 61784-3 or IEC 62280 series (so-called ‘black channel’).

An alternative approach to improve the functional safety integrity of FlexCAN may use multiple instances of the FlexCAN channels and use an appropriate protocol to redundantly communicate data, for example, using the CANopen Safety protocol. This approach communicates redundant data (for example, one message payload inverted, the other message payload not inverted) using different communication controller.

Due to the limited bandwidth and the point to point communication architecture for Universal Asynchronous Communication Controller (LINFlex) a simplified functional safety protocol layer may be only required.

5.5 Additional configuration information

5.5.1 Call stack

Call stack overflow and call stack underflow is a common mode fault due to systematic faults within application software. A stack overflow occurs when using too much memory (pushing to much date) on

the call stack. A stack underflow occurs when reading (pop) too much data from memory. The call stack contains a limited amount of memory, often determined during development of the application software. When a program attempts to use more space than is reserved (available) on the call stack (when accessing memory beyond the call stack's upper and lower bounds), the stack is said to overflow or underflow, typically resulting in a program crash.

It may be beneficial to implement a measure supervising the stack and respectively generating a fault signal in case of a call stack overflow and call stack underflow.

5.5.1.1 Initial checks and configurations

Safety requirement under certain preconditions: [SM_061] When call stack underflow and call stack overflow due to systematic faults within the application software endangers the item (system) level functional safety integrity measure respective functional safety mechanisms may be implemented to detect call stack underflow and call stack overflow faults. [end]

Rationale: To have a notification in case of an call stack overflow or call stack underflow error.

Implementation hint: The DAC1 and DAC2 resources maybe used for incremental stack overflow or stack underflow detection when not being used as a hardware or software debug resource. Stack limit checking is available regardless of EDM or IDM mode, and when resources used for stack limit checking are owned by software, will utilize a DSI or machine check exception.

A data address compare (DAC) exception is signaled when there is a data access address match as defined by the debug control registers and data address compare events are enabled. This could either be a direct data address match or a selected set of data addresses, or a combination of data address and data value matching. The debug interrupt is taken when no higher priority exception is pending.

Software-owned stack limit checking does not require IDM to be set. Hardware owned stack limit checking requires EDM to be set. When stack limit checking is enabled, and DAC resources used for stack limit checking are owned by software, DAC events are not generated for resources configured to perform stack limit checking, and no DBSR DAC status flag will be set due to a detected stack limit violation.

Instead, depending on the processor mode, a data storage interrupt or a machine check exception is signaled. When stack limit checking is enabled, and DAC resources used for stack limit checking are owned by hardware, DAC events will be generated for resources configured to perform stack limit checking, and the EDBSR0 DAC status flag will be set due to a detected stack limit violation, causing entry into debug halted mode in the same way as a DAC exception normally does. The only difference is that qualification of the access address is performed as discussed in the next paragraph.

Incremental stack limit checking may be implemented using two data address watchpoints defined by DAC1 and DAC2. As hardware does not qualify a load or store access address with the use of GPR R1 as the base or index register used to compute an effective address when a load or store instruction is executed, special care has to be taken the watchpoints are not used elsewhere in the application software (guard band address range). This measure does only enable incremental stack overflow, as it only detects data addressing of the limit (upper and lower) address. Addressing going beyond the limits will be undetected. When DAC resources configured to perform incremental stack limit checking are not owned by hardware, if a stack limit violation occurs when performing the load or store, the access is aborted, and an error report machine check is generated, with MCSRR0 pointing to the address of the load or store access which

generated the stack overflow/underflow. If DAC resources configured to perform stack limit checking are owned by hardware, then a normal DAC event is generated (but qualified with use of GPR R1), and debug mode entry will occur in the same manner as for a non-stack limit DAC event.

When stack limit checking is enabled for a stack access, and DAC*n* resources are owned by hardware, the EDBSR0 DAC status flag will be set due to a detected stack limit violation, to cause entry into debug halted mode or to generate a watchpoint, or both, i.e. after the access has completed.

Independent limit checks for supervisor and user accesses may be implemented by allocating independent DACn resources to each, or a single limit may be applied using a single DACn resource. If more than one DACn resource is utilized, a DAC hit on any resource utilized for stack limit checking will cause the corresponding stack limit exception action to occur. If both a hardware-owned and a software-owned resource generate a stack limit exception for a given load or store, the software resource will have priority, since it is detected prior to completion of the access, and the access is aborted, thus the hardware event will not occur.

NOTE

For DAC1 and DAC2, access type (read, write) control is part of DBCR0.

5.5.2 MCU configuration

Safety requirement: [SM_064] It is required that application software checks correct initialization of the MPC567xK before activating the safety-relevant functionality. [end]

Safety requirement: [SM_062] It is required that application software checks the configuration of the SCCM once after boot. [end]

Recommendation: It is recommended that SSCM is configured to trigger an exception in case of any access to a peripheral slot not used on the device.

Rationale: To detect erroneous addressing and fault in address and bus logic.

Recommendation: It is recommended that after the boot application software perform an intended access to an unimplemented memory space and check for the expected abort to occur.

Rationale: To detect erroneous addressing and fault in address and bus logic.

Recommendation: It is recommended that unused interrupt vectors point, or jump, to an address that is illegal to execute, contains an illegal instruction, or in some other way causes detection of their execution.

Recommendation: It is recommended that only hardware related software (OS, drivers) run in supervisor mode.

Rationale: To reduce the risk accidental writes configuration registers affecting the execution of the MPC567xK’s safety function or disable the safety mechanism due to their change.

Recommendation: It is recommended that all configurations registers, and registers that aren't modified during application execution, are protected with a Hard Lock Protection. Configuration registers, and registers which have limited writes every trip time, are protected with soft-lock protection.

Rationale: To reduce the risk accidental writes configuration registers affecting the execution of the MPC567xK's safety function or disable the safety mechanism due to their change.

Implementation hint: Most of the off-platform peripherals have their own REG_PROT. Each peripheral that may be protected through the REG_PROT has a Set Soft Lock bit in the Register Protection space. This bit may be asserted to enable the protection of the related peripheral.

Each peripheral register that may be protected through register protection has a Set Soft Lock bit reserved in the Register Protection address space. This bit may be asserted to enable the protection of the related peripheral registers. Moreover, the Hard Lock Bit (REG_PROT_GCR[HLB] = 1) may be set for best write protection.

5.5.2.1 Detection of unwanted resets

Safety requirement: [SM_071] It is required to implement application level measure to detect missing RESETs of safety relevant modules. [end]

Implementation hint: To scan the content of the configuration registers of all safety relevant registers once after reset and before initialization to detect RESET failures.

Safety requirement: [SM_063] It is required to detect unwanted reset of safety relevant modules by application level software and hardware counter measures. [end]

This is a brief description of each of the column headings in Table 10:

- Reset signal – The internal signal that resets one or more of the receiving modules.
- Receiving module – This module is reset by the specified reset signal in the "Reset Signal" column.
- Software control – The register (or registers) which can reset the specified "Receiving Module" by the specified "Reset Signal". If no register is listed, the specified "Reset Signal" cannot be controlled by the register interface.
- Reset effect and detection – The effect of the unwanted reset of the specified "Receiving Module" and shows some mechanisms that can detect the module where the event occurred. If multiple mechanisms are listed, the software can choose the mechanism that better fits its need (unless explicitly specified).
- Specific software action needed – Additional software mechanism, with respect to the application software, required to detect an unwanted reset of the specified "Receiving Module" (for example, polling coming from configuration registers).

Table 10. Effects of reset

Internal Reset Signal	Receiving Module	Receiving Reset Input Port	Software Control	Detection	Software action required or automatic?	Description
ipg_hard_async_reset_addtl_b(0:0)	lbist_interface_dummy_sog	monitored_reset_b(3:3)	—	Reset complete system	No action required	Same reset that goes to fccu. Phase 1 reset and STCU combined reset
	fccu0_prot	ipg_hard_async_reset_b				
	fccu0	ipg_hard_async_rst_clk_b				
ipg_hard_async_reset_addtl_b(1:1)	fccu0	ipg_hard_async_rst_clk_safe_b	—	Safe fault, no detection required	No action required	
ipg_hard_async_reset_addtl_b(3:3)	npc_wrapper	ipp_ind_jcomp_sync				
ipg_hard_async_reset_b(0:0)	mc_rgm_ipi_int_synchro	rst_b	—	Reset complete system	No action required	Synchronized phase 3 reset from MC_RGM
	mc_me_nex_idle_b_synchro	rst_b				
	ebi	ipg_hard_reset_b				
	ddrc	ipg_hard_sync_reset_b				
	ctu1	ipg_hard_async_reset_b_ipg				
	ctu0	ipg_hard_async_reset_b_ipg				
	acp_komodo_dpm_splitter_ebi_ddr	resetb	—	Reset complete system	No action required	Synchronized phase 3 reset from MC_RGM. AHB Bus splitter for ebi
	wkpu_prot	ipg_hard_async_reset_b	—	Reset complete system	No action required	Synchronized phase 3 reset from MC_RGM
	wkpu	ipg_hard_async_reset_b				
	stop_sync_adc1	ipg_hard_async_reset_master_b				
	stop_sync_adc0	ipg_hard_async_reset_master_b				
	sram_rccu_4_1	ipg_hard_async_reset_b				
	sram_rccu_4_0	ipg_hard_async_reset_b				
	siul_prot	ipg_hard_async_reset_b				
	siul	ipg_hard_async_reset_b				
	s3_axbs_rccu_6_1	ipg_hard_async_reset_b				

Table 10. Effects of reset (continued)

Internal Reset Signal	Receiving Module	Receiving Reset Input Port	Software Control	Detection	Software action required or automatic?	Description
ipg_hard_async_reset_b(0:0) (cont'd)	s3_axbs_rccu_6_0	ipg_hard_async_reset_b	—	Reset complete system	No action required	Synchronized phase 3 reset from MC_RGM
	rcosc16M_dig					
	pit_prot					
	pit					
	pdi_prot					
	pdi					
	osc_dig					
	npc_hndshk					
	mc_pcu_prot					
	mc_pcu					
	mc_me_prot					
	mc_me					
	m2_axbs_rccu_7_1					
	m2_axbs_rccu_7_0					
	linflex3_prot					
	linflex2_prot					
	linflex1_prot					
	linflex0_prot					
	lin3					
	lin2					
	lin1					
	lin0					
	leo3flash_rccu_5_1					

Table 10. Effects of reset (continued)

Internal Reset Signal	Receiving Module	Receiving Reset Input Port	Software Control	Detection	Software action required or automatic?	Description
ipg_hard_async_reset_b(0:0) (cont'd)	leo3flash_rccu_5_0	ipg_hard_async_reset_b	—	Reset complete system	No action required	Synchronized phase 3 reset from MC_RGM
	i2c2_prot	ipg_hard_async_reset_b				
	i2c2	ipg_hard_async_reset_b				
	i2c1_prot	ipg_hard_async_reset_b				
	i2c1	ipg_hard_async_reset_b				
	i2c0_prot	ipg_hard_async_reset_b				
	i2c0	ipg_hard_async_reset_b				
	flexray_prot	ipg_hard_async_reset_b				
	flexray	ipg_hard_async_reset_b				
	flexpwm2_prot	ipg_hard_async_reset_b				
	flexpwm2	ipg_master_hard_async_reset_b				
	flexpwm1_prot	ipg_hard_async_reset_b				
	flexpwm1	ipg_master_hard_async_reset_b				
	flexpwm0_prot	ipg_hard_async_reset_b				
	flexpwm0	ipg_master_hard_async_reset_b				
	flex_mux0	ipg_hard_async_reset_b				
	fec_prot	ipg_hard_async_reset_b				
	etimer2_prot	ipg_hard_async_reset_b				
	etimer2	ipg_master_hard_async_reset_b				
	etimer1_prot	ipg_hard_async_reset_b				
	etimer1	ipg_master_hard_async_reset_b				
	etimer0_prot	ipg_hard_async_reset_b				
	etimer0	ipg_master_hard_async_reset_b				

Table 10. Effects of reset (continued)

Internal Reset Signal	Receiving Module	Receiving Reset Input Port	Software Control	Detection	Software action required or automatic?	Description
ipg_hard_async_reset_b(0:0) (cont'd)	ebi_sync_clk	ipg_hard_async_reset_b	—	Reset complete system	No action required	Synchronized phase 3 reset from MC_RGM
	ebi_prot					
	dspi2_prot					
	dspi2					
	dspi1_prot					
	dspi1					
	dspi0_prot					
	dspi0					
	dma_rccu_1_1					
	dma_rccu_1_0					
	dma1_ch_mux_prot					
	dma1_ch_mux					
	dma0_ch_mux_prot					
	dma0_ch_mux					
	ddrc_prot					
	ddrc					
	ctu1_prot					
	ctu0_prot					
	crc_top_prot					
	crc_top					
	crc1_top_prot					
	crc1_top					
	core_rccu_0_1					

Table 10. Effects of reset (continued)

Internal Reset Signal	Receiving Module	Receiving Reset Input Port	Software Control	Detection	Software action required or automatic?	Description
ipg_hard_async_reset_b(0:0) (cont'd)	core_rccu_0_0	ipg_hard_async_reset_b	—	Reset complete system	No action required	Synchronized phase 3 reset from MC_RGM
	cmu2					
	cmu1					
	cmu0					
	can3_prot					
	can3					
	can2_prot					
	can2					
	can1_prot					
	can1					
	can0_prot					
	can0					
	bam					
	axbs_rccu_3_1					
	axbs_rccu_3_0					
	aips_rccu_2_1					
	aips_rccu_2_0					
	adc3_prot					
	adc2_prot					
	adc1_prot					
	adc1_ipsync	ipg_master_reset_b				
	adc1_ipsync	ipg_dma_hard_async_b				
	adc0_prot	ipg_hard_async_reset_b				

Table 10. Effects of reset (continued)

Internal Reset Signal	Receiving Module	Receiving Reset Input Port	Software Control	Detection	Software action required or automatic?	Description
ipg_hard_async_reset_b(0:0)	adc0_ipsync	ipg_master_reset_b	—	Reset complete system	No action required	Synchronized phase 3 reset from MC_RGM
	adc0_ipsync	ipg_dma_hard_async_b				
	acp_komodo_c	ipg_hard_async_reset_b				
	IPS_READ_MUX_1_3					
	IPS_READ_MUX_1_2					
	IPS_READ_MUX_1_1					
	IPS_READ_MUX_0_3					
	IPS_READ_MUX_0_2					
	IPS_READ_MUX_0_1					
ipg_hard_async_reset_b(1:1)	flexpwm2	ipg_slave_hard_async_reset_b				
	flexpwm1					
	flexpwm0					
	etimer2					
	etimer1					
	etimer0					
	ctu1	ipg_hard_async_reset_b_mt				
	ctu0					
	stop_sync_adc1	ipg_hard_async_reset_slave_b				
	stop_sync_adc0					
	adc3_flop	ipg_hard_async_reset_b				
	adc3					
	adc2_flop					
	adc2					

Table 10. Effects of reset (continued)

Internal Reset Signal	Receiving Module	Receiving Reset Input Port	Software Control	Detection	Software action required or automatic?	Description
ipg_hard_async_reset_b(1:1)	adc1_ipsync	ipg_slave_reset_b	—	Reset complete system	No action required	Synchronized phase 3 reset from MC_RGM
	adc1_ipsync	ipg_per_hard_async_b				
	adc1_flop	ipg_hard_async_reset_b				
	adc1					
	adc0_ipsync	ipg_slave_reset_b				
	adc0_ipsync	ipg_per_hard_async_b				
	adc0_flop	ipg_hard_async_reset_b				
	adc0					
ipg_hard_async_reset_b(4:4)	siul	ipg_hard_async_reset_osc_b				
	lbist_interface_dummy_sog	monitored_reset_b(2:2)				
ipg_hard_async_reset_cflash_b(0:0)	cflash0_prot	ipg_hard_async_reset_b	—	Reset complete system	No action required	Synchronized phase 1 reset from MC_RGM
ipg_hard_async_reset_cflash_b(0:0)	cflash0	ipg_hard_async_reset_b				
ipg_hard_async_reset_cflash_b(1:1)	lbist_interface_dummy_sog	monitored_reset_b(14:14)				
	cflash1_prot	ipg_hard_async_reset_b				
	cflash1					
ipg_hard_async_reset_dest_b	pd_glitch_fix	nreset	—	Reset complete system	No action required	Destructive reset (design glue logic to avoid glitch on PLL clock in event of reset)
	npc_wrapper	poreset_b	—	Reset complete system	No action required	Destructive reset
	mc_pcu	ipg_hard_async_reset_dest_b				
	gluelogic_sog	ipg_hard_async_reset_dest_b				
	dflash	ipg_pad_reset_b				
	cflash1	ipg_pad_reset_b				
	cflash0	ipg_pad_reset_b				

Table 10. Effects of reset (continued)

Internal Reset Signal	Receiving Module	Receiving Reset Input Port	Software Control	Detection	Software action required or automatic?	Description
ipg_hard_async_reset_dflash_b	dflash_prot	ipg_hard_async_reset_b	—	Reset complete system	No action required	Synchronized phase 1 reset from MC_RGM
	dflash					
ipg_hard_async_reset_pll_b	fmpll1	ipg_hard_async_rst_b	—	Wrong PLL frequency	No action required	Synchronized phase 3 reset from MC_RGM (except when PLL is used as source of mbist)
	fmpll0					
jtag_active_lbist_out	npc_wrapper	ipp_ind_jcomp	—	Reset complete system	No action required	—
—	rcosc16M_dig	power_on_rst_b				
	fmpll1	por_rst_b				
	fmpll0	por_rst_b				
—	macc	trstn	—	Safe	No action required	STCU - mbist reset
	flexray_rom	trstn				
	flexray_rom	rst_n				
	flexray_lut_ram	trstn				
	flexray_lut_ram	rst_n				
	flexray_data_ram	trstn				
	flexray_data_ram	rst_n				
	fec_rif_ram	trstn				
	fec_rif_ram	rst_n				
	fec_mib_ram	trstn				
	fec_mib_ram	rst_n				
	dma_ram	trstn				
	dma_ram	rst_n				
	can_rxim_1_2_3	trstn				
	can_rxim_1_2_3	rst_n				

Table 10. Effects of reset (continued)

Internal Reset Signal	Receiving Module	Receiving Reset Input Port	Software Control	Detection	Software action required or automatic?	Description
—	can_rxim_0	trstn	—	Safe	No action required	STCU - mbist reset
	can_rxim_0	rst_n				
	can_mb_1_2_3	trstn				
	can_mb_1_2_3	rst_n				
	can_mb_0	trstn				
	can_mb_0	rst_n				
	bam_rom	trstn				
	bam_rom	rst_n				

6 Failure rates and FMEDA

6.1 Mission profile

Table 11 shows the parameters of Mission profile 1 and profile 2 for typical applications. This document is based on these mission profiles although usage of MPC567xK is not limited to these values. Mission profile 1 is a typical automotive profile and Profile 2 is a an alternative profile with continuous operation.

Table 11. Mission profiles

Mission Parameters	Mission profile 1	Mission profile 2
Trip time (T_{trip}):	10 hours	continuous
FTTI:	10 ms	10 ms
Lifetime (T_{life})	20 years	5–10 years
Total operating hours:	12000 hours	50000 hours

Table 12 shows temperature profiles of the different package options for Mission profile 1.

Table 12. Temperature profile for Mission profile 1

Device type	Temperature range (°C)	Operation time (h)
Packaged device	125 – 135	120
	110 – 120	960
	90 – 100	7680
	30 – 40	3240
Bare die	120 – 125	120
	100 – 110	960
	80 – 90	7680
	20 – 30	3240

Table 13 shows temperature profiles of the different package options for profile 2.

Table 13. Temperature profile for Mission profile 2

Device type	Temperature range (°C)	Operation time (h)
Packaged device	10 – 60	21500 (43%)
	-40 – 10	28500 (57%)
Bare die	10 – 60	21500 (43%)
	-40 – 10	28500 (57%)

6.2 Overview

According to ISO 26262-4, chapter 7.4.3.1 and IEC 61508, Table B.6 a functional safety/failure analysis on hardware design shall be applied to identify the causes of failures and the effects of faults. A typical inductive analysis method is FMEDA (Failure Modes Effects and Diagnostic Analysis).

Dedicated FMEDA and failure rate tables for ISO 26262 and IEC 61508 were created for each of the following parts of MPC567xK:

- FMEDAs for basic elements:
  - Core: processing units (CPU)
  - SRAM: non-volatile memories (SRAMs)
  - Flash memory: volatile memory
  - Clock: clock generation and clock supervision
  - Power: Power generation and distribution
- Failure rates of application dependent functions:
  - I/O and peripherals

It is assumed, that the basic elements are used in every application and having low application dependency, whereas the use of peripheral and communications functions have a high application dependency. The functional safety architecture of basic elements may not interfere with application.

The application dependent functions need to be included into the functional safety concept on system level. Thus only raw failure rates and no failure metrics are given for these elements.

Table 14 lists all modules of MPC567xK and their mapping to the five FMEDAs.

Table 14. Module distribution over FMEDAs

Module	FMEDA for				
	Processing Unit	SRAM	Flash memory	Clock	Power
CPU	✓				
eDMA	✓				
INTC	✓				
MPU and MMU	✓				
FCCU and external FAULT signals	✓				
SWT, PIT, STM	✓				
SRAMC		✓			
SRAM (volatile memory)		✓			
Flash memory (non-volatile memory), including power supply pin			✓		
PFLASHC			✓		

Table 14. Module distribution over FMEDAs

Module	FMEDA for				
	Processing Unit	SRAM	Flash memory	Clock	Power
CMU				✓	
External crystal				✓	
FMPLL				✓	
IRCOSC				✓	
XOSC				✓	
XBAR	✓	✓	✓		
PMC (including power regulator pins)					✓
External voltage regulator					✓
Nexus, JTAG, BAM	✓				

The following are modules that are covered by the failure rates list:

- PBRIDGE_*n*
- BAM
- WAKEUP
- ADC_*n*
- CTU
- eTimer_*n*
- FlexPWM_*n*
- SIUL
- GPIO
- FlexRay
- FlexCAN_*n*
- DSPI_*n*
- LINFlexD_*n*

The FMEDA enables selection of functional safety mechanisms planed to be implemented in a specific application. Enabling or disabling the usage of functional safety mechanisms within an application is possible within the sheets.

The only failure modes used for the FMEDA are taken from table D.1 of ISO 26262-5, annex D. These are used for both ISO 26262 and IEC 61508 calculations.

The information in this section is valid as of the latest revision date of this document. Please ask your Freescale Semiconductor representative for updates when performing the system level functional safety analysis.

Furthermore, the complete FMEDA is available upon request when covered by a Freescale Semiconductor NDA (please contact your Freescale Semiconductor representative).

Significant key values of the FMEDA are presented in a FMEDA report document, for an example case, in which all typical functional safety mechanisms presented in this document are enabled.

The failure rate data used in these FMEDAs have been derived using failure data collected from Freescale components already in the market, and from accelerated High Temperature Operating Life tests (HTOL) performed on samples of MPC567xK or specific measurement, for example, using neutrons for single event failure rates.

The implementation hints documented are assumed to be implemented as functional safety integrity measures.

7 Provisions against dependent failures

ISO 26262 distinguishes between cascading failures and CMF. A cascading failure is a “failure of an element of an item causing another element or elements of the same item to fail” whereas a common cause failure is “a failure of two or more elements of an item resulting from a single specific event or root cause.”

7.1 Causes of dependent failures

ISO 26262-9 lists the following dependent failures, which are applicable to the MPC567xK on chip level:

- Random hardware failures, for example:
  - physical defects that are able to influence an element and its redundant element (transient faults are not considered initiator of common mode failures).
  - electrical dependencies:
    - latch-up
    - supply noise
    - faults of checking circuits (for example, RC)
    - shared logic
    - logic physically overlapping
    - signals crossing lakes
    - timing faults
- Environmental factors, for example:
  - temperature
  - EMI
- Failures of common signals (external resources), for example:
  - clock
  - power-supply

  - non-application control signals (for example, testing, debugging)
  - signals from non-replicated modules outside SoR.

Additionally, the following topics are mentioned in ISO 26262-9, which are beyond the scope of this document and may be considered in other documents (see documents referenced in Section 2.1, Assumed conditions of operation):

- Development faults:
  - development faults are systematic faults which are addressed by design-process
- Manufacturing faults:
  - manufacturing faults are usually systematic faults addressed by design-process and production test
- Installation and repair faults:
  - installation and repair faults need to be considered at system level
- Stress due to specific situations:
  - Specific situations may be considered at system level. Additionally, the result of stress (for example, wear and ageing due to electro-migration) usually lead to single-point faults and are not considered dependent failures.

7.2 Measures against dependent failures

7.2.1 Physical isolation

To maximize the independence of redundant components, lakes are formed. This results in generation of a partial, but nevertheless substantial, physical diversity in the silicon structure.

The duplicated computational elements in the SoR are separated in different lakes, lake 0 and lake 1. Peripherals for I/O communication are grouped into a single lake. System level countermeasure for common mode failures may therefore be required.

The redundant modules share a common silicon substrate. A failure of the substrate is typically fatal and has to be detected by external system level measures. It is assumed that an external timeout function (watchdog) is continuously monitoring the MPC567xK and is capable of detecting this CMF, and will switch the system to a Safe $state_{system}$ within the FTTI.

The MPC567xK device satisfies the standard AECQ100 for latch-up immunity.

7.2.2 Environmental conditions

7.2.2.1 Temperature

MPC567xK was designed to work within a maximum operational temperature profile (see the *Qorivva MPC5675K Microcontroller Data Sheet* for details). To cover common mode failures cause by temperature, a temperature sensor for supervision is implemented which is described in Section 5.2.4, Temperature Sensor (TSENS).

7.2.2.2 EMI and I/O

To cope with noise at digital inputs, the I/O circuitry provides input hysteresis on all digital inputs. Moreover, the RESET and NMI inputs contain glitch filtering capabilities, which are described in Section 5.2.27, Glitch filter.

To reduce interference due to digital outputs, the I/O circuitry provides signal slope control. An internal weak pull up or pull down structure is also provided to define the input state.

7.2.3 Failures of common signals

7.2.3.1 Clock

To cover common mode failures caused by erroneous clocks, supervisory modules are implemented as described in Section 5.2.10, Clock Monitor Unit (CMU). Major failures in the clock system are also detected by the use of the SWT (Section 5.2.5, Software Watchdog Timer (SWT)).

7.2.3.2 Power supply

To cover common mode failures caused by voltage, supervisory modules are implemented as described in Section 5.2.12, Power Management Controller (PMC).

Some common mode failures (for example, loss of power supply) will still be detected by the use of an external watchdog (Section 4.1.2, External Watchdog (EXWD)) because application software is no longer able to trigger the EXWD.

7.2.3.3 Non-application control signals

Modules and signals (for example, for scan, test and debug), which are not functional safety relevant and thus have no functional safety mechanism included should never be able to violate the functional safety goal. This can be achieved by either not interfering with the functional safety relevant parts of the MPC567xK or by detecting such interference. For example, there must be assurance that the system is not debugged (or unintentionally in debug mode), or in any other special mode different from normal application execution mode like test mode. FCCU failure indication is generated when one of the following conditions is fulfilled (please also refer to Table 1):

- The device leaves LSM or DPM.
- A self-test sequence of the STCU is unintentionally executed during normal operation of the device.
- Any of the configurations for production test are unintentionally executed during normal operation of the device.
- Any JTAGC instruction is executed that causes a system reset or Test Mode Select (TMS) signal is used to sequence the TAP controller state machine.

7.3 CMF avoidance on system level

It is recommended to not use adjacent input and output signals of peripherals, which are used redundantly, in order to reduce CMF. As internal pad position and external pin/ball position do not necessarily correspond to each other, the system integrator may take the following recommendations into consideration:

- Usage of non-contiguous balls of the package
- Usage of non-contiguous pads of the silicon
- Usage of peripheral modules sharing same PBRIDGE
- Non-contiguous routing of these signals on the PCB

Safety requirement under certain preconditions: [SM_060] If the system requires robustness regarding common mode faults, measures on item (system) level have to improve the robustness of redundant inputs for double read input functions in respect to common mode faults. [end]

Recommendation: Avoid physically adjacent inputs for double read input functions to avoid CMFs.

Rationale: To minimize common mode failures (CMF).

Implementation hint: Pad position as well as pin/ball position should be taken into consideration.

The pin/ball assignment for individual peripherals can be extracted from the *Qorivva MPC5675K Microcontroller Data Sheet*. However, this information is listed briefly here.

7.3.1 I/O pin/ball configuration

Safety requirement: [SM_072] The user must avoid configurations that place redundant signals on neighboring pads or pins. [end]

Whether two functions are adjacent to each other can easily be determined by looking at the mechanical drawings of the packages (see the *Qorivva MPC5675K Microcontroller Data Sheet*) together with the ball number information of the packages as seen in the *Qorivva MPC5675K Microcontroller Reference Manual* “System Integration Unit Lite (SIUL)” section and the “Pin muxing” table (see also Table 15).

An example on the BGA473 package as shown in Figure 29 has two balls belonging to port pins flexpwm0_X[1] and flexpwm0_B[0], which are balls N3 and P1, respectively. They are not directly adjacent to each other on the BGA package, but if you look at Table 15 you will notice that they are adjacent on the die, pads 61 and 60, respectively.

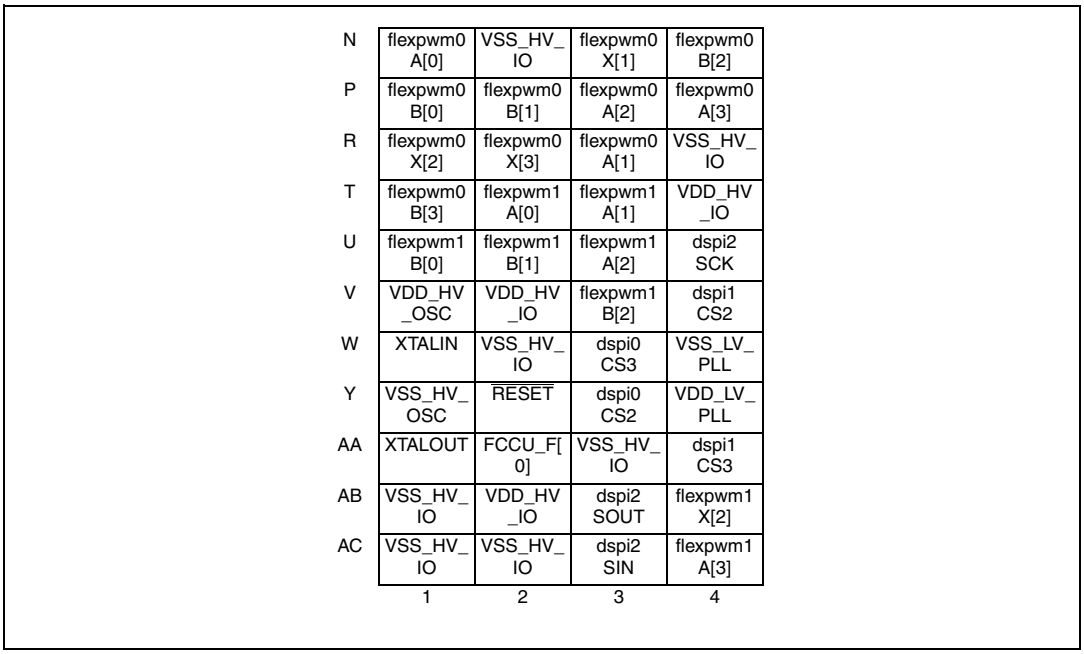


Figure 29. BGA473 adjacency

In another example looking at balls P3 and P4 in Figure 29, flexpwm0_A[2] and flexpwm0_A[3], respectively, you will notice that the balls are adjacent, but if you reference Table 15 you will also notice that the pads are not adjacent (66 and 73, respectively). Therefore, the two corresponding die pads are not adjacent to each other.

The above examples are valid for corresponding balls on the BGA473. For a thorough analysis of pin adjacency related to all signals see Table 15. This table can be used to determine whether two pins are adjacent in the internal die for all signals and packages. Two pins, identified by the columns ‘Ball Name’, are adjacent on the internal die if the numbers in the ‘Physical Pad Sequence’ column are consecutive (for example, pad number *n* and pad number *n* + 1 are adjacent).

Table 15. Physical pin displacement on internal die

Ball Name BGA 473	Ball Name BGA 257 (if different)	Ball Number BGA 473	Ball Number BGA 257	Physical Pad Sequence[1]
ADC0_ADC1_ANA11	—	Y14	U11	280
ADC0_ADC1_ANA12	—	AA14	T11	283
ADC0_ADC1_ANA13	—	AB14	R11	284
ADC0_ADC1_ANA14	—	AC14	P11	287
ADC0_ANA0	—	AB10	P7	258
ADC0_ANA1	—	AC10	T10	259
ADC0_ANA2	—	AA11	R10	260
ADC0_ANA3	—	AC11	—	261
ADC0_ANA4	—	AB11	—	262
ADC0_ANA5	—	AA12	—	263

Table 15. Physical pin displacement on internal die (continued)

Ball Name BGA 473	Ball Name BGA 257 (if different)	Ball Number BGA 473	Ball Number BGA 257	Physical Pad Sequence[1]
ADC0_ANA6	—	AB12	—	264
ADC0_ANA7	—	AB13	—	265
ADC0_ANA8	—	AA13	—	266
ADC1_ANA0	—	AA15	T12	307
ADC1_ANA1	—	AB15	R12	308
ADC1_ANA2	—	AA16	T13	309
ADC1_ANA3	—	AB16	—	310
ADC1_ANA4	—	AB17	—	311
ADC1_ANA5	—	AA17	—	312
ADC1_ANA6	—	Y18	—	314
ADC1_ANA7	—	AA18	—	313
ADC1_ANA8	—	Y17	—	315
ADC2_ADC3_ANA11	—	Y7	U7	216
ADC2_ADC3_ANA12	—	AA7	U8	219
ADC2_ADC3_ANA13	—	AB7	T8	220
ADC2_ADC3_ANA14	—	Y8	R8	223
ADC2_ANA0	—	AA8	R5	239
ADC2_ANA1	—	AB8	U6	240
ADC2_ANA2	—	AB9	T6	241
ADC2_ANA3	—	AC9	R6	242
ADC3_ANA0	—	Y6	T4	201
ADC3_ANA1	—	AA6	U4	204
ADC3_ANA2	—	AB6	U5	206
ADC3_ANA3	—	AC6	T5	207
CAN0_RXD	—	C20	C14	550
CAN0_TXD	—	B21	B15	549
CAN1_RXD	—	D3	D3	713
CAN1_TXD	—	B4	B4	712
CLKOUT0	—	E20	F14	540
CLKOUT1	—	B3	B3	715
DRAMC_ADDR0	—	U23	—	450
DRAMC_ADDR1	—	T22	—	449
DRAMC_ADDR10	—	AA23	—	436
DRAMC_ADDR11	—	Y22	—	435

Table 15. Physical pin displacement on internal die (continued)

Ball Name BGA 473	Ball Name BGA 257 (if different)	Ball Number BGA 473	Ball Number BGA 257	Physical Pad Sequence[1]
DRAMC_ADDR12	—	U21	—	433
DRAMC_ADDR13	—	V21	—	432
DRAMC_ADDR14	—	W21	—	430
DRAMC_ADDR15	—	Y21	—	429
DRAMC_ADDR2	—	V23	—	448
DRAMC_ADDR3	—	R21	—	447
DRAMC_ADDR4	—	W23	—	445
DRAMC_ADDR5	FLEXPWM1_B1	Y23	P17	444
DRAMC_ADDR6	FLEXPWM1_A2	U20	R16	443
DRAMC_ADDR7	FLEXPWM1_B2	W22	R17	441
DRAMC_ADDR8	—	T20	—	439
DRAMC_ADDR9	—	T21	—	437
DRAMC_BA0	FLEXPWM0_B3	D23	N14	497
DRAMC_BA1	FLEXPWM1_A0	D21	N16	495
DRAMC_BA2	FLEXPWM1_B0	E23	N17	494
DRAMC_CAS	FLEXPWM0_B2	C23	M14	499
DRAMC_CKE	FLEXPWM0_A0	R22	P16	451
DRAMC_CS0	FLEXPWM0_B1	E22	L16	501
DRAMC_D0	—	J20	—	492
DRAMC_D1	—	J21	—	491
DRAMC_D10	—	M23	—	469
DRAMC_D11	—	M22	—	468
DRAMC_D12	—	N23	—	463
DRAMC_D13	—	N22	—	462
DRAMC_D14	—	P20	—	460
DRAMC_D15	—	P21	—	459
DRAMC_D2	—	H20	—	490
DRAMC_D3	—	J22	—	489
DRAMC_D4	—	K21	—	481
DRAMC_D5	—	F23	—	480
DRAMC_D6	—	J23	—	479
DRAMC_D7	—	G23	—	478
DRAMC_D8	—	K22	—	471
DRAMC_D9	—	K23	—	470

Table 15. Physical pin displacement on internal die (continued)

Ball Name BGA 473	Ball Name BGA 257 (if different)	Ball Number BGA 473	Ball Number BGA 257	Physical Pad Sequence[1]
DRAMC_DM0	—	G22	—	486
DRAMC_DM1	—	N21	—	465
DRAMC_DQS0	—	G21	—	488
DRAMC_DQS1	—	N20	—	466
DRAMC_ODT	FLEXPWM1_A1	M20	M17	477
DRAMC_RAS	FLEXPWM0_A2	F20	N15	498
DRAMC_WEB	FLEXPWM0_A3	M21	P15	472
DSPI0_CS0	—	L1	H3	47
DSPI0_CS2	—	Y3	P3	92
DSPI0_CS3	—	W3	N3	91
DSPI0_SCK	—	K1	G3	45
DSPI0_SIN	—	M3	K4	52
DSPI0_SOUT	—	D4	D4	2
DSPI1_CS0	—	K2	H4	44
DSPI1_CS2	—	V4	M3	90
DSPI1_CS3	—	AA4	R4	98
DSPI1_SCK	—	K3	G4	42
DSPI1_SIN	—	J4	F4	36
DSPI1_SOUT	—	K4	F3	40
DSPI2_CS0	—	L3	J3	49
DSPI2_CS1	—	B6	B6	695
DSPI2_CS2	—	L2	J4	48
DSPI2_SCK	—	U4	L3	88
DSPI2_SIN	—	AC3	U3	100
DSPI2_SOUT	—	AB3	T3	99
EBI_ADDR28	FLEXPWM0_X0	F21	H17	509
EBI_ADDR29	FLEXPWM0_X1	D22	J17	508
EBI_ADDR30	FLEXPWM0_X2	G20	K14	507
EBI_ADDR31	FLEXPWM0_X3	C22	K15	505
EBI_CLKOUT	FLEXPWM0_B0	F22	K17	503
EBI_RD_WR	FLEXPWM0_A1	E21	K16	504
ETH_COL	—	C15	C13	610
ETH_CRS	—	C12	C11	629
ETH_MDC	—	D15	D13	615

Table 15. Physical pin displacement on internal die (continued)

Ball Name BGA 473	Ball Name BGA 257 (if different)	Ball Number BGA 473	Ball Number BGA 257	Physical Pad Sequence[1]
ETH_MDIO	—	A10	A13	607
ETH_RXCLK	—	C13	A11	623
ETH_RXD0	—	B12	A12	616
ETH_RXD1	—	C14	B12	618
ETH_RXD2	—	D14	A10	634
ETH_RXD3	—	B9	B10	635
ETH_RXDV	—	A9	D12	620
ETH_RXER	—	B10	B11	627
ETH_TXCLK	—	A11	B14	606
ETH_TXD0	—	B11	C12	619
ETH_TXD1	—	C11	D11	632
ETH_TXD2	—	C10	D10	636
ETH_TXD3	—	A13	A15	600
ETH_TXEN	—	A12	A14	603
ETH_TXER	—	B13	B13	609
ETIMER0_ETC0	—	D7	C6	673
ETIMER0_ETC1	—	C7	C7	669
ETIMER0_ETC2	—	C8	C8	663
ETIMER0_ETC3	—	C9	C9	661
ETIMER0_ETC4	—	C6	D7	671
ETIMER0_ETC5	—	D6	D6	672
ETIMER1_ETC0	—	AA20	T15	327
ETIMER1_ETC1	—	Y9	P5	251
ETIMER1_ETC2	—	Y10	P6	253
ETIMER1_ETC3	—	Y11	P8	254
ETIMER1_ETC4	—	Y16	P12	405
ETIMER1_ETC5	—	Y15	P13	407
EVTI_B	—	H4	L2	27
EVTO_B	—	H1	L1	30
FCCU_F[0]	—	AA2	R4	97
FCCU_F[1]	—	C4	C4	709
FLEXPWM0_A0	—	N1	—	59
FLEXPWM0_A1	—	R3	—	75
FLEXPWM0_A2	—	P3	—	66

Table 15. Physical pin displacement on internal die (continued)

Ball Name BGA 473	Ball Name BGA 257 (if different)	Ball Number BGA 473	Ball Number BGA 257	Physical Pad Sequence[1]
FLEXPWM0_A3	—	P4	—	73
FLEXPWM0_B0	—	P1	—	60
FLEXPWM0_B1	—	P2	—	62
FLEXPWM0_B2	—	N4	—	65
FLEXPWM0_B3	—	T1	—	74
FLEXPWM0_X0	—	M1	—	57
FLEXPWM0_X1	—	N3	—	61
FLEXPWM0_X2	—	R1	—	64
FLEXPWM0_X3	—	R2	—	72
FLEXPWM1_A0	—	T2	—	77
FLEXPWM1_A1	—	T3	—	79
FLEXPWM1_A2	—	U3	—	84
FLEXPWM1_A3	—	AC4	—	108
FLEXPWM1_B0	—	U1	—	78
FLEXPWM1_B1	—	U2	—	83
FLEXPWM1_B2	—	V3	—	87
FLEXPWM1_B3	—	AC5	—	113
FLEXPWM1_X0	—	Y5	—	103
FLEXPWM1_X1	—	AA5	—	104
FLEXPWM1_X2	—	AB4	—	105
FLEXPWM1_X3	—	AB5	—	107
FLEXRAY_CA_RX	—	E3	E3	708
FLEXRAY_CA_TR_EN	—	A8	A8	676
FLEXRAY_CA_TX	—	B8	B8	675
FLEXRAY_CB_RX	—	C5	C5	702
FLEXRAY_CB_TR_EN	—	B7	B7	678
FLEXRAY_CB_TX	—	A7	A7	677
JTAG_TDI	—	AA19	M15	417
JTAG_TDO	—	AB18	L17	413
LIN0_RXD	—	W20	T14	411
LIN0_TXD	—	V20	R14	409
LIN1_RXD	—	AB21	—	410
LIN1_TXD	—	AA22	—	408
MCKO	—	G1	J1	18

Table 15. Physical pin displacement on internal die (continued)

Ball Name BGA 473	Ball Name BGA 257 (if different)	Ball Number BGA 473	Ball Number BGA 257	Physical Pad Sequence[1]
MDO1	—	D1	E2	0
MDO10	—	F1	H1	15
MDO11	—	F2	F2	11
MDO12	—	J3	M4	37
MDO13	—	J2	L4	35
MDO14	—	B5	B5	701
MDO15	—	C2	C2	716
MDO2	—	E2	D1	5
MDO3	—	D2	D2	725
MDO4	—	F4	G1	14
MDO5	—	A4	A4	706
MDO6	—	F3	F1	12
MDO7	—	A5	A5	705
MDO8	—	G3	J2	20
MDO9	—	A6	A6	700
MSEO_B0	—	H3	K1	29
MSEO_B1	—	G4	K2	21
PDI_CLOCK	—	A17	C17	535
PDI_DATA0	—	B17	D16	534
PDI_DATA1	—	A16	D17	533
PDI_DATA10	—	A19	G15	521
PDI_DATA11	—	D18	G16	520
PDI_DATA12	—	C19	H14	519
PDI_DATA13	—	A20	H15	517
PDI_DATA14	—	B20	J14	515
PDI_DATA15	—	A21	J15	514
PDI_DATA2	—	C17	E15	532
PDI_DATA3	—	A15	E16	530
PDI_DATA4	—	B16	E17	529
PDI_DATA5	—	C16	C16	528
PDI_DATA6	—	B15	F15	526
PDI_DATA7	—	A18	F16	525
PDI_DATA8	—	C18	F17	523

Table 15. Physical pin displacement on internal die (continued)

Ball Name BGA 473	Ball Name BGA 257 (if different)	Ball Number BGA 473	Ball Number BGA 257	Physical Pad Sequence[1]
PDI_DATA9	—	B19	G14	522
PDI_FRAME_V	—	D19	G17	538
PDI_LINE_V	—	B18	E14	536
READY	—	J1	K3	33

NOTES:
1 Die pads not relevant for analysis, and non-functional pins (for example, power) are not shown.

7.3.2 Modules sharing PBRIDGE

The system designer needs to take into consideration how modules are distributed across the different PBRIDGEs. Whenever possible the redundant modules should be used such that each module is connected to a different PBRIDGE. For example, FlexPWM_0 and FlexPWM_1 on PBRIDGE_0 while FlexPWM_2 connects to PBRIDGE_1. So, when FlexPWM redundancy is required for the safety function, the designer should utilize PBRIDGE_2 with either PBRIDGE_1 or PBRIDGE_0.

7.3.3 External timeout function

A common mode failure may lead to a state where the MPC567xK is not able to signal an internal failure via its FCCU_F[*n*] signals (error out). With the use of a system level timeout function (for example, watchdog timer), the likelihood that CMFs affect the functional safety of the system can be reduced significantly.

In general, the external watchdog covers common mode failures which are related to:

- Missing/wrong power
- Missing/wrong clocks
- Missing/wrong resets
- General destruction of internal components (for example, latch-up at redundant input pads)
- Errors in mode change (for example, test, debug, sleep/wakeup)

Since these errors do not result in subtle output variations of the MCU but typically in a complete failure, a simple watchdog is sufficient.

The external watchdog function is in permanent communication with the CPU of MPC567xK. As soon as there are no correct communications, the external watchdog function switches the system to Safe $state_{system}$. Thus, either the MPC567xK or external watchdog function can transition the system to Safe $state_{system}$. The external watchdog function is required to be sufficiently independent of the MPC567xK (for example, regarding clock generation, power supply, and so on).

The external watchdog function does not necessarily need to be a dedicated IC, the requirements may also be fulfilled by another MCU (already used in the system) which is capable of detecting a communication problem and moving the system to Safe $state_{system}$.

8 Additional information

8.1 Safety function pseudo-code

CAUTION

For some functions, code examples are given to exemplify the intended functionality and give some hints on integration. This code does not cover all aspects of actual applications and, as such, is intended to be used as a general guideline.

In some code examples given in this chapter, ‘while’ loops are used. These loops may never terminate, depending on their condition. To prevent termination of software execution flow, a timeout function may be implemented (see Figure 30). For simplicity, such a timeout is not used in the code examples for the safety integrity functions, but may be used in system level application software.

```
start_time = read_timer();
do current_time = read_timer() - start_time;
while ((Current_time < Limit) && (Flag_not_set))
if (Flag_set)
        PASS;
else
        FAIL;
```

Figure 30. Code example: timeout

8.1.1 Flash memory

8.1.1.1 FLASH_SW_ECCTEST

```
definitions for FLASH_SW_ECCTEST

#define C90FL_BASE 0xC3F88000
#define C90FL_MCR (*(vuint32_t *)(C90FL_BASE))
#define C90FL_ADR (*(vuint32_t *)(C90FL_BASE+0x18))
#define C90FL_UT0 (*(vuint32_t *)(C90FL_BASE+0x3C))
#define C90FL_UT1 (*(vuint32_t *)(C90FL_BASE+0x40))
#define C90FL_UT2 (*(vuint32_t *)(C90FL_BASE+0x44))

#define C90FL_UT_PASSWORD 0xF9F99999

#define UT0_UTE (0x80000000)
#define UT0_SBCE (0x40000000)
#define UT0_DSI(x) (((x)&0xFF)<<16)                    // Bit definitions and macros for
#define UT0_EIE (0x00000008)                           // C90FL_UT0
#define UT0_AIS (0x00000004)
#define UT0_AIE (0x00000002)
#define UT0_AID (0x00000001)                           // Data syndrome input
                                                       // ECC data input enable
#define MCR_SBC (0x00002000)                           // Array integrity sequence
                                                       // Array integrity enable
#define C90FL_OK 0x0                                   // Array integrity done
#define C90FL_ECC_DECODE_FAIL 0xE

#define ReadAddrLong(addr) (*((vuint32_t *) (addr)))
#define RegFieldSet(preg, bit, value) preg &=          // return code ECC decode test pass
        ~bit(0xFFFFFFFF); preg |= bit(value)           // return code ECC decode test fail

                                                       // macros for reading/setting registers
                                                       // or addresses
```

Figure 31. Code example: FLASH_SW_ECCTEST (definitions)

```
FLASH_SW_ECCTEST() {

return_Code = 0xF9F99999;
C90FL_UT0= C90FL_UT_PASSWORD;                          // enable user-test mode
C90FL_UT0 |= UT0_SBCE;                                 // enable single-bit error correction
for(k = 0; k < 2; k++) {                               // visibility
   C90FL_UT0 |= UT0_EIE;                               // loop for two double-words in a
   C90FL_UT1 = 0xFFFFFFFF;                             // page
   C90FL_UT2 = 0xFFFFFFFF;                             // set EIE
   RegFieldSet(C90FL_UT0, UT0_DSI, 0xFF);              // Initialize UT0[DSI], UT1, UT2
                                                       // registers with all “1”s
   for(i = 0; i <= 72; i++) {
       if(i < 32) C90FL_UT1 = 0xFFFFFFFF & (~(1<<i));
       else if(i < 64) {                               // walking “0” through data and
           C90FL_UT1 = 0xFFFFFFFF;                     // ECC bits
           C90FL_UT2 = 0xFFFFFFFF & (~(1<<(i-32))); }  // walking ‘0’ in first word
       else if(i < 72) {
           C90FL_UT1 = 0xFFFFFFFF;
           C90FL_UT2 = 0xFFFFFFFF;                     // walking ‘0’ in second word
           RegFieldSet(C90FL_UT0, UT0_DSI, (0xFF &
(~(1<<(i-64)))));}
       else {
           C90FL_UT1 = 0xFFFFFFFF;                     // walking ‘0’ in ECC parity bits
           C90FL_UT2 = 0xFFFFFFFF;                     // all “1”s data vector
           RegFieldSet(C90FL_UT0, UT0_DSI, 0xFF); }
       addr = rdAddr + k*8 + i*0x40;
       C90FL_ADR = addr;                               // the address will skip 0x40 to
       data0 = ReadAddrLong(addr);                     // make
       data1 = ReadAddrLong(addr+4);                   // sure every read is fetch from
                                                       // Flash memory
       if(i = = 72) {                                  // do a normal read to address
           if( (data0 != 0xFFFFFFFF) ||                // saved
               (data1 != 0xFFFFFFFF) ||                // in ADR
               ((C90FL_MCR & MCR_SBC) != 0)) {
               return_Code = C90FL_ECC_DECODE_FAIL;
               break;                                  // check for expected data and ECC
           }                                           // correction
       }
       else {
           if( (data0 != 0xFFFFFFFF) ||
               (data1 != 0xFFFFFFFF) ||
               ((C90FL_MCR & MCR_SBC) = = 0)) {
               return_Code = C90FL_ECC_DECODE_FAIL;
               break;
           }
           else C90FL_MCR |= MCR_SBC;
       }
   }
}
C90FL_UT0 = 0;                                         // clear SBC
return return_Code;}
```

Return Values:
C90FL_OK: ECC logic check pass
C90FL_ECC_DECODE_FAIL: ECC logic check fail

Figure 32. Code example: FLASH_SW_ECCTEST

8.1.2 *<module>*_SWTEST_REGCRC

An example of how to implement ADC0_SWTEST_REGCRC is shown in this section. ADC_0 registers are processed by eDMA channel 1. Also, there is an example of how to add another function by the inclusion of SIUL_SWTEST_REGCRC for the SIUL. The SIUL registers for SIUL_SWTEST_REGCRC are processed by eDMA channel 2. CRC Checks for other modules (for example, eTimer, FlexPWM) can be added in a similar way.

The content of the registers which are to be monitored are transferred to the CRC with the scatter/gather algorithm of the eDMA module. Scatter/gather enables a DMA channel to scatter the DMA data to multiple destinations or gather it from multiple sources. Please refer to the “Dynamic programming” section in the “Enhanced Direct Memory Access (eDMA)” chapter of the *Qorivva MPC5675K Microcontroller Reference Manual* for detailed information about scatter/gather.

Figure 33 gives an example to initialize the CRC module. Different CRC contexts may be used for different eDMA channels.

```
init_CRC (sub_module, CRC_mode) {

CRC.CNTX[0].CFG.WORD = CRC_mode;
while(CRC.CNTX[sub_module].CFG.WORD!=CRC_mode) {}        // Failed

CRC.CNTX[0].CSTAT.WORD = 0xFFFFFFFF;                     // set seed
}
```

Parameters:
sub_module: CRC-context to be used
CRC_mode: CRC configuration (for example, 6)

Figure 33. Code example: CRC initialization

TCD structures are initialized, which contain all configurations for a module to be checked as a linked list. Before starting the algorithm, the first TCD structure of each module is uploaded (Init_DMA_TCD_SG) to an eDMA channel (Figure 34).

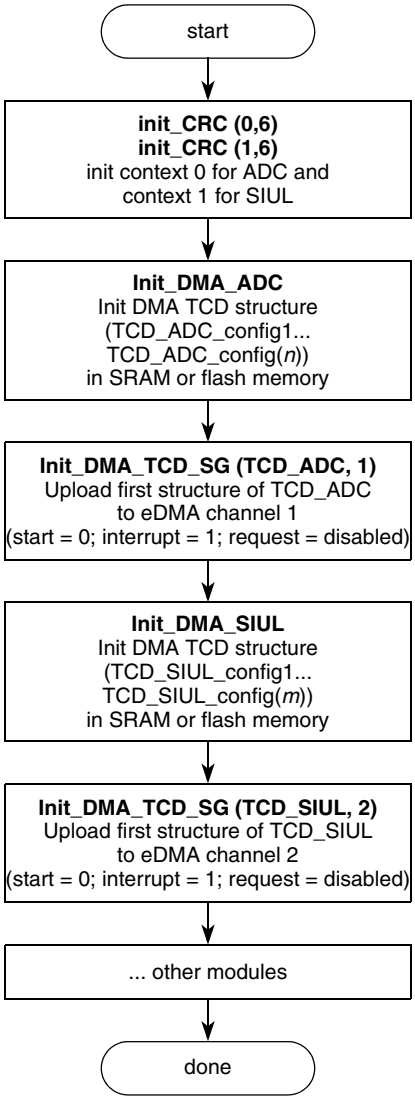


Figure 34. TCD structures configuration and upload

Each TCD structure specifies a register (or a couple of contiguous registers) to be monitored as well as corresponding eDMA parameters. An example for ADC_0 (TCD_ADC_config) is shown in Figure 35 and the upload of the first structure to the eDMA is shown in Figure 36.

If a check of the second ADC registers is required (ADC1_SWTEST_REGCRC, ADC2_SWTEST_REGCRC, ADC3_SWTEST_REGCRC), they can simply be added to the ADC_0 configuration shown in Figure 35 without using any additional eDMA channel.

```
Init_DMA_ADC (){

TCD_ADC_config1.sadr = & ADC0.IMR.R;                    // ADC_0 - IMR & CIMR0
TCD_ADC_config1.smod = 0;                               // source address modulo
TCD_ADC_config1.ssize= 2;                               // source data transfer size
TCD_ADC_config1.dmod= 0;                                // destination address Module
TCD_ADC_config1.dsize= 2;                               // destination data transfer size
TCD_ADC_config1.soff= 4;                                // source address signed offset
TCD_ADC_config1.nbytes= 8;                              // inner minor byte transfer count
TCD_ADC_config1.slast= 0;                               // last source address adjustment
TCD_ADC_config1.dadr=& CRC.CNTX[0].INP.R;               // destination address = input of CRC
TCD_ADC_config1.citer_e_link= 0;                        // context 0
TCD_ADC_config1.citer_linkch= 0;                        // enable c2c linking (minor loop compl.)
TCD_ADC_config1.citer= 1;                               // link channel number
TCD_ADC_config1.doff= 0;                                // current major iteration count
TCD_ADC_config1.dlast_sga=&TCD_ADC_config2;             // destination address signed offset
TCD_ADC_config1.biter_e_link= 0;
TCD_ADC_config1.biter_linkch= 0;
TCD_ADC_config1.biter= 1;                               // enable c2c linking (minor loop compl.)
TCD_ADC_config1.bwc= 0;                                 // bandwidth control
TCD_ADC_config1.major_linkch= 0;                        // link channel number
TCD_ADC_config1.done= 0;                                // channel done
TCD_ADC_config1.active= 0;                              // channel active
TCD_ADC_config1.major_e_link= 0;                        // enable c2c linking (major loop compl.)
TCD_ADC_config1.e_sg= 1;                                // enable scatter/gather processing
TCD_ADC_config1.d_req= 0;                               // disable request
TCD_ADC_config1.int_half= 0;                            // enable interrupt (major count)
TCD_ADC_config1.int_maj= 0;                             // enable interrupt (major iteration)
TCD_ADC_config1.start= 0;                               // channel start
TCD_ADC_config1.smloe= 0;
TCD_ADC_config1.dmloe= 0;
TCD_ADC_config1.mmloff= 0;

TCD_ADC_config2.sadr= & ADC0.WTISR.R;                   // ADC_0 - WTISR & WTIMR
TCD_ADC_config2.dlast_sga= &TCD_ADC_config3;
TCD_ADC_config2.d_req= 0;
TCD_ADC_config2.int_maj= 0;
TCD_ADC_config2.start= 1;
TCD_ADC_config2.smloe= 0;
[all other TCD elements]

TCD_ADC_config(n).sadr= & ADC0.DMAE.R;                  // ADC_0 - DMAE & DMAR0
TCD_ADC_confign(n).dlast_sga= &TCD_ADC_config1;
TCD_ADC_confign(n).d_req= 1;
TCD_ADC_confign(n).int_maj= 1;
TCD_ADC_confign(n).start= 1;
TCD_ADC_confign(n).smloe= 0;
[all other TCD elements]
}
```

Figure 35. Code example: TCD_ADC configuration in SRAM

```
Init_DMA_TCD_SG (T, channel){

DMA_CH_MUX.CHCONFIG[channel].B.SOURCE = 28;
DMA_CH_MUX.CHCONFIG[channel].B.TRIG = 0;
DMA_CH_MUX.CHCONFIG[channel].B.ENBL = 1;

SPP_DMA2.CHANNEL[channel].TCDWORD0_.B.SADDR = T.sadr;
SPP_DMA2.CHANNEL[channel].TCDWORD16_.B.DADDR = T.dadr;
SPP_DMA2.CHANNEL[channel].TCDWORD4_.B.SMOD = T.smod;
SPP_DMA2.CHANNEL[channel].TCDWORD4_.B.DMOD = T.dmod;
SPP_DMA2.CHANNEL[channel].TCDWORD4_.B.DSIZE = T.dsize;
SPP_DMA2.CHANNEL[channel].TCDWORD4_.B.SSIZE = T.ssize;
SPP_DMA2.CHANNEL[channel].TCDWORD4_.B.SOFF = T.soff;
SPP_DMA2.CHANNEL[channel].TCDWORD20_.B.DOFF = T.doff;
SPP_DMA2.CHANNEL[channel].TCDWORD12_.B.SLAST = T.slast;
[…]                                                          // add all other
}
```

Parameters:
T: TCD structures
channel: eDMA channel

Figure 36. Code example: first structure upload to eDMA

Figure 37 shows the execution flow of the eDMA modules scatter/gather algorithm. Once the first channel is started by setting DMASERQ = 1, all structures (TCD_ADC_config) are executed consecutively. An interrupt is issued after the last eDMA transfer has been finished (for example, TCD_ADC_config(*n*) in Figure 37). Then the calculated CRC value for all TCD_ADC_config1…TCD_ADC_config(*n*) can then

be compared to the expected (predetermined) CRC value. All channels used for <*module*>_SWTEST_REGCRC are repetitively started (DMASERQ) within the FTTI.

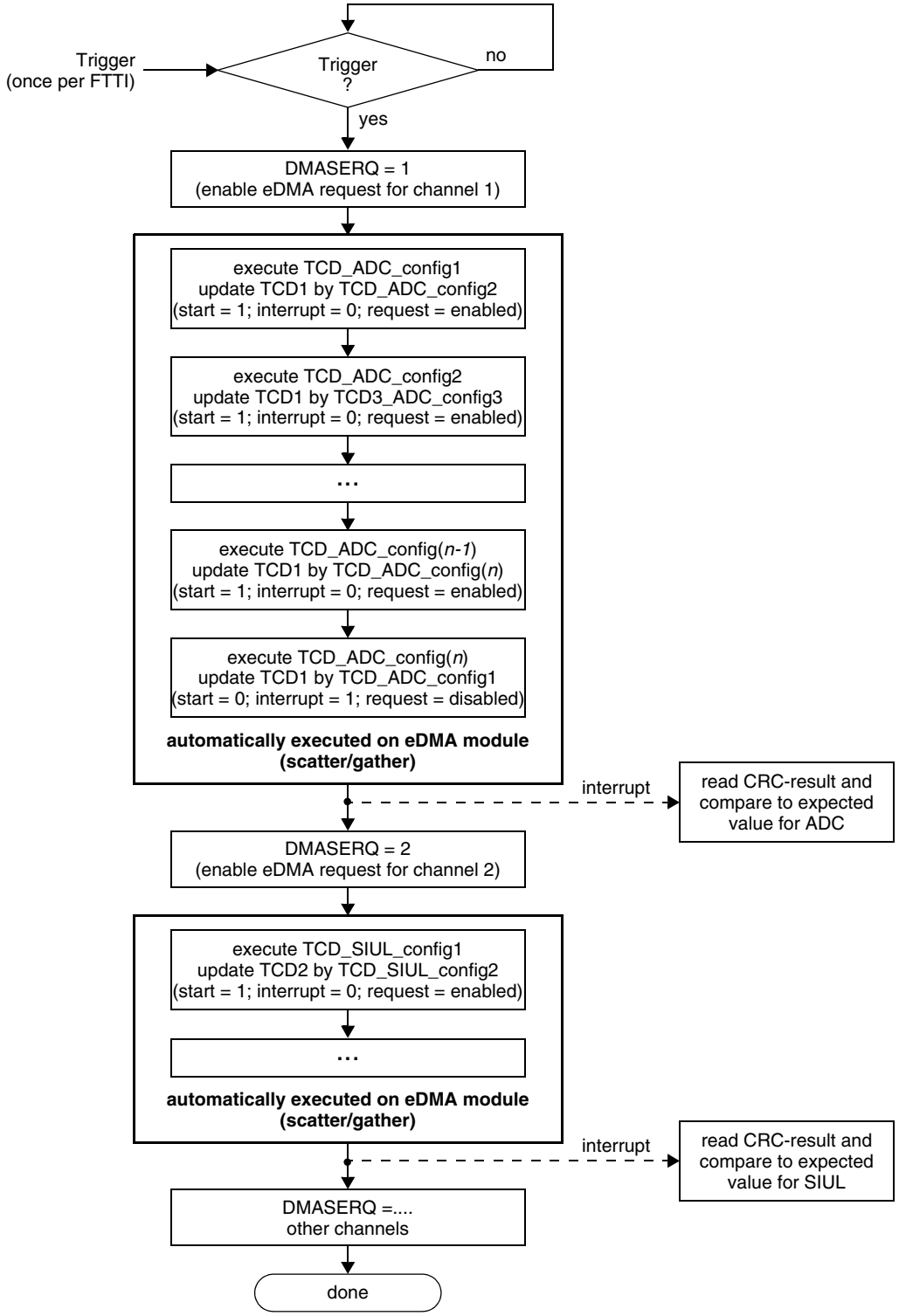


Figure 37. <*module*>_SWTEST_REGCRC flow diagram

8.1.3 CTU

```
Init_CTU(trig, mode) {

if (trig <= 3) {
        if (mode) {
                CTU.THCR1.R |= (1 <<          // enable ETIMER_0 output
((trig*8)+1)); }
        else {                                // disable ETIMER_0 output
                CTU.THCR1.R &= ~(1 <<
((trig*8)+1)); }
        }
else {
        trig -= 4;                            // enable ETIMER_0 output
        if (mode) {
                CTU.THCR2.R |= (1 <<          // enable ETIMER_0 output
((trig*8)+1)); }
        else {
                CTU.THCR2.R &= ~(1 <<
((trig*8)+1));                                // set some BS value to avoid compare error
        }
}
                                              // set Global Reload
CTU.TGSCRR.R = 0x10;                          // load new configuration into CTU
CTU.TGSCCR.R = 0x100;
                                              // clear the CMP_ERR bit (??)
CTU.CTUCR.B.GRE = 1;
CTU.CTUCR.B.MRS_SG = 1;                       // clear any pending interrupts

CTU.CTUEFR.R |= 0x0400;                       // enable error interrupts (CTU reports overruns via
                                              // interrupts)
dummy = CTU.CTUIFR.R;
dummy = CTU.CTUEFR.R;
CTU.CTUIR.B.IEE = 1;
}
```

Parameters
trig: selects the trigger to be linked to the timer
mode: controls the SET/CLEAR of the selected trigger

Figure 38. Code example: CTU initialization

8.1.3.1 CTU_HWSWTEST_ADCCOMMAND

```
CTU_HWSWTEST_ADCCOMMAND (){

{
[#define ADC_NUMBER 4;]                                  // number of test channels - max 24

ADC_seq[ADC_NUMBER] = {11, 12, 13, 14};                 // channels of ADC to be measured
temp[2*ADC_NUMBER] = {0,0,0,0,0,0,0,0};                 // data from CTU FIFO memory


CTU.CTUCR.B.MRS_SG = 1;                                 // start

while (!(ADC0.CDR[ADC_seq[ADC_NUMBER-1]].B.VALID));

for (i = 0; i<(ADC_NUMBER*2); i++) {
         switch (i/2) {
                  case 0:temp[i] = CTU_0.FR0.R;break;
                  case 1:temp[i] = CTU_0.FR1.R;break;
                  case 2:temp[i] = CTU_0.FR2.R;break;
                  case 3:temp[i] = CTU_0.FR3.R;break;
         }
}

for (i = 0; i<ADC_NUMBER*2; i++) {
         if (ADC_seq[i/2] != ((temp[i]>>16)&0xF))       // check channels and ADCs
return FAIL;                                            // ADC_0 has value “1” and ADC_1
         if (((temp[i]>>20)&0x1)==(i%2)) return FAIL;   // has value ‘0’ in FIFO memory!
}
return PASS;
}
```

Return Values:
PASS: ADC number and channel number are correct
FAIL: otherwise

Figure 39. Code example: CTU_HWSWTEST_ADCCOMMAND

8.1.3.2 CTU_SWTEST_ETIMERCOMMAND

An example configuration for implementing the CTU_SWTEST_ETIMERCOMMAND is shown in Figure 40. Channel 2 of eTimer_0 generates the Master Reload Signal (MRS) for the CTU. Based on this signal, the CTU generates 8 output triggers in triggered mode. These are counted by channel 0 of

eTimer_0. With every interrupt from channel 2 of eTimer_0 occurring with a new MRS, the counter register (CNTR) of channel 0 can be verified for accuracy.

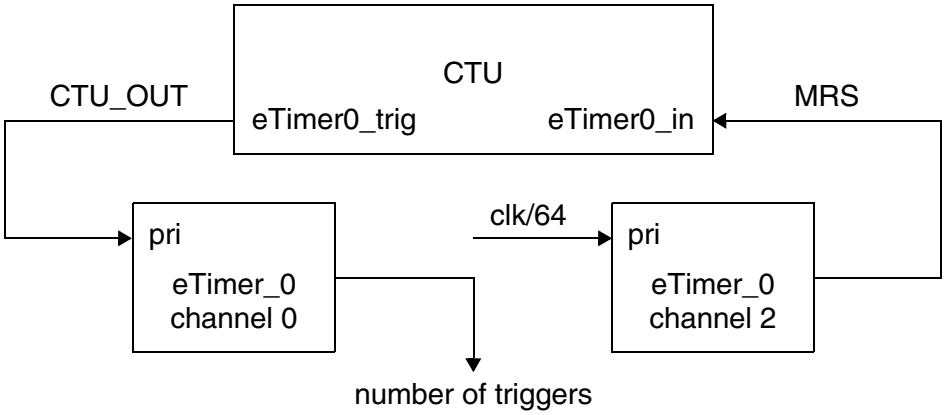


Figure 40. Configuration for sequential mode example 1

```
void Init_CTU(uint16_t period,uint8_t nof){

CTU.TGSCR.B.PRES = 0x3;                          //Configure TGS counter - prescaler 4
CTU.TGSCR.B.MRS_SM = 26;                         //MRS signal from eTimer_0 [2]
CTU.TGSCRR.R = 0x0;
CTU.TGSCCR.R = 0x7FFF;

if (period < 9) CTU.COTR.R = 30*period;          //always 1/4 total period, max. 2.125 µs
else CTU.COTR.R = 0xFF;

CTU.CTUCR.B.T0_SG = 0x0;

if (period>273) period = 273*30;                 // keep value inside 16-bit range of
else period*=30;                                 // all triggers during 1
                                                 // trigger compare registers
if (nof>=1){                                     //configuration scheduler unit
         CTU.T0CR.R = period;
         CTU.THCR1.B.T0_E = 0x1;
         CTU.THCR1.B.T0_ETE = 0x1;
         CTU.THCR1.B.T0_T1E = 0x1;
}
[… trigger 1 to 7]

if (nof>=8){
         CTU.T7CR.R = 8*period;
         CTU.THCR2.B.T7_E = 0x1;
         CTU.THCR2.B.T7_ETE = 0x1;
         CTU.THCR2.B.T7_T1E = 0x1;
}

CTU.TGSISR.B.I13_RE = 0x1;                       //allow external signal for MRS - eTimer_0 [2]
CTU.CTUCR.B.TGSISR_RE = 0x1;                     //reload global setting
CTU.CTUCR.B.GRE = 0x1;
CTU.CTUIR.B.MRS_IE = 0x1;

CTU.CTUIR.B.IEE = 1;                             //enable error interrupts
}
```

Parameters
period: period of trigger to be generated
nof: number of trigger to be generated

Figure 41. Code example: CTU initialization

```
Init_Etimer(period,nof) {

ETIMER_0.ENBL.R = 0;                              //stop all channels

ETIMER_0.CHANNEL[0].COMP1.R = 0x0;                // set compare and load values to zero
ETIMER_0.CHANNEL[0].COMP2.R = 0x0;
ETIMER_0.CHANNEL[0].LOAD.R = 0x0;

ETIMER_0.CHANNEL[0].CTRL.B.CNTMODE = 0x1;         // count rising edges of primary source
ETIMER_0.CHANNEL[0].CTRL.B.PRISRC = 0x8;          // primary source IPB 1:1 and capturing on rising
ETIMER_0.CHANNEL[0].CTRL.B.SECSRC = 0x6;          // edge of secondary source (output CTU)
ETIMER_0.CHANNEL[0].CTRL.B.LENGTH = 0x0;
ETIMER_0.CHANNEL[0].CTRL2.R = 0x0;

ETIMER_0.CHANNEL[0].INTDMA.R = 0x0;               // disable DMA, interrupts, input filter, etc.
ETIMER_0.CHANNEL[0].CMPLD1.R = 0x0;
ETIMER_0.CHANNEL[0].CMPLD2.R = 0x0;
ETIMER_0.CHANNEL[0].FILT.R = 0x0;

ETIMER_0.ENBL.R |= (1<<0);                        // enable timer channel

ETIMER_0.CHANNEL[2].COMP1.R =                     // delay between MRS pulses
2*(nof*period+2);                                 // 2*(nof*period+2) approximately
ETIMER_0.CHANNEL[2].COMP2.R =                     120/64*(nof*period+2)
2*(nof*period+2)+0xA;                             // 2 - is reserve between two pulses MRS for checking
                                                  // numbers of triggers events
ETIMER_0.CHANNEL[2].LOAD.R = 0x0;                 // 120/64 - it is period of input signal (primary source)
                                                  of
                                                  // eTimer_0 [2]
                                                  // 0xA = width of MRS signal
ETIMER_0.CHANNEL[2].CTRL.B.CNTMODE = 0x1;         // Count rising edges of primary source
ETIMER_0.CHANNEL[2].CTRL.B.PRISRC = 0x1E;         //primary source IPB 1:1 and capturing on rising
ETIMER_0.CHANNEL[2].CTRL.B.SECSRC = 0x6;          //edge of secondary source (output CTU)
ETIMER_0.CHANNEL[2].CTRL.B.LENGTH = 0x1;

ETIMER_0.CHANNEL[2].CTRL2.B.OEN = 0x1;
ETIMER_0.CHANNEL[2].CTRL2.B.OUTMODE = 0x8;

ETIMER_0.CHANNEL[2].CCCTRL.B.CLC2 = 0x7;
ETIMER_0.CHANNEL[2].CCCTRL.B.CPT2MODE = 0x0;
ETIMER_0.CHANNEL[2].CCCTRL.B.CPT1MODE = 0x0;
ETIMER_0.CHANNEL[2].CCCTRL.B.CFWM = 0x0;
ETIMER_0.CHANNEL[2].CCCTRL.B.ARM = 0x0;

ETIMER_0.CHANNEL[2].INTDMA.R = 0x0;
ETIMER_0.CHANNEL[2].CMPLD1.R = 0x0;               //disable DMA, interrupts, input filter, etc.
ETIMER_0.CHANNEL[2].CMPLD2.R = 0x0;
ETIMER_0.CHANNEL[2].FILT.R = 0x0;

ETIMER_0.ENBL.R |= (1<<2);
}                                                 //enable timer channel
```

Parameters
period: period of trigger to be generated
nof: number of trigger to be generated

Figure 42. Code example: eTimer initialization

8.1.3.3 CTU_HWSWTEST_TRIGGEROVERRUN

```
CTU_HWSWTEST_TRIGGEROVERRUN (* store_var){

temp = (*store_var & 0x0BE0);

if (temp) {                                   // mask all errors except external trigger,
         return (FAIL);                       // timer N triggers and ADC command
         }                                    // overrun

return (PASS);                                // if there are any, this is a error
}                                             // error detected, thus FAIL
```

Parameters
*store_var: When the CTUEFR register is read, it's content is cleared. The original content is stored in address give by store_var

Return Values:
PASS: no overrun
FAIL: otherwise

Figure 43. Code example: CTU_HWSWTEST_TRIGGEROVERRUN

8.1.3.4 CTU_SWTEST_TRIGGERTIME (sequential mode)

An example configuration for implementing the CTU_SWTEST_TRIGGERTIME for eight triggers in sequential mode is shown in Figure 44. Channel 2 of eTimer_0 generates an Event Signal (EV) for the CTU. Channel 2 of eTimer_1 generates the MRS for the CTU. Based on these signals, the CTU generates the desired delayed output signal (CTU_OUT). In this example, the delay is configured to increase with every event signal (defined in CTU initialization shown in Figure 44). Channel 0 of eTimer_0 drives a signal which is high during the delay caused by the CTU (from rising edge of EV to rising edge of CTU_OUT). Channel 1 of eTimer_0 measures the delay which can then be checked for accuracy.

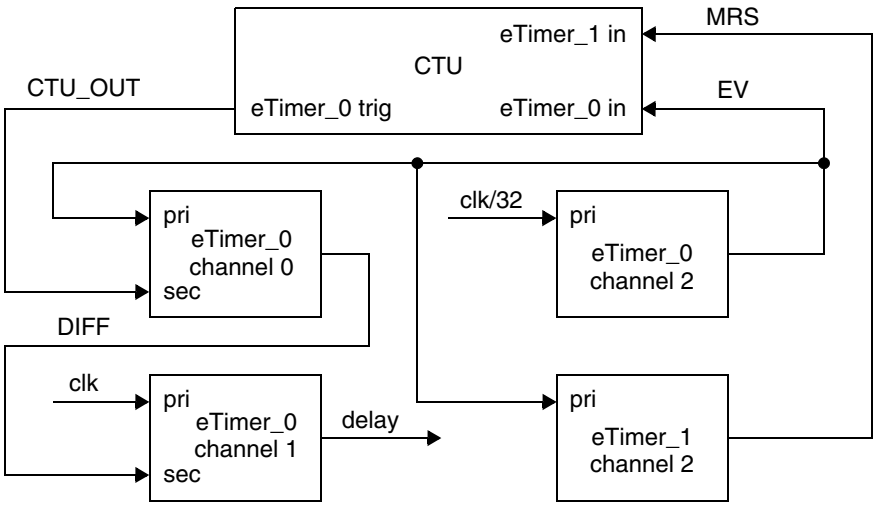


Figure 44. Configuration for sequential mode example 2

Figure 45 shows a timing example of this configuration.

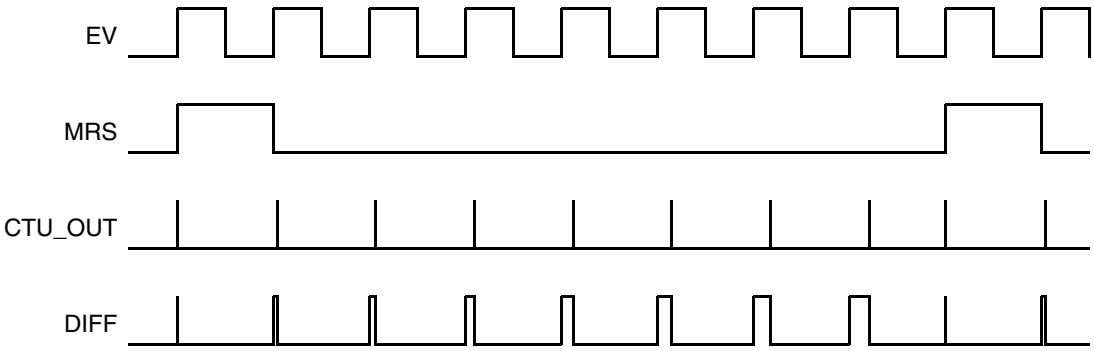


Figure 45. Timing for sequential mode example 2

Example code for eTimers and CTU are shown in Figure 46, Figure 47 and Figure 48.

```
Init_Etimer_seq () {

ETIMER_0.ENBL.R = 0;                                    // stop all channels
ETIMER_0.CHANNEL[2].COMP1.R = 0x400;                    // set compare and load values to zero
ETIMER_0.CHANNEL[2].COMP2.R = 0x800;
ETIMER_0.CHANNEL[2].LOAD.R = 0x0;

ETIMER_0.CHANNEL[2].CTRL.B.CNTMODE = 0x1;               // count rising edges of primary source
ETIMER_0.CHANNEL[2].CTRL.B.PRISRC = 0x1D;               // IP BUS 1:32
ETIMER_0.CHANNEL[2].CTRL.B.LENGTH = 0x1;
ETIMER_0.CHANNEL[2].CTRL2.B.OEN = 0x1;
ETIMER_0.CHANNEL[2].CTRL2.B.OUTMODE = 0x8;

ETIMER_0.CHANNEL[2].CCCTRL.B.CLC1 = 0x7;                // disable DMA, interrupts, input filter
ETIMER_0.CHANNEL[2].CCCTRL.B.CPT2MODE = 0x0;
ETIMER_0.CHANNEL[2].CCCTRL.B.CPT1MODE = 0x0;
ETIMER_0.CHANNEL[2].CCCTRL.B.CFWM = 0x3;

ETIMER_0.CHANNEL[2].INTDMA.R = 0x0;
ETIMER_0.CHANNEL[2].CMPLD1.R = 0x0;
ETIMER_0.CHANNEL[2].CMPLD2.R = 0x0;
ETIMER_0.CHANNEL[2].FILT.R = 0x0;

ETIMER_0.ENBL.R |= (1<<2);                              // enable etimer_0 channel 2


ETIMER_1.ENBL.R = 0;                                    // stop all channels
ETIMER_1.CHANNEL[2].COMP1.R = 0x6;                      // set compare and load values to zero
ETIMER_1.CHANNEL[2].COMP2.R = 0x7;
ETIMER_1.CHANNEL[2].LOAD.R = 0x0;

ETIMER_1.CHANNEL[2].CTRL.B.CNTMODE = 0x1;               // count rising edges of primary source
ETIMER_1.CHANNEL[2].CTRL.B.PRISRC = 0x9;                // auxiliary input #1 (= eTimer_0[2] out)
ETIMER_1.CHANNEL[2].CTRL.B.LENGTH = 0x1;
ETIMER_1.CHANNEL[2].CTRL2.B.OEN = 0x1;
ETIMER_1.CHANNEL[2].CTRL2.B.OUTMODE = 0x8;

ETIMER_1.CHANNEL[2].CCCTRL.B.CLC1 = 0x7;                // disable DMA, interrupts, input filter
ETIMER_1.CHANNEL[2].CCCTRL.B.CPT2MODE = 0x0;
ETIMER_1.CHANNEL[2].CCCTRL.B.CPT1MODE = 0x0;
ETIMER_1.CHANNEL[2].CCCTRL.B.CFWM = 0x3;

ETIMER_1.CHANNEL[2].INTDMA.R = 0x0;
ETIMER_1.CHANNEL[2].CMPLD1.R = 0x0;
ETIMER_1.CHANNEL[2].CMPLD2.R = 0x0;
ETIMER_1.CHANNEL[2].FILT.R = 0x0;

ETIMER_1.ENBL.R |= (1<<2);                              // enable eTimer_1 channel 2

[continue on next page]
```

Figure 46. Code example: eTimer initialization (seq.)

```
ETIMER_0.CHANNEL[0].COMP1.R = 0x1;                    //set compare and load values to zero
ETIMER_0.CHANNEL[0].COMP2.R = 0x0;
ETIMER_0.CHANNEL[0].LOAD.R = 0x0;

ETIMER_0.CHANNEL[0].CTRL.B.CNTMODE = 0x3;
ETIMER_0.CHANNEL[0].CTRL.B.PRISRC = 0x12;             // counter #2 output
ETIMER_0.CHANNEL[0].CTRL.B.SECSRC = 0x8;              // auxiliary input_0 (from CTU)
ETIMER_0.CHANNEL[0].CTRL.B.LENGTH = 0x1;
ETIMER_0.CHANNEL[0].CTRL2.B.SIPS = 0x1;
ETIMER_0.CHANNEL[0].CTRL2.B.OPS = 0x1;
ETIMER_0.CHANNEL[0].CTRL2.B.OEN = 0x1;
ETIMER_0.CHANNEL[0].CTRL2.B.OUTMODE = 0x8;

ETIMER_0.CHANNEL[0].CCCTRL.B.CLC1 = 0x6;              // disable DMA, interrupts, input filter
ETIMER_0.CHANNEL[0].CCCTRL.B.CPT2MODE = 0x0;
ETIMER_0.CHANNEL[0].CCCTRL.B.CPT1MODE = 0x0;
ETIMER_0.CHANNEL[0].CCCTRL.B.CFWM = 0x3;

ETIMER_0.CHANNEL[0].INTDMA.R = 0x0;                   // enable eTimer_0 channel 0
ETIMER_0.CHANNEL[0].CMPLD1.R = 0x0;
ETIMER_0.CHANNEL[0].CMPLD2.R = 0x0;
ETIMER_0.CHANNEL[0].FILT.R = 0x0;                     // eTimer_0[1] measure width impulse
                                                      // set compare and load values to zero
ETIMER_0.ENBL.R |= (1<<0);


ETIMER_0.CHANNEL[1].COMP1.R = 0xEA60;
ETIMER_0.CHANNEL[1].COMP2.R = 0x0;
ETIMER_0.CHANNEL[1].LOAD.R = 0x0;

ETIMER_0.CHANNEL[1].CTRL.B.CNTMODE = 0x3;
ETIMER_0.CHANNEL[1].CTRL.B.PRISRC = 0x18;             // IP BUS 1:1
ETIMER_0.CHANNEL[1].CTRL.B.SECSRC = 0x10;             // counter #0 output
ETIMER_0.CHANNEL[1].CTRL.B.LENGTH = 0x1;
ETIMER_0.CHANNEL[1].CTRL2.B.SIPS = 0x0;

ETIMER_0.CHANNEL[1].CCCTRL.B.CLC1 = 0x6;              //disable DMA, interrupts, input filter
ETIMER_0.CHANNEL[1].CCCTRL.B.CPT2MODE = 0x0;
ETIMER_0.CHANNEL[1].CCCTRL.B.CPT1MODE = 0x0;
ETIMER_0.CHANNEL[1].CCCTRL.B.CFWM = 0x3;

ETIMER_0.CHANNEL[1].INTDMA.R = 0x0;                   // enable eTimer_0 channel 1
ETIMER_0.CHANNEL[1].CMPLD1.R = 0x0;
ETIMER_0.CHANNEL[1].CMPLD2.R = 0x0;
ETIMER_0.CHANNEL[1].FILT.R = 0x0;

ETIMER_0.ENBL.R |= (1<<1);
}
```

Figure 47. Code example: eTimer initialization (seq.)

```
Init_CTU_seq(uint16_t period) {

CTU.TGSCR.B.PRES = 0x3;                          //TGS counter - prescaler 4
CTU.TGSCR.B.TGS_M = 0x1;                         // sequential mode
CTU.TGSCR.B.MRS_SM = 0x1C;
CTU.TGSCR.B.ET_TM = 0x0;

CTU.TGSCRR.R = 0x0;
CTU.TGSCCR.R = 0x2000;

if (period < 9) CTU.COTR.R = 30*period;          // always 1/4 total period, max. 2.125 µs
else CTU.COTR.R = 0xFF;

if (period>546) period = 546*30;                 // keep value inside 16-bit range of
else period*=30;                                 // trigger compare registers

CTU.COTR.R = 0xAF;                               //set values of comparators
                                                 // [µs] - [base 10] - [base 16]
CTU.T0CR.R = period;                             // 5 - 600 - 258
CTU.T1CR.R = 3*period;                           // 15 - 1800 - 708
CTU.T2CR.R = 6*period;                           // 30 - 3600 - E10
CTU.T3CR.R = 10*period;                          // 50 - 6000 - 1770
CTU.T4CR.R = 15*period;                          // 75 - 9000 - 2328
CTU.T5CR.R = 21*period;                          // 105 - 12600 - 3138
CTU.T6CR.R = 28*period;                          // 140 - 16800 - 41A0
CTU.T7CR.R = 36*period;                          // 168 - 20160 - 4EC0

CTU.THCR1.B.T0_E = 0x1;                          // configuration scheduler unit
CTU.THCR1.B.T0_ETE = 0x1;
CTU.THCR1.B.T0_T1E = 0x1;
[… repetition for trigger 1 to 7]

CTU.CTUEFR.R = 0xFFFF;
CTU.CTUIFR.R = 0xFFFF;
CTU.CTUIR.R = 0x0;
                                                 // reload global setting
CTU.TGSISR.B.I13_RE = 0x1;                       // source eTimer_1 rising edge
CTU.TGSISR.B.I14_RE = 0x1;                       // source eTimer_2 rising edge

CTU.CTUCR.B.TGSISR_RE = 0x1;                     // reload TGSISR reload enable

CTU.CTUIR.B.T0_I = 0x1;                          // allowed interrupts
[… repetition for interrupts 1 to 7]

CTU.CTUIR.B.MRS_IE = 0x1;                        // reload global settings
                                                 // enable error interrupts
CTU.CTUCR.B.GRE = 0x1;                           // MRS to start CTU
CTU.CTUIR.B.IEE = 0x0;
CTU.CTUCR.B.MRS_SG = 0x1;
}
```

Figure 48. Code example: CTU initialization (seq.)

An updated measurement result of eTimer_0 channel 1 issues an interrupt, and the updated result may be stored in a variable. After a complete cycle of eight triggers, the measured delay can be compared to the expected delay value (Figure 49).

```
CTU_seq_test(tol, period){

if ( (abs(delay_0[0] - 120*1*period) > tol) ||         //checking of measured values
         (abs(delay_0[1] - 120*3*period) > tol) ||
         (abs(delay_0[2] - 120*6*period) > tol) ||
         (abs(delay_0[3] - 120*10*period) > tol) ||
         (abs(delay_0[4] - 120*15*period) > tol) ||
         (abs(delay_0[5] - 120*21*period) > tol) ||
         (abs(delay_0[6] - 120*28*period) > tol) ||
         (abs(delay_0[7] - 120*36*period) > tol) )
return FAIL;
         else return PASS;
}
```

Figure 49. Code example: CTU_SWTEST_TRIGGERTIME (seq.)

8.1.3.5 CTU_SWTEST_TRIGGERTIME

Figure 50 shows an example of eTimer_2 initialization. eTimer_2 is configured to count rising edges of the IP Bus clock (assumed f = 120 MHz) as primary count source and rising edges of auxiliary input 0 (output from CTU) as secondary count source.

```
Init_Etimer_trigger () {

ETIMER_2.ENBL.R = 0;                                  // stop all channels
ETIMER_2.CHANNEL[0].COMP1.R = 0x0;                    // set compare and load values to zero
ETIMER_2.CHANNEL[0].COMP2.R = 0x0;
ETIMER_2.CHANNEL[0].LOAD.R = 0x0;

ETIMER_2.CHANNEL[0].CTRL.B.CNTMODE = 0x1;             // Count rising edges of primary source
ETIMER_2.CHANNEL[0].CTRL.B.PRISRC = 0x18;             // IP Bus clock divide by 1 prescaler
ETIMER_2.CHANNEL[0].CTRL.B.SECSRC = 0x8;              // Auxiliary input #0
ETIMER_2.CHANNEL[0].CTRL2.R = 0x0;

ETIMER_2.CHANNEL[0].CCCTRL.B.CPT2MODE = 0x2;          // rising edges of secondary count source
ETIMER_2.CHANNEL[0].CCCTRL.B.CPT1MODE = 0x2;          // rising edges of secondary count source
ETIMER_2.CHANNEL[0].CCCTRL.B.CFWM = 0x3;
ETIMER_2.CHANNEL[0].CCCTRL.B.ARM = 0x1;

ETIMER_2.CHANNEL[0].INTDMA.R = 0x0;                   // disable DMA, interrupts, input filter
ETIMER_2.CHANNEL[0].CMPLD1.R = 0x0;
ETIMER_2.CHANNEL[0].CMPLD2.R = 0x0;
ETIMER_2.CHANNEL[0].FILT.R = 0x0;

ETIMER_2.ENBL.R |= (1<<0);                            // enable etimer channel
}
```

Figure 50. Code example: Etimer initialization (triggered)

Figure 51 shows an example of CTU initialization.

```
Init_CTU_trigger(period) {

CTU.TGSCR.B.PRES = 0x3;                                  // TGS counter - prescaler 4
CTU.TGSCR.B.TGS_M = 0x0;                                 // triggered mode
CTU.TGSCRR.R = 0x0;                                      // 2700
CTU.TGSCCR.R = 0xA8C;
                                                         // always 1/4 total period, max. 2.125 µs
if (period < 9) CTU.COTR.R = 120*period/4;
else CTU.COTR.R = 0xFF;

CTU.CTUCR.B.T0_SG = 0x0;
                                                         // keep value inside 16-bit range of
if (period>546) period = 546 * 30;                       // trigger compare registers
else period*=30;
                                                         // set values of comparators
CTU.T0CR.R = period;
CTU.T1CR.R = 2*period;
CTU.T2CR.R = 3*period;
CTU.T3CR.R = 4*period;
                                                         // configuration scheduler unit
CTU.THCR1.B.T0_E = 0x1;
CTU.THCR1.B.T0_ETE = 0x1;
CTU.THCR1.B.T0_T3E = 0x1;

[… repetition for timer 1 to 3]
                                                         // reload global setting
CTU.CTUCR.B.GRE = 0x1;                                   // enable error interrupts
CTU.CTUIR.B.IEE = 1;
}
```

Figure 51. Code example: CTU initialization (triggered)

Figure 52 shows the consecutive CTU trigger event generation. The time of each rising edge is stored in CAPT_1 and CAPT_2 FIFOs. Once a capture occurs on capture circuit 1, capture circuit 1 is disarmed and capture circuit 2 is armed and vice versa. The safety integrity measure CTU_SWTEST_TRIGGERTIME (see Figure 53) calculates the period of the CTU signal and reports FAIL, in case the period does not match the expectations.

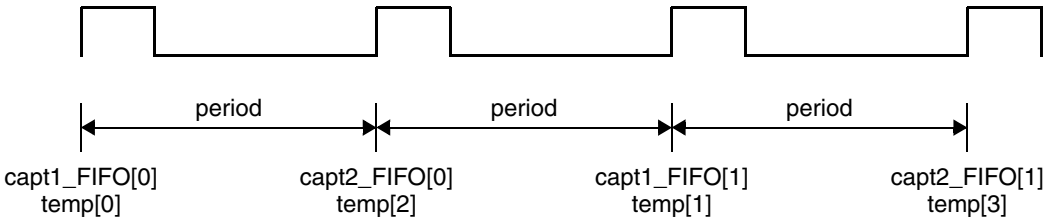


Figure 52. Timing of CAPT1 and CAPT2 values

```
CTU_SWTEST_TRIGGERTIME (period)

{
temp[4] = {0,0,0,0};                                    // MRS to start CTU (here just for
CTU.CTUCR.B.MRS_SG = 0x1;                               // illustration)

while(ETIMER_2.CHANNEL[0].CTRL3.B.C2FCNT<2);            // wait until two entries in FIFO 2

ETIMER_2.CHANNEL[0].CCCTRL.B.ARM = 0x0;                 // stop counting to read values

temp[0] = ETIMER_2.CHANNEL[0].CAPT1.R;                  // reading capture values from FIFO
temp[1] = ETIMER_2.CHANNEL[0].CAPT1.R;
temp[2] = ETIMER_2.CHANNEL[0].CAPT2.R;
temp[3] = ETIMER_2.CHANNEL[0].CAPT2.R;

ETIMER_2.CHANNEL[0].CCCTRL.B.ARM = 0x1;
if (
        ((temp[2]-temp[0])!=120*period) ||               // assuming fclock = 120 MHz
        ((temp[3]-temp[1])!=120*period) ||
        ((temp[1]-temp[2])!=120*period))
                return FAIL;
else return PASS;
}
```

Parameters:
period: expected time between two triggers. The value is in μs

Return Values:
PASS when difference fits into the tolerance
FAIL otherwise

Figure 53. Code example: CTU_SWTEST_TRIGGERTIME (trig.)

8.1.3.6 CTU_HWSWTEST_TRIGGERNUM

```
CTU_HWSWTEST_TRIGGERNUM_ISR() {

g_ctuefr = CTU.CTUEFR.R;                                // a critical event occurs it immediately attends
CTU.CTUEFR.R = 0xFFFF;                                  // store the error cause in a variable

if (g_ctuefr&(1<<11)); [place reaction                  // external trigger generation overrun error
here]                                                   // timer 4 trigger generation overrun error
if (g_ctuefr&(1<<9)); [place reaction here]             // timer 3 trigger generation overrun error
if (g_ctuefr&(1<<8)); [place reaction here]             // timer 2 trigger generation overrun error
if (g_ctuefr&(1<<7)); [place reaction here]             // timer 1 trigger generation overrun error
if (g_ctuefr&(1<<6)); [place reaction here]             // ADC command generation overrun error
if (g_ctuefr&(1<<5)); [place reaction here]             // TGS overrun error in sequential mode
if (g_ctuefr&(1<<4)); [place reaction here]             // Master Reload Signal overrun error
if (g_ctuefr&(1<<3)); [place reaction here]             // trigger overrun error (more than 8 EV) in TGS
if (g_ctuefr&(1<<1)); [place reaction here]             sequential mode
}
```

Figure 54. Code example: CTU_HWSWTEST_TRIGGERNUM

8.1.3.7 CTU_HW_CFGINTEGRITY

```
CTU_HW_CFGINTEGRITY (* store_var){

if ((*store_var & 0x1) || (*store_var & 0x8)) {    // if there was a reload error (MRS_RE)
         return (FAIL);                            // (0x1) or an overrun (MRS_O) (0x8) the
         }                                         // configuration is/could be incorrect
return (PASS);
}
```

Parameters
*store_var: When the CTUEFR register is read, it's content is cleared. The original content is stored in address give by store_var

Return Values:
PASS: no configuration error
FAIL: otherwise

Figure 55. Code example: CTU_HW_CFGINTEGRITY

8.1.4 Digital inputs

8.1.4.1 ETIMERI_SWTEST_CMP

This test is to redundantly read two PWM inputs. The pulse widths of the input signals are captured and compared for this test. However, other parameter of the input signal could be used if configured with a simple configuration change.

To demonstrate, this example uses hard-coded I/O and eTimer channels.

```
ETIMERI_SWTEST_CMP () {

                                                    // configure SIUL inputs
SIUL.PCR[0].R = 0x0500;                             // A[0] to eTimer_0
SIUL.PCR[4].R = 0x0500;                             // A[4] to eTimer_1

ETIMER_0.CHANNEL[0].CTRL1.R = 0x3800;               // eTimer0: configure period measurement
ETIMER_0.CHANNEL[0].CCCTRL.R = 0x0065;              // ARM, capture 1:rising, capture2:falling

ETIMER_1.CHANNEL[0].CTRL1.R = 0x3800;               // eTimer_1: configure period measurement
ETIMER_1.CHANNEL[0].CCCTRL.R = 0x0065;              // ARM, capture 1:rising, capture 2:falling

while(ETIMER_0.CHANNEL[0].STS.B.ICF1 == 0);         // eTimer_0: wait for 1st edge to capture
eT0_time1 = ETIMER_0.CHANNEL[0].CAPT1.R;            // eTimer_0: read time 1st edge captured at

while(ETIMER_1.CHANNEL[0].STS.B.ICF1 == 0);         // eTimer_1: wait for 1st edge to capture
eT1_time1 = ETIMER_1.CHANNEL[0].CAPT1.R;            // eTimer_1: read time 1st edge captured at

while(ETIMER_0.CHANNEL[0].STS.B.ICF2 == 0);         // eTimer_0: wait for 2nd edge to capture
eT0_time2 = ETIMER_0.CHANNEL[0].CAPT2.R;            // eTimer_0: read time 2nd edge captured at

while(ETIMER_1.CHANNEL[0].STS.B.ICF2 == 0);         // eTimer_1: wait for 2nd edge to capture
eT1_time2 = ETIMER_1.CHANNEL[0].CAPT2.R;            // eTimer_1: read time 2nd edge captured at

eT0_Result = eT0_time2 - eT0_time1;                 // eTimer_0: calculate Period
eT1_Result = eT1_time2 - eT1_time1;                 // eTimer_1: calculate Period

if (     (eT0_Result == eT1_Result)||               // Check if results are the same within one count
         (eT0_Result+1 == eT1_Result)||
         (eT0_Result-1 == eT1_Result)) return
eT1_Result;
else return error;
}
```

Parameters: none
Return values:
error: Inputs did not match
value: Pulse widths matched and the value of the matching pulse width is returned.

Figure 56. Code example: ETIMERI_SWTEST_CMP

8.1.4.2 GPI_SWTEST_CMP

```
GPI_SWTEST_CMP (pin1, pin2){

if(pin1== pin2) return read_error;                    // Make sure that I/Os are different

SIUL.PCR[pin1].R = 0x0100;                            // Configure SIUL GPIO as input
SIUL.PCR[pin2].R = 0x0100;

pin1_val = SIUL.GPDI[pin1].R;                         // Read Inputs
pin2_val = SIUL.GPDI[pin2].R;

if(pin1_val != pin2_val) return read_error;           // Check values match and return
else return pin1_val;
}
```

Parameters
pin1: Defines the first of the two input GPIO to be read
pin2: Defines the second GPIO to be read

Return values
0: Input value read as low on both Pin 1 and Pin 2
1: Input value read as high on both Pin 1 and Pin 2
read_error: Input values read on Pin1 and Pin2 do not match

Figure 57. Code example: GPI_SWTEST_CMP

8.1.5 Digital outputs

8.1.5.1 GPODW_SWAPP_WRITE

```
GPODW_SWAPP_WRITE (pin1, pin2, op_state){

pin1_reg = pin1 >> 5;                                   // Calculate number of SIUL.PGPDO
pin2_reg = pin2 >> 5;                                   // of first and second GPIO

if((pin1_reg != pin2_reg) | (pin1=pin2) {               // Both GPIOs different and
         return(error);                                 // programmed in
         }                                              // the same parallel data register?
SIUL.PCR[pin1].R = 0x0200;
SIUL.PCR[pin2].R = 0x0200;                              // Configure GPIO as output
                                                        // Configure GPIO as output
pin1_bit = pin1 - (pin1_reg * 32);
pin2_bit = pin2 - (pin1_reg * 32);                      // Calculate values to write to parallel
                                                        // data register
temp1 = 0x1 << pin1_bit;
temp2 = 0x1 << pin2_bit;                                // Create bit Position Masks
temp1 = temp1 + temp2;

temp1 = reverse(temp1);
                                                        // Reverse the bit Order
if (op_state == 0){
         SIUL.PGPDO[pin1_reg].R = (SIUL.PGPDO[pin1_reg].R
& !temp1);                                              // Clear Output
         if (SIUL.PGPDO[pin1_reg].R & temp1) return error // Readback required (atomic
         else return output_ok;                         operation)
         }
else if (op_state == 1){
         SIUL.PGPDO[pin1_reg].R = (SIUL.PGPDO[pin1_reg].R
| temp1);                                               // Set Output
         if (SIUL.PGPDO[pin1_reg].R & temp1) return     // Readback required (atomic
output_ok                                               operation)
         else return error;
         }
else return error;
}
```

Parameters:
pin0: The first GPIO number that output is to be generated on
pin1: The second GPIO number that output is to be generated on
op_state: Desired value of the GPIO

Return Values:
output_ok: Output written successfully
error: Output configuration does not read back as written: pads are not in the same parallel data out register.

Figure 58. Code example: GPODW_SWAPP_WRITE

8.1.5.2 GPOIRB_SWTEST_CMP

This software test executes the comparison between the desired output values and the value read back via internal read back configuration.

```
GPOIRB_SWTEST_CMP (pin, op_state){

SIUL.PCR[pin_op].R= 0x0300;                  // Configure Pin[pin_op] as GPIO output with enabled
                                             input buffer
SIUL.GPDO[pin_op].R = op_state;
                                             // Set Pin[pin_op] Output State
if(SIU.GPDI[pin_op].R != op_state)
return(error);                               // Does read back value matches the desired op_state?
else return(output_ok);
}
```

Parameters:
pin_op: The GPIO number that output is to be generated on and read back from.
op_state: Desired value of the GPIO

Return values:
output_ok: Input value read back matches written output
input_error: Output does not read back as written

Figure 59. Code example: GPOIRB_SWTEST_CMP

8.1.5.3 GPOERB_SWTEST_CMP

The output is externally connected to an input pin. After writing the output value to the pin, the input is read to check that the correct output value is present.

```
GPOERB_SWTEST_CMP (pin_op, pin_rb, op_state) {

SIUL.PCR[pin_op].R= 0x0200;                  // Configure Pin[pin_op] as GPIO output
SIUL.PCR[pin_rb].R= 0x0100;                  // Configure Pin[pin_rb] as GPIO input

SIUL.GPDO[pin_op].R = op_state;              // Set Pin[pin_op] Output State

[short pause]                                // There is a short propagation delay before the output is
                                             // present at the input of the read back pin.

if(SIU.GPDI[pin_rb].R != op_state)           // Does read back value match the desired out_val?
return(error);
else return(output_ok);
}
```

Parameters:
pin_op: The GPIO number that output is to be generated on.
pin_rb: The GPIO number that the output is to be read back on.
op_state: Desired value of the GPIO

Return values:
output_ok: Input value read back matches output written
error: Output does not read back as written

Figure 60. Code example: GPOERB_SWTEST_CMP

8.1.5.4 PWMRB_SWTEST_CMP

In this test case the pulse width is measured and used for comparison. The eTimer could be configured to compare the period by changing the configuration to capture the times at which 2 consecutive edges of the same type occur.

The result might be a few counts different from the output signal. It is good practice to take this into account on the system level.

The FlexPWM is configured to generate a center aligned PWM. This configuration is shown in Figure 61.

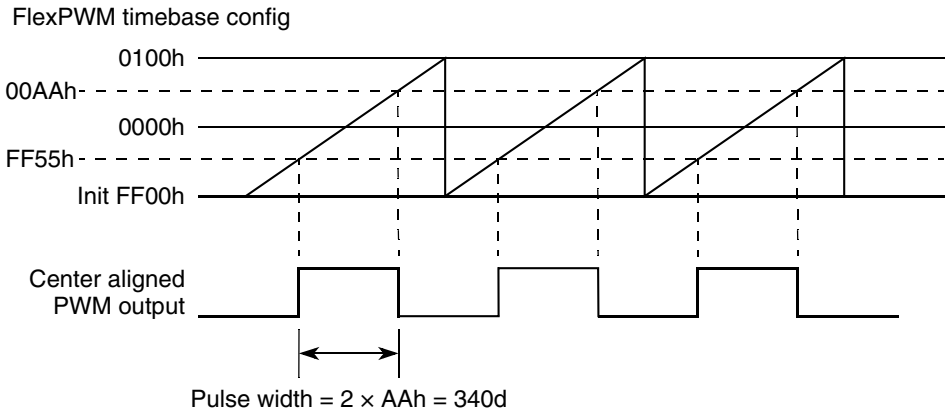


Figure 61. PWM output signal and Timing

Application software validates the values read by comparing these with expected values (340 (154h) in this example).

```
PWMRB_SWTEST_CMP (){

                                                    // configure SIUL inputs
SIUL.PCR[4].R = 0x0500;                             // input: A[4] to eTimer_1
SIUL.PCR[12].R = 0x0A00;                            // output: FlexPWM_0 A[2] to A[12]

                                                    // configure FlexPWM for center aligned PWM

FLEXPWM_0.SUB[2].INIT.R = 0xFF00;                   // initial count value
FLEXPWM_0.SUB[2].CTRL.B.FULL = 1;                   // full Cycle reload
FLEXPWM_0.SUB[2].CTRL2.B.INDEP = 0x1;               // independent Outputs
FLEXPWM_0.SUB[2].CTRL2.B.DBGEN = 0x1;               // PWM runs in debug
FLEXPWM_0.SUB[2].DTCNT0.R = 0x0000                  // PWM dead time 0
FLEXPWM_0.SUB[2].DTCNT1.R = 0x0000                  // PWM dead time 1
FLEXPWM_0.SUB[2].DISMAP.R = 0xF000;                 // reset Fault Dis
FLEXPWM_0.SUB[2].CTRL2.B.DBGEN = 1;                 // PWM runs in debug mode
FLEXPWM_0.SUB[2].VAL[0].R = 0x0000;                 // mid-cycle reload point
FLEXPWM_0.SUB[2].VAL[1].R = 0x0100;                 // max value for counter
FLEXPWM_0.SUB[2].VAL[2].R = 0xFF55;                 // PWMA high 0xFF55
FLEXPWM_0.SUB[2].VAL[3].R = 0x00AA;                 // PWMA low 0x00AA
FLEXPWM_0.OUTEN.B.PWMA_EN = 0x4;                    // enable PWM A - channel 2

FLEXPWM_0.MCTRL.B.LDOK = 0x4;                       // load configuration values into buffers
FLEXPWM_0.MCTRL.B.RUN = 0x4;                        // Go!

ETIMER_1.CHANNEL[0].CTRL1.R = 0x3800;               // configure period measurement
ETIMER_1.CHANNEL[0].CCCTRL.R = 0x0065;              // ARM, capture1:rising, capture 2:falling

while(ETIMER_1.CHANNEL[0].STS.B.ICF1 == 0);         // wait for 1st edge to capture
time1 = ETIMER_1.CHANNEL[0].CAPT1.R;                // read time 1st edge captured at

while(ETIMER_1.CHANNEL[0].STS.B.ICF2 == 0);         // wait for 2nd edge to capture
time2 = ETIMER_1.CHANNEL[0].CAPT2.R;                // read time 2nd edge captured at

Result = time2 - time1;                             // calculate Period

 return Result;
}
```

Return Values:
The time between the rising edge and falling edge.

Figure 62. Code example: PWMRB_SWTEST_CMP

8.1.5.5 PWMDW_SWAPP_WRITE

This example configures the FlexPWM and uses channel A of two of the FlexPWM modules. The code for PWM generation is next to output configuration, whereas in the real application, it is very likely that the code for PWM generation is kept separate. For demonstration, this example uses hard-coded I/O and

FlexPWM outputs, because a generic example considering the restrictions in mapping FlexPWM outputs to GPO would to go beyond the scope of this document.

```
PWMDW_SWAPP_WRITE (){

FLEXPWM_0.SUB[2].INIT.R = 0xFF00;                    // Initial count value
FLEXPWM_0.SUB[2].CTRL.B.FULL = 1;                    // Full Cycle reload
FLEXPWM_0.SUB[2].CTRL2.B.INDEP = 0x1;                // Independent Outputs
FLEXPWM_0.SUB[2].CTRL2.B.DBGEN = 0x1;                // PWM runs in debug
FLEXPWM_0.SUB[2].DTCNT0.R = 0x0000;                  // PWMA dead time
FLEXPWM_0.SUB[2].DTCNT1.R = 0x0000;                  // PWMB dead time
FLEXPWM_0.SUB[2].DISMAP.R = 0xF000;                  // Reset Fault Dis
FLEXPWM_0.SUB[2].CTRL2.B.DBGEN = 1;                  // PWM runs in debug mode
FLEXPWM_0.SUB[2].VAL[0].R = 0x0000;                  // Mid-Cycle Reload Point
FLEXPWM_0.SUB[2].VAL[1].R = 0x0100;                  // Max value for counter
FLEXPWM_0.SUB[2].VAL[2].R = 0xFF55;                  // Max value for counter
FLEXPWM_0.SUB[2].VAL[3].R = 0x00AA;                  // PWMA Low 0x00AA
FLEXPWM_0.SUB[2].VAL[4].R = 0xFFAA;                  // PWMB High 0xFFAA
FLEXPWM_0.SUB[2].VAL[5].R = 0x0055;                  // PWMB Low 0x0055

FLEXPWM_0.OUTEN.B.PWMA_EN = 0x4;                     // Enable PWM A

FLEXPWM_0.MCTRL.B.LDOK = 0x4;                        // Load configuration values into
FLEXPWM_0.MCTRL.B.RUN = 0x4;                         buffers
                                                     // Go!
FLEXPWM_1.SUB[0].INIT.R = 0xFF00;
FLEXPWM_1.SUB[0].CTRL.B.FULL = 1;                    // Initial count value
FLEXPWM_1.SUB[0].CTRL2.B.INDEP = 0x1;                // Full Cycle reload
FLEXPWM_1.SUB[0].CTRL2.B.DBGEN = 0x1;                // Independent Outputs
FLEXPWM_1.SUB[0].DTCNT0.R = 0x0000;                  // PWM runs in debug
FLEXPWM_1.SUB[0].DTCNT1.R = 0x0000;                  // PWMA dead time
FLEXPWM_1.SUB[0].DISMAP.R = 0xF000; /                // PWMB dead time
FLEXPWM_1.SUB[0].CTRL2.B.DBGEN = 1;                  // Reset Fault Dis
FLEXPWM_1.SUB[0].VAL[0].R = 0x0000;                  // PWM runs in debug mode
FLEXPWM_1.SUB[0].VAL[1].R = 0x0100;                  // Mid-Cycle Reload Point
FLEXPWM_1.SUB[0].VAL[2].R = 0xFF55;                  // Max value for counter
FLEXPWM_1.SUB[0].VAL[3].R = 0x00AA;                  // PWMA High 0xFF55
FLEXPWM_1.SUB[0].VAL[4].R = 0xFFAA;                  // PWMA Low 0x00AA
FLEXPWM_1.SUB[0].VAL[5].R = 0x0055;                  // PWMB High 0xFFAA
                                                     // PWMB Low 0x0055
FLEXPWM_1.OUTEN.B.PWMA_EN = 0x1;
                                                     // Enable PWM A
FLEXPWM_1.MCTRL.B.LDOK = 0x1;
FLEXPWM_1.MCTRL.B.RUN = 0x1;                         // Load configuration values into
                                                     buffers
                                                     // Go!

SIUL.PCR[12].R = 0x0A00;
SIUL.PCR[117].R = 0x0600;                            // configure SIUL outputs
}                                                    // FlexPWM_0 A[2] to A[12]
                                                     // FlexPWM_1 A[0] to H[5]
```

Figure 63. Code example: PWMDW_SWAPP_WRITE

8.1.6 Analog inputs

8.1.6.1 ADC_SWTEST_TEST1

Figure 64 shows the ADC_SWTEST_TEST1 configuration using ADC_0. Depending on the type of input signal to the ADC, $V_{DD\_HV\_ADR}$, $V_{SS\_HV\_ADR}$, or alternating between these, may be used for this function. This function may also be used for ADC_1, ADC_2, and ADC_3.

```
ADC0_SWTEST_TEST1_Init_ADC(){

ADC0.MCR.B.MODE = 0;                          // Set one shot mode
ADC0.NCMR0.B.CH0 = 1;                         // Enable channel 0
ADC0.NCMR0.B.CH1 = 1;                         // Enable channel 1
[... all other used channels]

SIUL.PCR[23].R=0x2100;                        // ADC_0: enable B[7] for AN[0]
SIUL.PCR[24].R=0x2100;                        // ADC_0: enable B[8] for AN[1]
[... all other used channels]

ADC0.PSCR.B.PREVAL0=0;                        // Select V_DD_HV_ADR0 as presampling voltage
ADC0.PSCR.B.PREVAL1=0;                        // Select V_DD_HV_ADR0 as presampling voltage (errata
ADC0.PSCR.B.PREVAL2=0;                        //#4016)
ADC0.PSCR.B.PRECONV=1;                        // Select V_DD_HV_ADR0 as presampling voltage (errata
ADC0.PSR0.R=0xFFFF;                           //#4016)
                                              // The ADC will perform a presampling followed by a
[VSS_HV_ADR0 can be used alternatively]       conversion
                                              // Presampling enabled for channels 0-15
ADC0.IMR.B.MSKECH=1;
}                                             // Application dependent

                                              // Configuration of End of Chain interrupt
```

Parameters:
none

Return Values:
none

Figure 64. Code example: ADC_SWTEST_TEST1

Figure 65 shows the execution of ADC_SWTEST_TEST1. This function is called when an end of chain interrupt occurs on ADC_0. This function tests whether the converted value is near to the value of a converted reference value. If the converted value is very close to the value of a converted reference voltage (for example, FFFh or 0), an open failure, or misconfiguration, of the multiplexing circuit is assumed. In this case, g_ADC0_SWTEST_TEST1_result = 1, which indicates a failed test. In this example, all channels are tested. If fewer channels are used in an application, the unused channels must be excluded from testing.

```
ADC0_SWTEST_TEST1_handler(){

ADC0.ISR.B.ECH=1;                                                    //if the difference between
if(ADC0.PSCR.B.PREVAL0==0){                                          presampling value and measured
   for(i=0;i<16;i++){                                                value is lower than the tolerance
      if (abs(0xFFF - ADC0.CDR[i].B.CDATA) <= D_TOLERANCE)
      g_ADC0_SWTEST_TEST1_result=1;                                  //if the difference between
   }                                                                 presampling value and measured
}                                                                    value is lower than the tolerance

if(ADC0.PSCR.B.PREVAL0==1){
   for(i=0;i<16;i++){
      if (abs(0 - ADC0.CDR[i].B.CDATA) <= D_TOLERANCE)
         g_ADC0_SWTEST_TEST1_result=1;
   }
}
}
```

Figure 65. Code example: ADC_SWTEST_TEST1

8.1.6.2 ADC_SWTEST_TEST2

Figure 66 shows the configuration of ADC_0 for ADC_SWTEST_TEST2. This function may also be used for ADC_1, ADC_2, and ADC_3 as needed.

```
Init_ADC(){

ADC0.MCR.B.MODE = 0;                  // Set one shot mode
ADC0.NCMR0.B.CH2 = 1;                 // Enable channels to be used, channels 2 and 5 in this example
ADC0.NCMR0.B.CH5 = 1;

ADC0.PSCR.B.PREVAL0=0;                // Select V_DD_HV_ADR0 as presampling voltage
ADC0.PSCR.B.PREVAL1=0;                // Select V_DD_HV_ADR0 as presampling voltage
ADC0.PSCR.B.PREVAL2=0;                // Select V_DD_HV_ADR0 as presampling voltage
ADC0.PSCR.B.PRECONV=1;                // The ADC will perform a presampling followed by a conversion
ADC0.PSR0.R=0x24;                     // Presampling enabled for channels 2 and 5

ADC0.IMR.B.MSKECH=1;                  // Configuration of End of Chain interrupt
}
```

Figure 66. Code example: ADC_SWTEST_TEST2

Figure 67 shows the execution of ADC_SWTEST_TEST2. This function is called when an end of chain interrupt occurs on ADC_0. This function tests whether the difference of the converted values for both reference voltages are within an expected range (D_TOLERANCE). ADC0_SWTEST_TEST2 = 1 if the test fails.

```
ADC0_SWTEST_TEST2_handler(){

ADC0.ISR.B.ECH=1;
if(ADC0.PSCR.B.PREVAL0==0) {
   g_ADC0_channel2=ADC0.CDR[2].B.CDATA;
   ADC0.PSCR.B.PREVAL0=1;                          //Select V_SS_HV_ADR0 as presampling voltage
   ADC0.PSCR.B.PREVAL1=1;                          //Select V_SS_HV_ADR0 as presampling voltage
   ADC0.PSCR.B.PREVAL2=1;                          //Select V_SS_HV_ADR0 as presampling voltage
   ADC0.MCR.B.NSTART=1;                            //start the chain conversion.
}
if(ADC0.PSCR.B.PREVAL0==1){
   if (abs(0xFFF - g_ADC0_channel2 + ADC0.CDR[5].B.CDATA)
   >= D_TOLERANCE)
      g_ADC0_SWTEST_TEST2_result=1;

   ADC0.PSCR.B.PREVAL0=0;                          // select V_DD_HV_ADR0 as presampling voltage
   ADC0.PSCR.B.PREVAL1=0;                          // select V_DD_HV_ADR0 as presampling voltage
   ADC0.PSCR.B.PREVAL2=0;                          // select V_DD_HV_ADR0 as presampling voltage
}
```

Figure 67. Code example: ADC_SWTEST_TEST2

8.1.6.3 ADC_SWTEST_CMP

```
ADC_SWTEST_CMP (tol, ADCH1, ADCH2){

ADC0.MCR.B.NSTART = 1;                             // start a single conversion with ADC_0
ADC1.MCR.B.NSTART = 1;                             // start a single conversion with ADC_1

while (
    (!ADC0.CDR[ADCH1].B.VALID) ||
    (!ADC1.CDR[ADCH2].B.VALID))
       {;}                                         // wait till conversion is complete

if (abs(ADC0.CDR[ADCH1].B.CDATA -
    ADC1.CDR[ADCH2].B.CDATA) >= tol) return (FAIL);   // difference between the results is
    return (PASS);                                 // greater than the tolerance
}
```

Parameters:
tol: Defines the allowed difference from the reference
ADCH1: first ADC channel
ADCH2: second ADC channel

Return Values:
PASS: results match within the given tolerance
FAIL: results do not match

Figure 68. Code example: ADC_SWTEST_CMP

8.2 Checks and configurations

Below is a list of the minimum number of checks by the safety integrity functions which need to pass before executing any safety function:

- Lock-step mode check

- STCU check
- Flash Array Integrity Self check
- SUPPLY SELF-TEST
- Temperature sensor check
- SWT enabled
- CMU check
- PLL_SW_CHECK
- IRC_SW_CHECK
- PMC check
- FCCU_F[*n*] signal check[1]

Correct execution of these checks is a prerequisite for functional safety.

Below lists all checks that are repeated (at least once every FTTI) during normal operation of the device:

- FLASH_SW_ECCTEST
- FLASH_SW_ECCREPORT
- Temperature check
- CRC calculation[2]

1. Required for single FCCU signal usage only
2. Safety requirement once per FTTI for single read functions only / Safety requirement once after programming for all other functions

9 Acronyms and abbreviations

A short list of acronyms and abbreviations used in this document is summarized for completeness:

Table 16. Acronyms and abbreviations

Terms	Meanings
ADC	Analog-to-Digital Converter
BAM	Boot Assist Module
BIST	Built-in-Self Test
BIU	Bus Interface Unit
CF	Critical Fault
CCF	Common Cause Failure
CMF	Common Mode Failure
CMU	Clock Monitor Unit
CRC	Cyclic Redundancy Check
CTU	Cross-Triggering Unit
DC	Diagnostic Coverage
DMA	Direct Memory Access
DED	Dual Error Detection
ECC	Error Correcting Code
ECSM	Error Correction Status Module
ECU	Electronic Control Unit
eDMA	Enhanced Direct Memory Access
ERRM	Error Out Monitor function
EXWD	External Timeout (Watchdog) function
FCCU	Fault Collection and Control Unit
FMEDA	Failure Modes, Effects & Diagnostic Analysis
FMPLL	Frequency-Modulated Phase-Locked Loop
FTTI	Single-Point Fault Tolerant Time Interval
GPIO	General Purpose I/O
HVD	High Voltage Detector
INTC	Interrupt Controller
LBIST	Logic Built-In-Self-Test
LOC	Loss-of-clock
LOL	Loss-of-lock
LVD	Low Voltage Detector

Table 16. Acronyms and abbreviations (continued)

Terms	Meanings
MBIST	Memory-Built-In-Self-Test
MC_CGM	Clock Generation Module
MC_ME	Mode Entry
MC_RGM	Reset Generation Module
MCU	Microcontroller Unit
MMU	Memory Management Unit
MPU	Memory Protection Unit
NCF	Non-Critical Fault
NMI	Non-Maskable Interrupt
NVM	Non-Volatile Memory
PMC	Power Management Controller
PSM	Power Supply and Monitor function
PWM	Pulse Width Modulation
RCCU	Redundancy Control and Checking Unit
SEC	Single Error Correction
SFF	Safe Failure Fraction
SIL	Safety Integrity Level
SM	Safety Manual
SoR	Sphere of Replication
SSCM	System Status and Configuration Module
STCU	Self-Test Control Unit
SWG	Sine Wave Generator
SWT	Software Watchdog Timer
XOSC	External Oscillator

10 Document revision history

Table 17 summarizes revisions to this document.

Table 17. Revision history

Revision	Date	Description of Changes
1	7 Dec 2012	• Initial document release

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

MPC5675KSM
Rev. 1
12/2012