Freescale Semiconductor
Application Note

Document Number:AN4527
Rev. 0, 07/2012

Software Routines for Functional Safety Usage of the Qorivva Interrupt Controllers

Single, Dual, and Multicore Devices

by: Inga Harris and Markus Baumeister

Contents

1 Introduction

This application note discusses software measures for the use of the Interrupt Controller (INTC) in selected MPC57xx Qorivva Microcontroller Unit (MCU) Multicore devices in safety-relevant applications. Since INTC is not protected by special hardware measures on these devices, potential safety relevant failures need to be detected in software.

This application note explains different INTC software measures with their purposes and also some examples for implementation of these measures. At the end there is an overview that shows the different measures together.

The automotive industry is under pressure to provide new and improved vehicle safety systems, ranging from basic airbag-deployment systems to extremely complex advanced driver assistance systems (ADAS) with accident prediction and avoidance capabilities. These safety functions are increasingly carried out by electronics, and the ISO 26262 standard has been developed to guide the design of electronic systems that can prevent dangerous failures or control them if they occur. Freescale's SafeAssure program is aligned with the international standards at the heart of automotive safety applications to allow system engineers to design with confidence and efficiently achieve system-level compliance.

Functional safety is achieved when there is absence of unreasonable risk due to hazards caused by the malfunctioning of electrical/electronic systems. Industries impose functional safety standards as a way to ensure that safety-related systems will offer the necessary risk reduction required to achieve safety for the equipment. The functional safety standards include IEC 61508, which covers functional safety for the general industry, and ISO 26262, which covers functional safety for road vehicles.

Freescale's SafeAssure program provides system designers with easier identification of hardware and software designed to target functional safety applications. SafeAssure products are conceived to simplify system-level functional safety design and standard compliance and come with a rich set of enablement collateral facilitating failure analysis and hardware and software integration. Moreover, SafeAssure products provide a clear support interface to ensure that we are there to serve your needs at each step of the system design and compliance process.

As part of SafeAssure, this application note provides guidance to how INTCs on Qorriva MCUs can be used by safety-relevant applications.

2 Document conventions

This application note discusses measures for the use of select MPC57xx Qorivva Microcontroller Unit (MCU) Multicore devices Interrupt Controller (INTC) in safety-relevant applications. It describes several measures that were assumed to be in place when analyzing the safety of the MPC5746M, MPC5744K, and MPC5777M in their Failure Modes, Effects and Diagnostics Analyses (FMEDAs). These measures are thus the recommended way to use the INTC in safety-relevant, Automotive Safety Integrity Level D (ASIL D), applications on these devices. In the future, other MPC57xxx devices may refer to this application note, and the described procedures can also be used on other devices after sufficient adaptation.

This document also contains guidelines on how to configure and operate the mentioned MCUs for ASIL D applications. These guidelines are preceded by one of the following bold text statements:

Implementation: Verbal description of the expected implementation

Recommended: Additional measures possible

Failure modes: Failure modes diagnosed (at least partially) by this measure

Example: Pseudo code examples

Generic examples are given in pseudo code. Example on how this generic version can be optimized for special cases are given in a pseudo assembler language.

Execution times for the functions and instructions are based on a C-code implementation of the pseudo code example and were calculated using MPC5644A at 150 MHz with Greenhills Compiler 2012 and the timings are for executions with no error events.

All the function's characteristics are summarized in the section Summary.

3 Initialization

This section describes measures to apply during initialization of the MCU. Preferably they should be executed before the INTC is activated. They are assumed to be in place before the safety-relevant application has completed initialization and taken operational control.

3.1 Unused interrupts

A functional safety application should have all unused interrupt vectors set up to point/jump to an address that is illegal or contains an illegal instruction which would cause the core to raise an exception. This function must be executed inside every unused Interrupt Service Routine (ISR) vector. This mechanism helps to detect wrong ISR execution. Not enabling a way to catch an unused interrupt jump could lead to code runaway or general erroneous operation.

Implementation: A “dummy” ISR is used in all unused ISR instances that jumps to an illegal instruction or accesses an illegal address causing a core exception. This is configured during initialization.

Failure modes: Spurious or wrong IRQ calls an unused IRQ. This can only happen due to misconfiguration or failures in the INTC itself.

Generic Example: INTC_UNUSED_ILLEGAL

```
Initialize a pointer to an illegal address
Write a value to that pointer
```

The execution time is nearly instantaneous, around four instructions depending on the compiler optimization.

Assembler Example:

```
ILLEGAL                # illegal instruction in the interrupt vector table
```

4 Run-time INTC checks

This section describes measures that can be taken to achieve functional safety of the INTC module during operation. It assumes that initial checks and configurations are already safe, in other words, after startup, the application software has already ensured that a safe condition is satisfied before the INTC safety-relevant functions described in the document are enabled. These are described in the individual device's safety manual, if available.

After starting safety-relevant operations, it is assumed that the application software will check the conditions described in this section. Some conditions require periodic checking while others require checking only when needed by the application.

A description of the failure mode and its effect is provided for each function.

4.1 Events, flag, and triggers

The INTC functionality depends upon the interrupt events being correctly and timely presented to and executed by the INTC module. This can be assessed by SW and raised as an issue to the main core if some hardware operation is not responding as expected. There are four types of hardware operation that can be monitored:

- Events such as buffer overflow and communication errors
- Flags that are set inside the module registers to indicate an event regardless of the interrupt being enabled or not
- Triggers which are signals sent to the INTC if the event occurs and the interrupt is enabled
- Spurious interrupt with no legal event

Failure of these operations is classified as an interrupt event not causing an interrupt, a serviced interrupt remaining flagged or the related event not responding and an event triggering the wrong interrupt routine.

4.1.1 Nonexecuted interrupt flags and events

Interrupt service routines could fail to be called due to failures in the INTC or signaling lines. To detect this, a functional safety application should periodically check that interrupt flags in peripherals with enabled IRQs are all clear. For this check to be effective, its execution must not prevent flags from being cleared. This check should be run periodically within the required FTTI which is typically between 10 ms and 200 ms. The system OS, scheduler, or an on-chip watchdog timer could be used to provide this frequency.

Implementation: The check should be executed when IRQs are enabled and outside of an ISR or within an ISR with low priority. If executed in an ISR, only the flags of interrupts with a higher priority can be tested. The check, triggered by the OS/scheduler/timer, scans different memory addresses for flags which should not be set if an appropriate ISR is executed. If a flag is not clear, the error handler needs to be notified[1] . There is a certain latency between an IRQ flag switching to on and the INTC activating the ISR in the core. For that reason, a not cleared flag needs to be validated by rechecking after the known INTC latency. If the number of executions of an ISE is important to the application's operation, this ISR must be executed often enough to detect a single missing IRQ.

Recommended: The ISR routine itself could also check that its IRQ flag is clear before leaving the ISR as this eases detection of SW faults and stuck-on IRQ flags.

Failure modes: Wrong or No ISR executed for an IRQ flag

Generic Example: ISR_CHECK_TRIGGER_CLEAR

```
Initialize the array of register addresses that are to be checked (only possible for higher
priority IRQs)
Initialize the array of bit masks which are to be used with the address array
Initialize an array of counter variables

For each element of the array
    Bitwise AND the register contents with the bit mask
    If the result is non zero
        Wait for the latency of the INTC
        Repeat the check
        If the result is still nonzero
                Increment the error counter variable
    If the error counter is nonzero or above a value determined by the application, flag an
issue to the error handling function
    Then clear the counter element
```

The execution time depends on the number of ISRs used in application. Each register address check will take around 20 instructions depending on the compiler optimization and this equates to a cycle count around 50 in the case one register needs to be checked, 370 for eight registers. This time can be reduced if the loop is manually "unrolled" and static addresses are used.

A possible workaround for a stuck-on IRQ flag is to disable the IRQ (preventing constant IRQs) and triggering the IRQ if necessary by enabling and then disabling the respective IRQ. In many MPC57xx implementations the interrupts are edge triggered and as such if an interrupt flag is stuck-on and is not cleared by the ISR, the ISR will not be recalled after the initial ISR is executed regardless of further events. If this situation is discovered, the ISR could be forced again by disabling and then re-enabling the enable or mask bits as described in the Interrupt Request Sources section of the reference manual, but this does not solve the issue of the ISR because it can no longer be triggered by the hardware event.

4.1.2 Execution without a trigger

If the INTC experiences a fault, for example an internal bit flip, it could request the core to execute an ISR without an actual request being available. A functional safety application should check that each[2] ISR is triggered by a valid trigger such as a Timer Overflow Interrupt in the Timer Overflow ISR. This function should be executed inside every ISR routine. If a valid trigger is not found, this is a fault condition.

Implementation: Before any trigger event is serviced inside the ISR, the function must check that a valid event and flag are observed. If a valid event or flag is not seen, an error is indicated.

Recommended: For software-triggered IRQs, that is, IRQ triggered by writes to INTC_SSCIRn, it is recommended to check an additional flag in RAM, set by the triggereing function and reset by the ISR, to detect spurious triggering by out-of-control software.

Failure modes: Spurious or constant interrupt signal or triggering the wrong interrupt vector .

Generic Example: ISR_CHECK_TRIGGER_SET

1. This notification should not happen via an interrupt because the INTC could have failed to propagate IRQs to ISRs.
2. This is assumed to be done for all ISRs and not only safety-relevant ISRs because the “wrong ISR called” failure mode is visible only in the wrongly called ISR that might not be safety-relevant itself but not in the uncalled ISR, which might be the safety-relevant one.

```
Initialize variable valid trigger

Calculate a value for valid trigger based on appropriate register contents
If the value is not as expected, flag an issue to the error handling function
```

Each register address check will take around five instructions depending on the compiler optimization and this equates to a cycle count around 15.

Assembler Example:

```
LOAD  R0, DSPI0_SR         # using DSPI Status register as an example
AND_I R0, 0x01             # Assuming only “Transfer Complete”
                           # is a valid IRQ source
BEQ       _ERRORHANDLER
```

If several different interrupt sources are handled by the same ISR, this has to be taken into account when checking for the set trigger bits.

4.1.3 Checking interrupt effects

Instead of checking IRQ flags for some IRQs it is also possible to directly check for functional effects of wrong IRQ processing. For this a functional safety application can periodically check for effects such as buffer over/underflow and communication errors. This check should be run periodically within the required fault tolerant time interval (FTTI), which is typically between 10 ms and 200 ms. The system OS, scheduler, or an on-chip watchdog timer could be used to provide this frequency. This check can often replace the inclusion of ISR_CHECK_TRIGGER_SET/CLEAR if all effects of spurious/lost IRQs can be detected.

NOTE

High volumes of interrupts can mean that lower priority interrupts can see a long lag from event to ISR execution to the extent that some interrupts could be missed, that is, a second interrupt event can occur before the first one is serviced. It could also be the case that an interrupt request was erroneously acknowledged. A check of flags could signal an error in such situations even when no dangerous effects are caused. A correctly implemented check for effect is more probable to identify only dangerous IRQ processing failures.

Implementation: The OS/Scheduler/Timer scans different memory addresses for events/conditions which should not exist if an appropriate ISR has been executed. In cases where the interrupt load is not high this function could run when all ISRs are serviced or run as a very low-priority ISR, but in this case it must be taken into account that lower/same priority ISRs might still require execution. If an event exists without a flagged IRQ, an error should be signaled, for example using the FCCU. Alternatively, notifying the Software Watchdog timer (SWT) managing routine could be flagged to not trigger the SWT causing a reset. It is recommended not to use IRQs to signal the error as an incorrectly working INTC can be the cause.

Failure modes: Several failure modes can be detected (Late, Lost, Spurious, Wrong ISR executions) but this highly depends on the application and the detectable effects.

Example: ISR_CHECK_EFFECT (highly application dependent)

```
Initialize the array of register addresses that are to be checked
Initialize the array of bit masks that are to be used with the address array

For each element of the array
Bitwise and the register contents with the bit mask
If the result is nonzero flag an issue to the error handling function
```

The execution time depends on the number of ISRs used in application. Each register address checked will take around 10 instructions depending on the compiler optimization and this equates to a cycle count around 25 in the case one register needs to be checked, 150 for eight registers.

4.1.3.1 Periodic interrupts

A special kind of effect is the periodic execution of an ISR. If the period is fixed and the expected jitter, for example, because of higher priority interrupts, is known, ISR_CHECK_EFFECT can check for the correct periodicity of the ISR execution by checking against a free running timer every time the ISR is called, and then store the current value of the timer and compare it against the stored value from the last ISR invocation. This can detect the failure modes of spurious and late ISR execution.

In general, ISR_CHECK_EFFECT implementation depends significantly on the interrupt source and the event signaled by the interrupt. It is not possible to list all varieties.

4.2 Interrupt characteristics checks

The interrupt execution depends upon the interrupt characteristics being correctly presented to and evaluated in the INTC module(s). This can be assessed by SW and raised as an issue. There are two interrupt characteristics that can be monitored:

- Actual interrupt priority versus expected priorities and pending interrupts
- Multicore characteristic checking by using the core assignment functionality

Failure of these operations is classified as an interrupt not being executed in a timely manner, increasing latency between an event and reaction or a spurious interrupt to one or more cores with no legal cause.

4.2.1 Interrupt priority

A functional safety application should check that an ISR is called with the correct priority assignment. This function must be executed inside every ISR routine[3]. If the priority is not as expected, this is a fault condition. If this fault occurs, the result is that high priority interrupts could be delayed beyond the system's specifications or that low-priority IRQs are not executed at all. It can also be an indication of a wrong ISR being executed.

Implementation: Before any trigger event is serviced inside the ISR, the function must check that the interrupt priority is as expected by comparing it against a priority table fixed in memory (implementation if the application allows priorities to be legally changed) or in the function itself (if the priority cannot be legally changed in the application). While using the priority table implementation, this table must be initialized by the CPU SW when it is configuring the interrupt priorities during system initialization. Special care has to be taken when IRQ priority escalation is conducted by the OS. When the priority does not match, an error indication is generated.

Recommended: The function could also have the ability to check that no higher priority interrupts are queued for service.

Failure Modes: Late, spurious or wrong ISR execution; execution with wrong priority

Generic Example: ISR_CHECK_PRIORITY

```
Initialize the array of all IVOR4 vector priorities

In the ISR, compare the INTC_CPR value with vector number element of the array
If they are not equal, flag an issue to the error handling function
```

The execution will take around seven instructions depending on the compiler optimization and this equates to a cycle count around eight. If the priority of an ISR is fixed, even shorter tests are possible such as shown in the following example.

Assembler Example:

```
LOAD    R0,  INTC_CPR0              # current priority register for core 0
CMP_I   R0, 0x10                    # Assuming fixed priority of 16
BNEQ    _ERRORHANDLER
```

3. This is assumed to be done for all ISRs and not only safety-relevant ISRs because the “late ISR” failure mode caused by a wrong INTC priority could be visible only in the low-priority ISR erroneously executed under high priority, which might not be safety-relevant.

4.2.2 Correct core assignment

A multicore functional safety application should check that an ISR is executed on the expected core. This function must be executed inside ISR routines[4]. If an ISR is not executed by the correct core this a fault condition and it may have system-level implications.

Implementation: Each ISR must have a target core specified. As with the interrupt priority function, if the target core is variable, a core assignment table must be set up and this must be managed by SW when configuring the target cores. Preferably the check for the correct core should be executed at the start of the ISR but if latency is critical it is also possible to execute the test later in the ISR if failure detection and reaction time is still shorter than the FTTI. If the executing core is not the expected core, an error indication is generated.

Failure Modes: Wrong core ISR execution

Generic Example: ISR_CHECK_CORE

```
Initialize an array of all IVOR4 core assignments

Compare the array element for that ISR with the core value running it
If they do not match, flag an issue to the error handling function that the ISR was
initiated on the wrong core
```

The execution time will take less than 10 instructions depending on the compiler optimization and this equates to a cycle count around 10.

Assembler Example:

```
LOAD    R0,CORE_PIR          # This actually needs an MFSPR instruction
CMP_I   R0,0x01              # Assuming to be executed on core 1 only
BNEQ    _ERRORHANDLER
```

4.3 Dual/Multicore interrupt checks

Some dual and multicore MCUs have multiple INTC modules, one per core. This allows the system operation to perform an INTC independent check to test for the correct operation of an INTC module. This is achieved by assigning an IRQ to two cores but one ISR is purely checking operation of the other.

4.3.1 Secondary monitoring core

A multicore functional safety application can check that an ISR is executed in one core by another core. This function must be executed inside ISR routines. This is recommended to implement if low latency IRQs exist where already short additional delays can violate the safety goal. The other measures described above cannot diagnose such additional short delays with sufficient coverage. If this monitoring detects an ISR not being executed timely or is executed incorrectly, then this is a fault condition.

Implementation: A shared access status flag is used to indicate the execution of an interrupt by the execution core. The executing core sets the flag at the start and clears it at end of the ISR. The monitoring core is also triggered by the interrupt event but instead of executing any functionality it checks that the flag is set in an appropriate time frame. If execution time of

4. This is assumed to be done for all ISRs and not only safety-relevant ISRs because the execution of an ISR on the wrong core can not only slow down the ISR (due to code and/or data not being in local memory) but also increases load on the wrong core potentially delaying the execution of safety-relevant functions. If such effects are expected to be diagnosed by other measures, for example, a watchdog, inclusion into safety-relevant ISRs is sufficient.

the ISR is also safety-relevant, another check for clearing the flag should be scheduled or the primary ISR could store a timestamp at the end of its execution in shared memory. If the monitoring core is not satisfied with triggering time or execution time, an error indication is generated. This function assumes that the ISR core assignment is fixed.

Failure Modes: All INTC failure modes

Example: ISR_DOUBLE_ASSIGNMENT

```
Initialize the shared access status flag array

Executing Core:
At start of ISR increment the variable which will mean the variable is always odd
At the end of the ISR increment the variable which will mean the variable is always even.
Optionally store the value of a free running timer in shared memory.

Monitoring Core:
Triggered by same IRQ:
Optionally store the value of the free running timer
Calculate the modulus2 of the shared variable
If the number is even and the IRQ trigger flag is clear, exit[5]
If the number is even and the event is still set, flag an issue to the error handling
function that the ISR was not taken timely

Optionally at a later time when the ISR is expected to have finished:
Check that the number is even.
If not, the primary ISR has not yet finished which will typically be an error.
Optionally compare the timestamp stored at the end of the primary ISR against the timestamp
stored locally.
If difference is too large, the ISR did not finish executing in time, which will typically
constitute an error.
```

The above implementation assumes the checked ISR has the highest IRQ priority on its core and is not delayed by disabled IRQs. For low-latency IRQs this should normally be the case. If not, additional instructions are necessary to determine whether the time between IRQ triggering and ISR execution is in line with latency requirements.

The execution time for the code run by the executing core will take around 10 instructions depending on the compiler optimization and this equates to a cycle count around 20.

The execution time for the code run by the monitoring core will take around five instructions depending on the compiler optimization and this equates to a cycle count around 10 if run inside an ISR. If this function is executed as a separate monitoring function, called periodically by a system timer, the monitoring core will take around 40 cycles to execute the function depending on the compiler optimization and over 200 cycles for eight registers.

5. This is only necessary if there is a possibility that the executing core finishes the ISR before the monitoring core can sufficiently execute the monitoring function. Note that such a possibility seriously hampers the ability of the monitoring core to ensure a low latency of the “execution” ISR.

5 Summary

The following table summarizes each of the previously discussed functions that should be incorporated in a functional safety application to insure robust system operation. The function type, where and how it should be executed, failure mode, and how long a typical error free routine would take to run are listed.

Function[1]	Type	Execute for which ISRs	Periodic	Int driven	Failure mode	Locations	Cycles (N=1)[2]	Cycles (N=8)[2]
ISR_CHECK_EFFECT	Event/ Flag/ Trigger	(SR)[3]	X	—	(Late), Spurious, Lost, Wrong Interrupts	Scheduled Function	23	150
ISR_CHECK_TRIGGER_CLEAR	Event/ Flag/ Trigger	SR	X	—	Lost, Wrong Interrupts	Scheduled function	53	369
ISR_CHECK_TRIGGER_SET	Event/ Flag/ Trigger	All	—	X	Spurious, Wrong, (Constant) Interrupt	ISR	16	16
INTC_UNUSED_ILLEGAL	Event/ Flag/ Trigger	Unused	—	—	Wrong, Spurious Interrupt	ISR	0	0
ISR_CHECK_PRIORITY	Interrupt Characteristics	All	—	X	Increased Interrupt Latency, Wrong Interrupt	ISR	7	7
ISR_DOUBLE_ASSIGNMENT (EXECUTING)	Interrupt Characteristics	(LL-SR)[4]	—	X	All INTC failures except misconfiguration	ISR on safety core	19	19
ISR_DOUBLE_ASSIGNMENT (MONITORING IN ISR)	Interrupt Characteristics	(LL-SR)[4]	—	X	All INTC failures except misconfiguration	ISR on other core	9	9
ISR_CHECK_CORE	Interrupt Characteristics	All	—	X	Wrong Core execution	ISR	9	9

1. Executed on e200z4 on MPC5644A @ 150 MHz with Greenhills Compiler 2012
2. N = number of event or flag registers to be checked
3. If possible as an alternative to ISR_CHECK_TRIGGER_SET/CLEAR
4. SR=Safety-relevant; LL-SR=Low latency, safety-relevant

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

Document Number: AN4527
Rev. 0, 07/2012