AN4344
Application note

Software routine for safe usage the interrupt controller SPC57EM80x/SPC574K72x according to ISO26262

Introduction

Aim of this document is to give some examples on implementing the needed software mechanisms to verify the integrity of the interrupt controller embedded in SPC57EM80x/SPC574K72x.

These safety mechanisms are listed in the Safety Manual and FMEDA of SPC57EM80x/SPC574K72x.

This document does not provide a list of all possible implementations of these safety mechanisms because the implementation of some of them is application dependent. The document helps the user to choose the methodology which better fits user's needs.

Contents

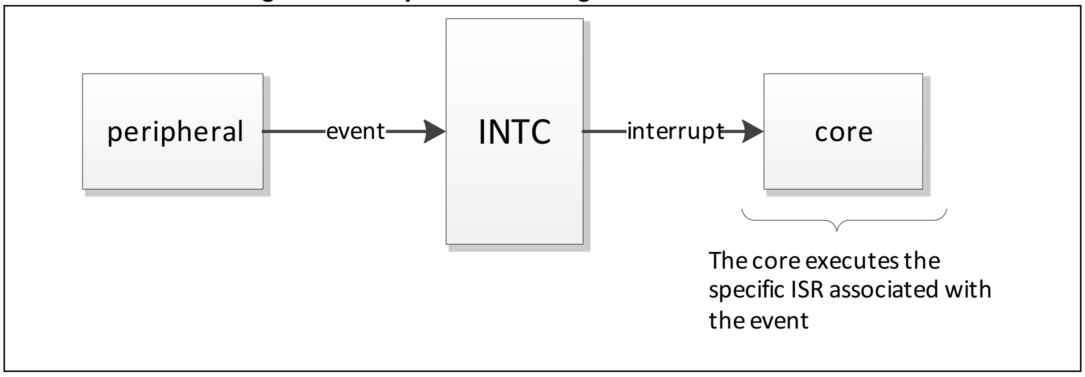
List of figures

1 Overview

SPC57EM80x/SPC574K72x is a multicore device which embeds a single instantiation of the Interrupt Controller (INTC).

The function of the INTC is to forward correctly and timely[a] interrupt events triggered by peripherals to the one or more cores.

Figure 1. Simplified working schema of the INTC


peripheral
event
INTC
interrupt
core
The core executes the specific ISR associated with the event

Interrupt controller is prone to different failure modes, for example:

- spurious interrupt
- lost interrupt
- interrupt serviced with wrong priority
- interrupt request forwarded to the wrong core
- interrupt latency higher than expected
- and so on

These failure modes can be detected by some software mechanisms. In case the software detects an unexpected situation a fault management routine shall be called. The reaction of this routine is application dependent and it’s out of the scope of the document.

The above failure modes can be caused, for instance, by a permanent damage in the silicon substrate (hard error according to the ISO26262) or by a particle strike due to an electromagnetic wave (soft error according to the ISO26262) which causes a temporal change of the MCU state.

To guarantee an acceptable risk according to the ISO26262, some mechanisms must be introduced to detect, and then react, to these failure modes. These mechanisms can be implemented either by software or hardware.

The full list of the considered failure modes, the mechanisms to detect them and their estimated diagnostic coverage can be found in the SPC57EM80x/SPC574K72x FMEDA.

SPC57EM80x/SPC574K72x is not the first MCU which has been thought to meet the ISO26262 requirements. The SPC56EL60 is the first MCU developed by

a. Timely means with the expected latency; correctly means with proper computation parameters as priority and location of the ISR.

STMicroelectronics which has been certified to meet requirements providing “Systematic Integrity: ASIL D”.

SPC56EL60 implements most of safety mechanisms for the INTC by hardware. It exploits the hardware redundancy to guarantee the integrity of the INTC. Two INTCs are embedded and, simplifying the real implementation, the output of the INTC_0 is compared with the output of INTC_1.

Such a solution still requires some software safety mechanisms to detect:

- spurious interrupt signaled to both INTCs for example due to a glitches on the line source of the interrupt, or
- lost interrupt due to an unwanted masking to both INTCs.

As drawback this hardware redundancy requires a larger silicon area and a higher consumption.

This solution has been optimized on SPC57EM80x/SPC574K72x which embeds a single INTC and bases the verification of the INTC integrity mainly on a software implementation.

In addition to the software mechanisms there is a hardware mechanism, i.e. LBIST, to detect latent failures affecting the INTC.

Solution showed in the document shall be considered not as mandatory, but recommended and taken as an example on the implementation of the safety mechanisms.

This document focuses on the detection of the occurrence of a failure. Since the best reaction is application dependent, it’s out of the scope of this document.

In the described examples, each time a failure is detected the function *error_handling_ISR* is called. This function shall implement the error management.

1.1 Safety Mechanisms implemented by software

According to the Safety Manual (and then to the safety concept and FMEDA) the software mechanisms required to guarantee the integrity of the INTC are:

- INTC_UNUSED_ILLEGAL - Unused interrupt vectors shall point, or jump, to an address which is illegal to execute, contains an illegal instruction, or in some other way causes detection of their execution;
- ISR_CHECK_TRIGGER_SET - The ISR will check that the triggering module actually shows a requested interrupt (for example, reading the interrupt request or status register in the peripheral);
- ISR_CHECK_PRIORITY - The ISR shall check that it was called with the correct priority;
- ISR_CHECK_CORE - The ISR checks that it is executed on the expected core using the relevant core register;
- ISR_CHECK_TRIGGER_CLEAR - Periodically check that interrupt flags in peripherals are cleared;
- ISR_DOUBLE_ASSIGNMENT - Interrupts where a short latency is safety-relevant are assigned to two (or more) cores and one of those cores checks (for example, using a shared variable) that the other core actually executes the ISR within the expected latency after the IRQ occurred;
- ISR_CHECK_EFFECT - Periodically check for effects of lost interrupts (for example, buffer overflow or underflow).

These safety mechanisms shall run online while the safety application runs. Only exception is the INTC_UNUSED_ILLEGAL which is related the INTC initialization.

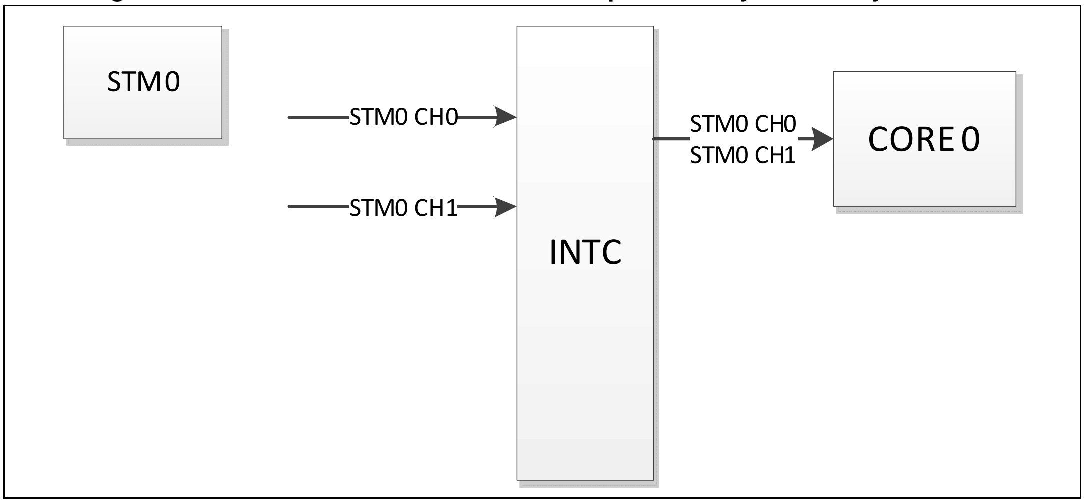
The user has the responsibility to choose the implementation which better fits his or her needs.

1.2 Use case example

To describe the software which shall be implemented, a use case example has been assumed. All described concepts can be reused for other use cases.

Let's assume 2 periodic interrupts are needed for functional purpose. These interrupts are serviced by the CORE0 and are implemented via 2 hardware interrupt sources, i.e. STM0 CH0 and STM0 CH1. A block schema of this simple example without any safety mechanism is shown on *Figure 2*.

Figure 2. Hardware resources and interrupts used by the safety function

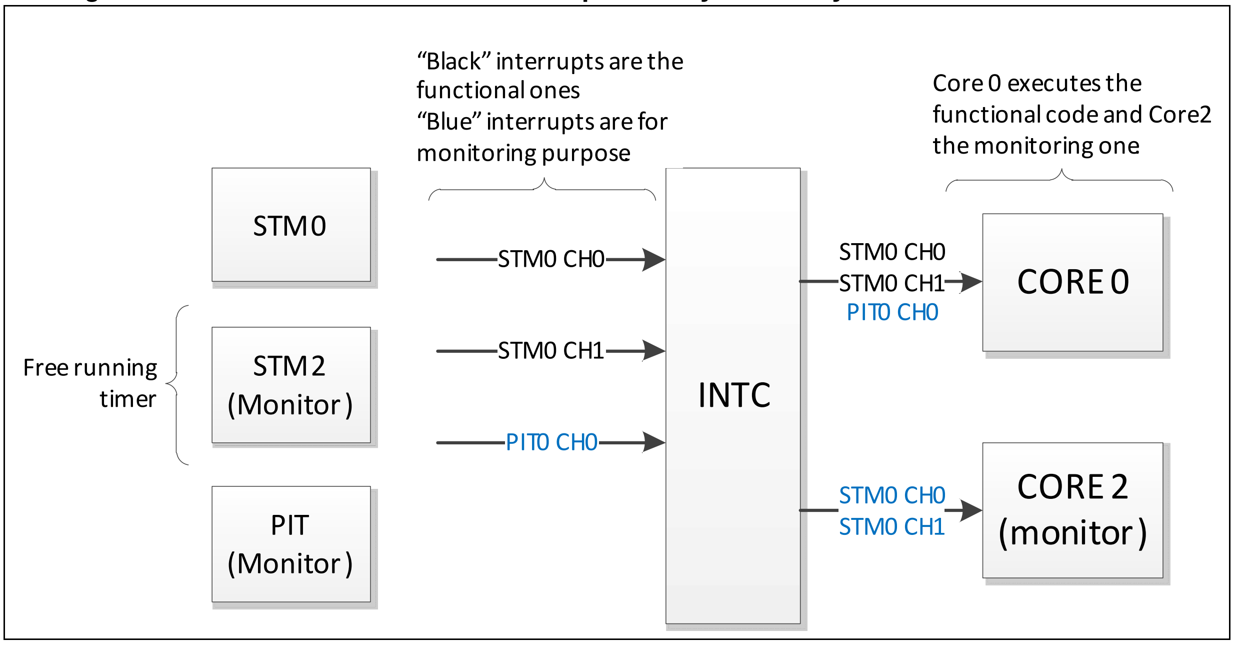


For safety purpose

- these 2 interrupts are serviced also by the CORE2 which can be considered the monitor of the correct execution in terms of timing of the interrupt service routine executed by CORE0;
- an additional interrupt is implemented, i.e. PIT0_CH0, to monitor the correct number of triggered functional interrupts;
- a free running timer, implemented by STM2, is used to check execution time of the interrupt service routine.

Figure 3 shows a block scheme of the hardware resources and interrupts to implement the safety function and its monitors.

Figure 3. Hardware resources and interrupt used by the safety function and its monitors


"Black" interrupts are the functional ones
"Blue" interrupts are for monitoring purpose
Core 0 executes the functional code and Core2 the monitoring one
STM0
Free running timer
STM2 (Monitor)
PIT (Monitor)
STM0 CH0
STM0 CH1
PIT0 CH0
INTC
STM0 CH0
STM0 CH1
PIT0 CH0
CORE 0
STM0 CH0
STM0 CH1
CORE 2 (monitor)

Next sections give details and hints about the implementation of these software mechanisms.

2 Initialization

This section describes mechanisms which must be applied during the initialization of the INTC, usually before the INTC is enabled.

Safety analysis assumes these mechanisms are applied before the safety application starts.

2.1 INTC_UNUSED_ILLEGAL

A faulty INTC can trigger unused ISRs. These ISR can have an impact on the safety function by executing an unexpected code.

To detect such a failure mode the INTC_UNUSED_ILLEGAL mechanism shall be implemented. Quoting the SM:

Unused interrupt vectors shall point, or jump, to an address which is illegal to execute, contains an illegal instruction, or in some other way causes detection of their execution.

To implement this mechanism all unused functions in the vector table shall not point to any unexpected location, but to a defined function, i.e. *dummy* in the *Example 1*.

Example 1: Unused interrupts point to a defined function

```
const uint32_t IntcIsrVectorTable[ ] = {
...
(unsigned int)&dummy, (unsigned int)&STM0_CH0_ISR, (unsigned
int)&STM0_CH1_ISR, ... /* ISRs 35 - 39 */
...
}
```

If executed, this function shall trigger an alarm to the application. This alarm can be a core exception or other means.

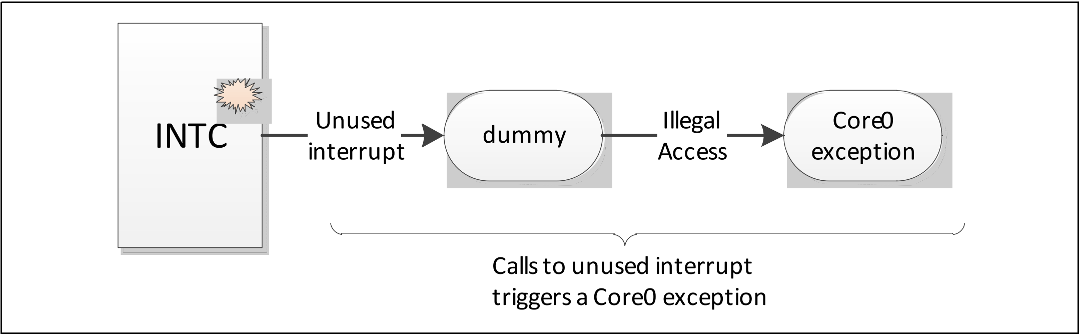
In the *Example 2* a core exception is triggered by writing to an illegal address.

Example 2: **Unused interrupts shall trigger an exception if executed**

```
void dummy (void) {
   tU32 *pointTest = 0x34567890;
   *pointTest = 40;
}
```

This software mechanism is executed only in case of failure. Until the INTC works properly, the *dummy* function is never executed.

Figure 4. Example of INTC_UNUSED_ILLEGAL implementation



3 Run-time checks

This section describes the software mechanisms to be executed in run-time, i.e. while the safety application runs. Most of these mechanisms are triggered either by the application which calls them periodically or by interrupts.

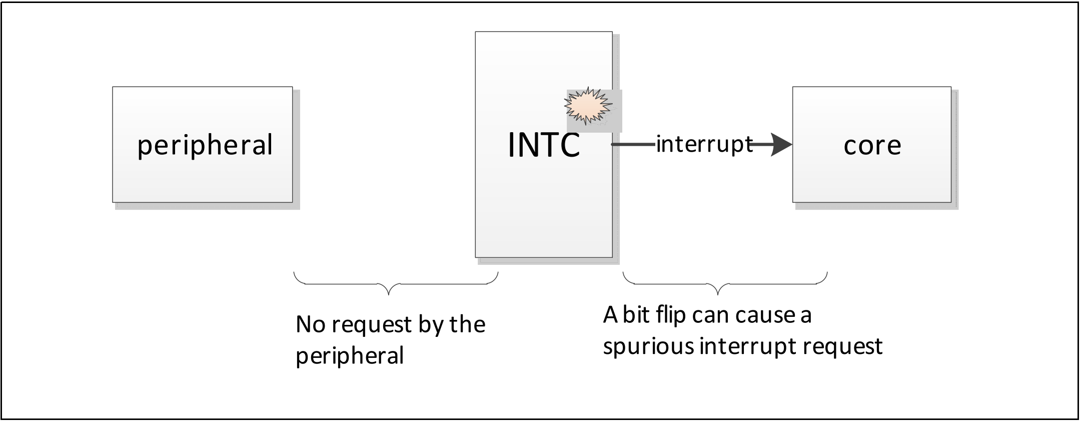
It is assumed that the INTC, and the whole device, has been “properly” configured before the safety application starts. “Properly” configured means accordingly to the functional and safety requirements.

3.1 INT_CHECK_TRIGGER_SET

The INTC requests the core to execute the ISR if an event is triggered by a peripheral, see *Figure 1*.

Due to a fault (e.g. an internal bit flip), the INTC could request the execution of an ISR even without any event by a peripheral, see *Figure 5*.

Figure 5. A bit flip in the INTC can cause a spurious interrupt request


peripheral
INTC
interrupt
core
No request by the peripheral
A bit flip can cause a spurious interrupt request

To detect such a spurious interrupt, the INT_CHECK_TRIGGER_SET mechanism shall be implemented. Quoting the SM:

The ISR will check that the triggering module actually shows a requested interrupt (for example, reading the interrupt request or status register in the peripheral).

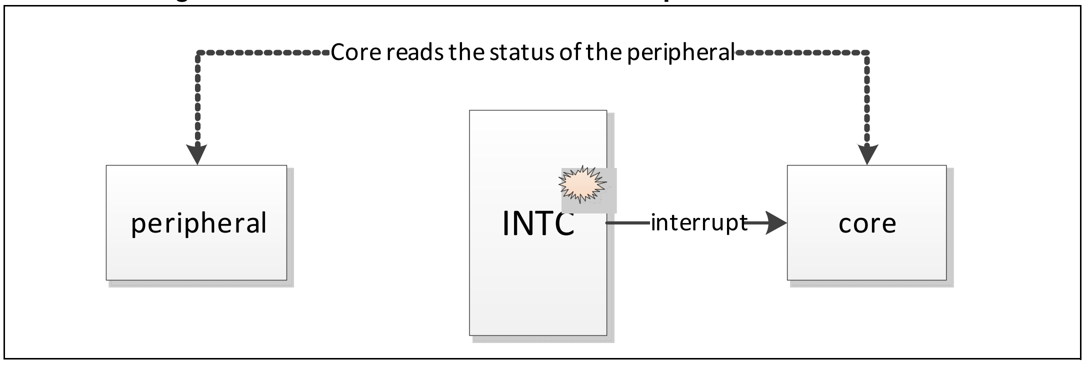
Each peripheral, which can request interrupts, embeds a means to verify whether the interrupt has been requested or not. In most cases this mechanism is a status flag or register, e.g.:

- Channel Interrupt Flag (CIF) implemented by the STM,
- Transmit FIFO Fill Flag (TFFF) implemented by the DSPI.

To implement this safety mechanism the core, each time is requested to execute an ISR, shall verify the status flag of the peripheral triggering the event. A failure is detected if the peripheral status flag shows that not all requests have been sent to the INTC.

Note: This check must be done before the ISR executes any instruction which can have an impact on the safety goal, for example before sending any safety related messages or before calling any safety related routines.

Figure 6. INT_CHECK_TRIGGER_SET implementation schema



Example 3 shows how this software mechanism can be implemented by software in case the requesting peripherals is the STM0.

Example 3: Interrupt service routine checks the status flash of the requesting module

```
void STM0_CH1_ISR_CORE0(void) {
...
   if (STM_0.Channel[1].CIR.B.CIF != 0x1) {
       error_handling_ISR(); //error handling
   }
...
}
```

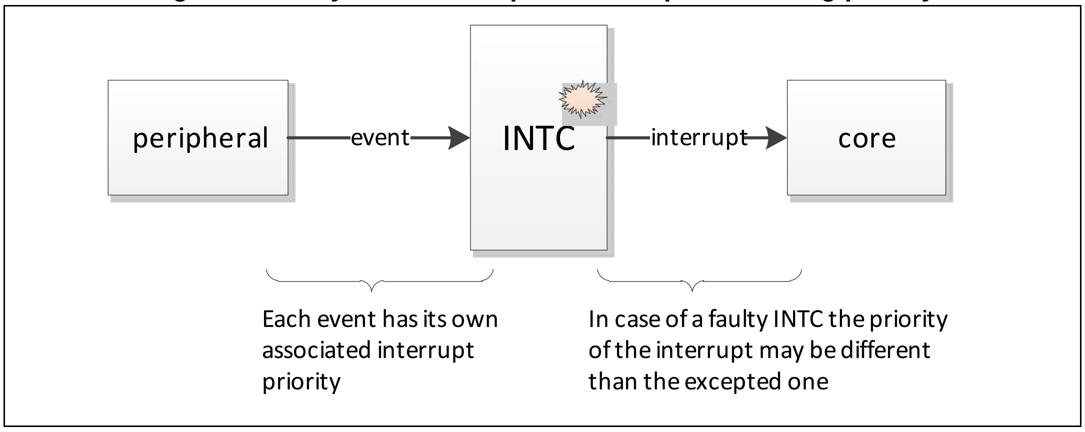
3.2 ISR_CHECK_PRIORITY

INTC provides priority-based preemptive scheduling of interrupt requests. A higher priority interrupt preempts a lower priority one.

Due to a failure in the INTC, e.g. a bit flip, an interrupt can be serviced with a priority which is higher or lower than the configured one. In such a case the preemption mechanism doesn't work as configured.

This failure has an impact on the safety function because the execution flow of interrupt service routines can be different than the expected one.

Figure 7. Faulty INTC can request interrupt with wrong priority


peripheral
event
INTC
interrupt
core
Each event has its own associated interrupt priority
In case of a faulty INTC the priority of the interrupt may be different than the excepted one

For example, let’s assume 2 ISRs configured to have different priorities (63 is the highest priority):

- ISR_A with priority 5 (lower priority)
- ISR_B with priority 6 (higher priority)

Safety function expects that ISR_A is preempted by ISR_B, but due to a failure in the INTC ISR_A is executed with the highest priority, i.e. 63. In such a case the ISR_B can’t preempt the ISR_A and the safety function is impacted.

To detect such failure mode, the ISR_CHECK_PRIORITY mechanism shall be implemented. Quoting the SM:

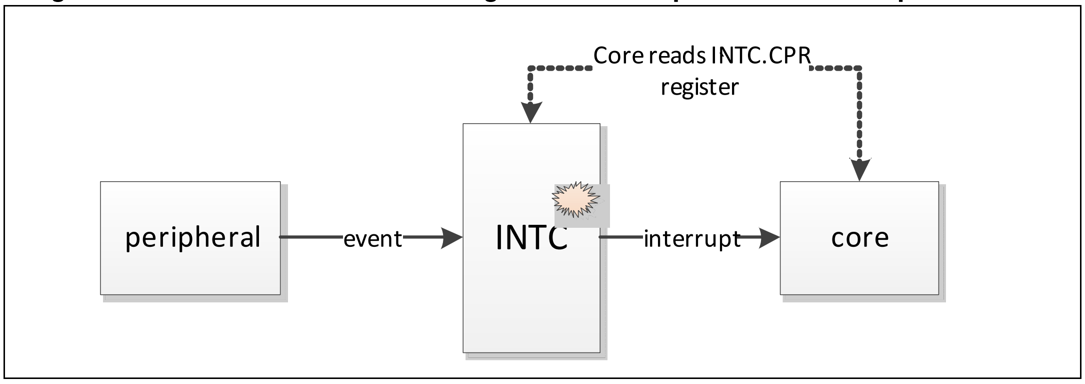
The ISR shall check that it was called with the correct priority.

The INTC contains a register showing the priority with which an ISR is executing, i.e. current priority register.

To implement this safety mechanism the core, each time is requested to execute an ISR, shall read the current priority in the INTC and compare it with the expected one.

Note: *This check must be done before the ISR executes any instruction which can have an impact on the safety goal, for example before sending any safety related messages or before calling any safety related routines.*

Figure 8. Core read the INTC.CPR register and compare it with the expected value



Example 4: Interrupt service routine verifies the current priority

```
void STM0_CH1_ISR_CORE0(void) {
...
   u32_CurrentPriority = INTC.CPR[0].B.PRI;
   if (u32_CurrentPriority!=PRIORITY_INT_STM_CH1) {
      error_handling_ISR(); //error handling
   }
...
}
```

It’s recommended to implement this safety mechanism not only inside safety related ISRs, but inside every ISR routine. Because if this failure mode affects a non safety related ISR, an impact to the safety related ones can appear. For example:

- ISR_A is NOT safety related with priority 5 (lower priority)
- ISR_B is safety related with priority 6 (higher priority)

If a failure in the INTC causes the execution of ISR_A with the highest priority, ISR_B can't preempt the ISR_A. As a result the ISR_B suffers for "late ISR" failure mode. This failure mode is detected by the ISR_CHECK_PRIORITY execute inside the ISR_A.

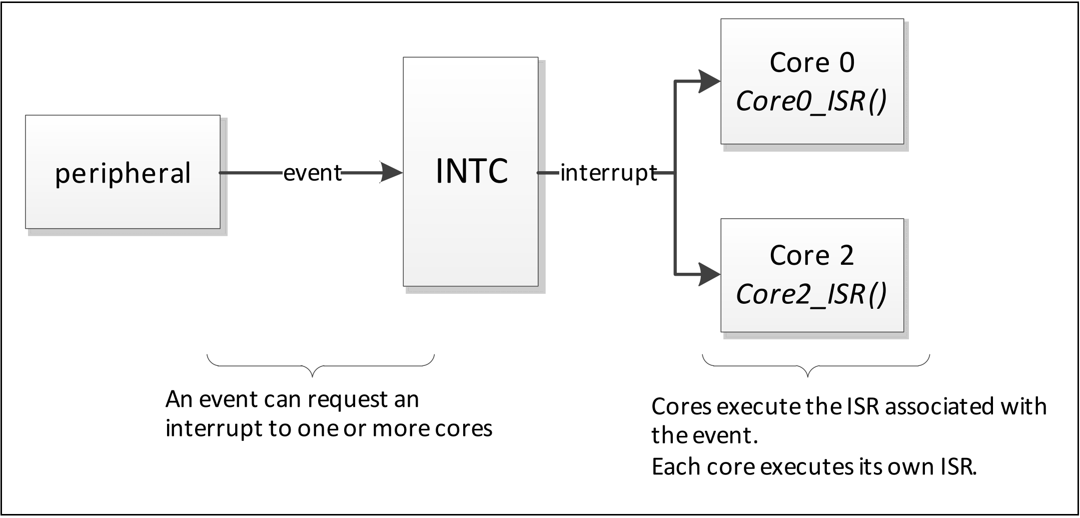
3.3 ISR_CHECK_CORE

In a multi core MCU as SPC57EM80x/SPC574K72x, the INTC can request an interrupt to one or more cores (see *Figure 9*). Usually the executed interrupt service routine is different core by core[b].

Figure 9 shows a peripheral which triggers an event to the INTC which as a result requests an interrupt to both cores:

- Core 0 executes the *Core0_ISR(),* and
- Core 1 executes the *Core1_ISR().*

Figure 9. INTC can request an interrupt to one or more cores

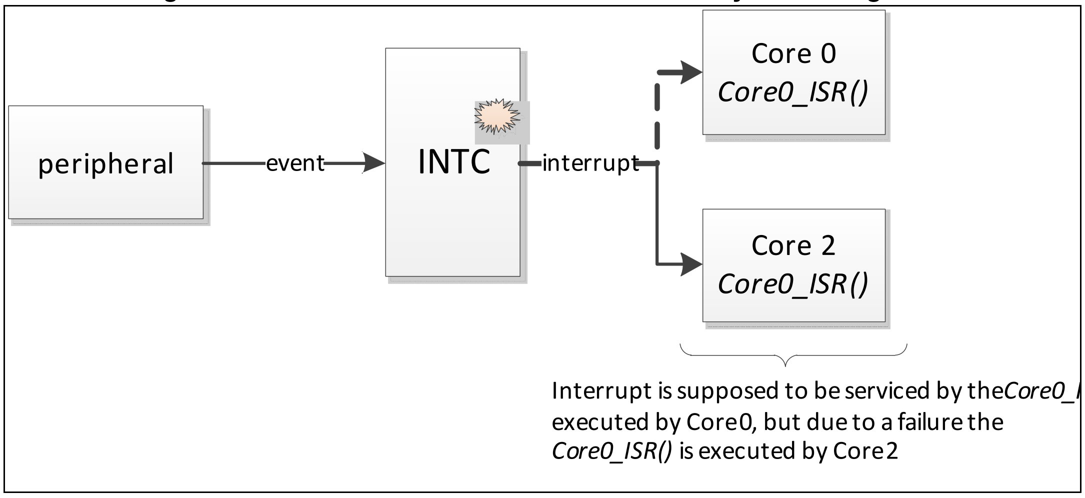

peripheral
event
INTC
interrupt
Core 0
Core0_ISR()
Core 2
Core2_ISR()
An event can request an interrupt to one or more cores
Cores execute the ISR associated with the event.
Each core executes its own ISR.

Due to a fault, e.g. a bit flip, the INTC can request a core to execute an interrupt service routine which should be executed by the other core. For example in *Figure 10* the INTC requests Core 2 to execute the *Core0_ISR* which is supposed to be executed by Core 0.

This failure mode can have an impact on the safety function because it may generate unexpected situations, for example Core2 tries to access resources reserved to Core 0.

b. If properly designed the same interrupt service routine can be executed by both cores.

Figure 10. Due to a failure the ISR is serviced by the wrong core


peripheral
event
INTC
interrupt
Core 0
Core0_ISR()
Core 2
Core0_ISR()
Interrupt is supposed to be serviced by the Core0_I
executed by Core0, but due to a failure the
Core0_ISR() is executed by Core2

To detect such failure mode, the ISR_CHECK_CORE mechanism shall be implemented. Quoting the SM:

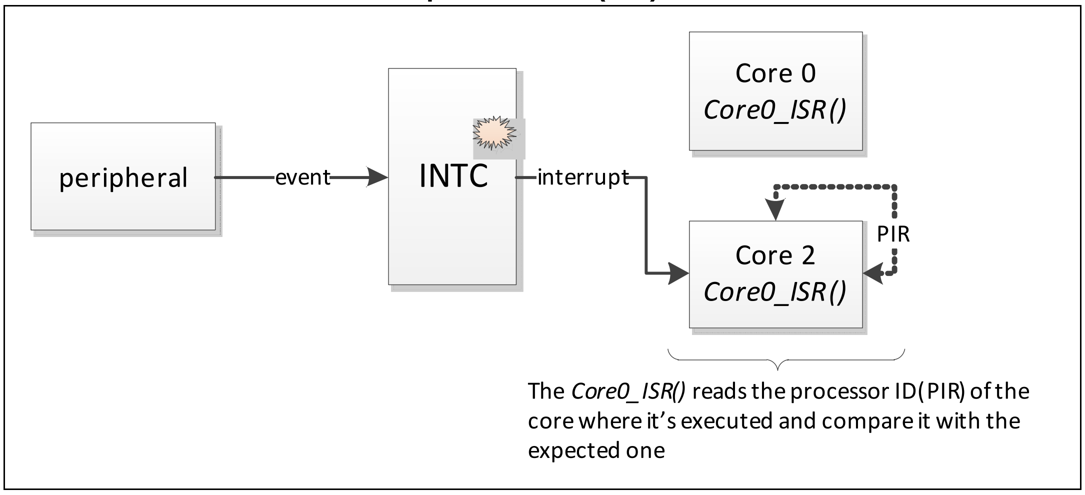
The ISR checks that it is executed on the expected core using the relevant core register.

To implement this safety mechanism the ISR, each time is requested to be executed, shall read processor ID and compare it with the expected one, see *Figure 11*.

Note: This check must be done before the ISR executes any instruction which can have an impact on the safety goal, for example before sending any safety related messages or before calling any safety related routines.

The processor ID (PIR) is a special purpose register of the core[c].

Figure 11. Core0_ISR verifies it's executed on the exepected core by reading the processor ID (PIR)


peripheral
event
INTC
interrupt
Core 0
Core0_ISR()
Core 2
Core0_ISR()
PIR
The Core0_ISR() reads the processor ID(PIR) of the
core where it's executed and compare it with the
expected one

Example 5 shows how this software mechanism can be implemented.

c. PIR register can be accessed in supervisor mode only.

Example 5: Example of implementation of the ISR_CHECK_CORE mechanism

```
void STM0_CH0_ISR_CORE0(void) {
...
   u32_CoreId = ProcessorID_GET();//the id of the core executing the ISR is
read
   if (u32_CoreId != EXPECTED_EXECUTION_CORE_0) {
      error_handling_ISR(); //error handling
   }
...
}

tU32 ProcessorID_GET(void){
   asm("mfspr r3, 286");
   return;
}
```

This mechanism is assumed to be done for all ISRs and not only for the safety-relevant ones. The execution of an ISR on the wrong core has an impact on the safety function because not only it slows 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.

3.4 ISR_CHECK_TRIGGER_CLEAR

As described in *Section 3.1: INT_CHECK_TRIGGER_SET*, each peripheral embeds a means to verify whether an interrupt request is pending or not. In most cases this mechanism is a status flag or a register.

This status flag:

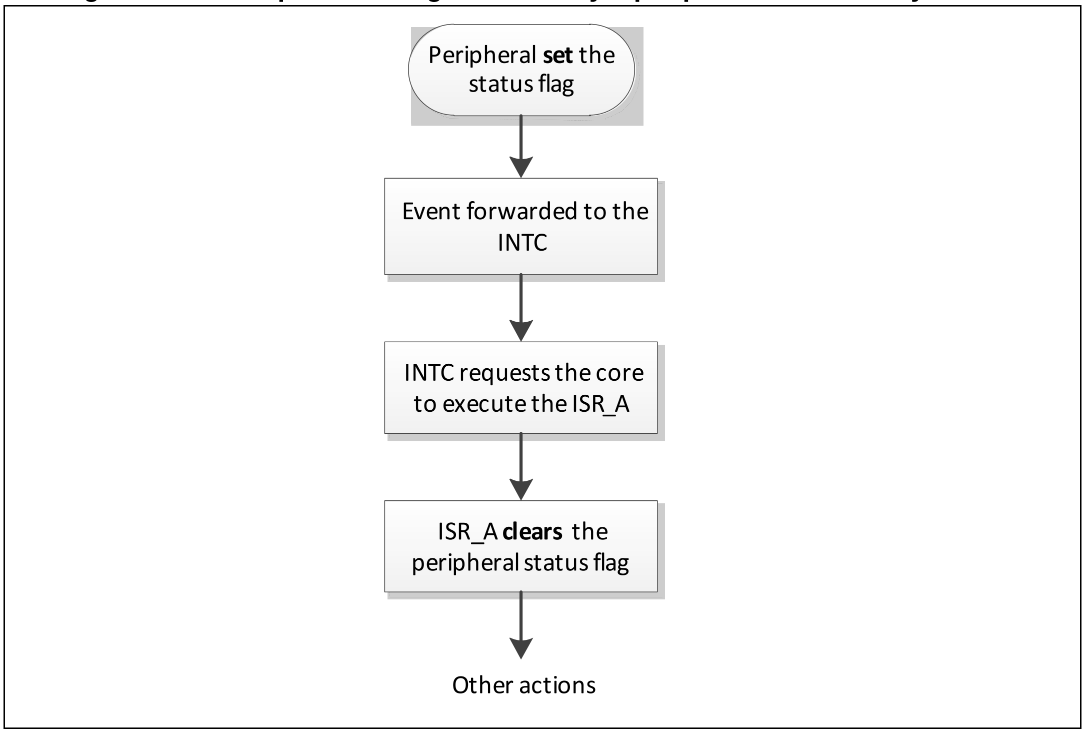
- is set by the peripheral when it triggers an event to the INTC, and
- is reset by software inside the ISR (e.g. ISR_A in *Figure 12*).

Assuming there is a periodic ISR, i.e. ISR_B[d], which has a lower priority than ISR_A. The ISR_A, triggered by the STM, can preempt the ISR_B execution.

With such assumptions and if everything works fine, the ISR_B never sees the STM status flag set to one. The ISR_B is never executing while the STM status flag set to one.

d. ISR_A and ISR_B are triggered by different interrupt sources.

Figure 12. Interrupt status flag is set the by a peripheral and clear by the ISR


Peripheral **set** the status flag
Event forwarded to the INTC
INTC requests the core to execute the ISR_A
ISR_A **clears** the peripheral status flag
Other actions

This is not true if there is a failure in the INTC. For example the STM set the flags[e] and forward the event to the INTC, but the INTC doesn't request the execution of ISR_A (i.e. missing interrupt failure mode). In this case the status flag is kept asserted to 1.

This is the idea behind the ISR_CHECK_TRIGGER_CLEAR. Quoting the SM:

Periodically check that interrupt flags in peripherals are cleared.

To implement such a mechanism, a periodic interrupt for monitoring purpose shall be added. The period of such an interrupt shall be smaller than the assumed FTTI and its priority shall be lower than other monitored safety relevant interrupts.

This periodic interrupt must check the interrupt flags of safety relevant interrupts. If at least one of this flag is set 1, a failure has been detected.

Example 6 shows how the ISR_CHECK_TRIGGER_CLEAR is implemented in the example. The periodic monitor interrupt is generated by the PIT. It monitors 2 safety relevant interrupts coming from channel 0 and 1 of STM_0.

Status flag to be verified by the *PIT0_ISR* are:

- STM_0.Channel[0].CIR.B.CIF and
- STM_0.Channel[1].CIR.B.CIF.

If one of these status flags is set to 1, the mechanism has potentially detected a missing interrupt. In such a case the error handling function should be called.

e. ISR_A of the example.

Since there is certain latency between an IRQ flag switching to 1 and the INTC activating the ISR in the core, a not reset flag needs to be validated by re-checking it after the known INTC latency[f].

Example 6: Implementation of the ISR_CHECK_TRIGGER_CLEAR mechanism

```
void PIT0_Isr(void) {
...
   if ((1==STM_0.Channel[0].CIR.B.CIF) ||
         (1==STM_0.Channel[1].CIR.B.CIF)) {
      DELAY(50); //due to the latency, flags are verified again
      if ((1==STM_0.Channel[0].CIR.B.CIF) ||
            (1==STM_0.Channel[1].CIR.B.CIF))
         error_handling_ISR();
   }
...
}
```

3.5 ISR_DOUBLE_ASSIGNMENT

In a multicore environment as SPC57EM80x/SPC574K72x, an interrupt can be forwarded to two cores (see *Figure 9*). One core executes the safety relevant ISR and the other one verifies its expected execution in terms of time, and eventually of result.

This mechanism is effective in case a safety function requires a short latency and execution time of an ISR. Other mechanisms described in this document can detect different failure modes (e.g. missing interrupt, wrong priority), but not the timely execution of an ISR.

Quoting the SM:

Interrupts where a short latency is safety-relevant are assigned to two (or more) cores and one of those cores checks (for example, using a shared variable) that the other core actually executes the ISR within the expected latency after the IRQ occurred.

The core0, executing the functional ISR, can use a shared resource, e.g. a flag saved in RAM, to indicate the status of the execution of the ISR to the other core. The shared flag is set when the functional ISR starts and is reset when it ends.

The monitoring ISR is triggered by the same interrupt event. It must monitor if the shared flag is set/reset in the appropriate time frame.

The implementation of this mechanism depends on the application and on the peripherals triggering the interrupts. A generic implementation is shown in *Figure 13*:

- Interrupt is triggered by a peripheral at time t_0
- At time t_1 the safety relevant ISR, i.e. ISR_A starts (it ends at time t_4)
- It's executed by core 0
- At time t_5 the monitoring ISR, i.e. ISR_B starts[g] (it ends at time t_6)

f. To have all details about the interrupt latency checks the section "Interrupt Latency" of the Reference Manual (see *Appendix A: Reference documents*).

g. ISR_B may start before than ISR_A; all comments are still valid.

- It's executed by core 2
- At time t_2, core 0 sets a shared flag to communicate to core 2 that ISR_A is starting executing the safety relevant actions
- Core 2 waits for the clearing of the shared flag, which is de-asserted at time t_3 by core 0 after the execution of the safety relevant actions
- ISR_B can verify the ISR_A timing (e.g. latency and execution time) based on an independent timer.

Figure 13. Generic implementation of the ISR_DOUBLE_ASSIGNMENT mechanism

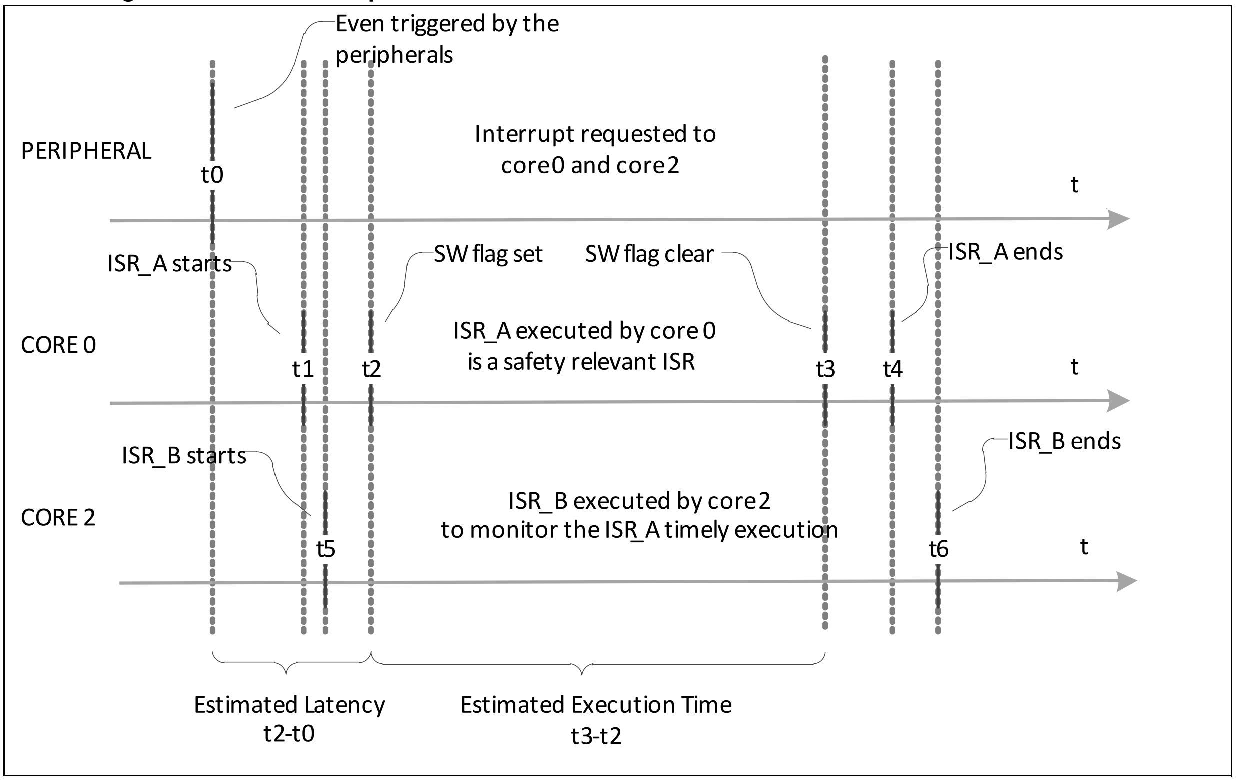

Even triggered by the peripherals
PERIPHERAL
t0
Interrupt requested to core0 and core2
t
ISR_A starts
SW flag set
SW flag clear
ISR_A ends
CORE 0
t1
t2
ISR_A executed by core 0 is a safety relevant ISR
t3
t4
t
ISR_B starts
ISR_B ends
CORE 2
ISR_B executed by core 2 to monitor the ISR_A timely execution
t5
t6
t
Estimated Latency t2-t0
Estimated Execution Time t3-t2

Figure 13 shows a generic example about how to implement the ISR_DOUBLE_ASSIGNMENT mechanism. Details about the implemented reference code are given hereafter (*Figure 14*).

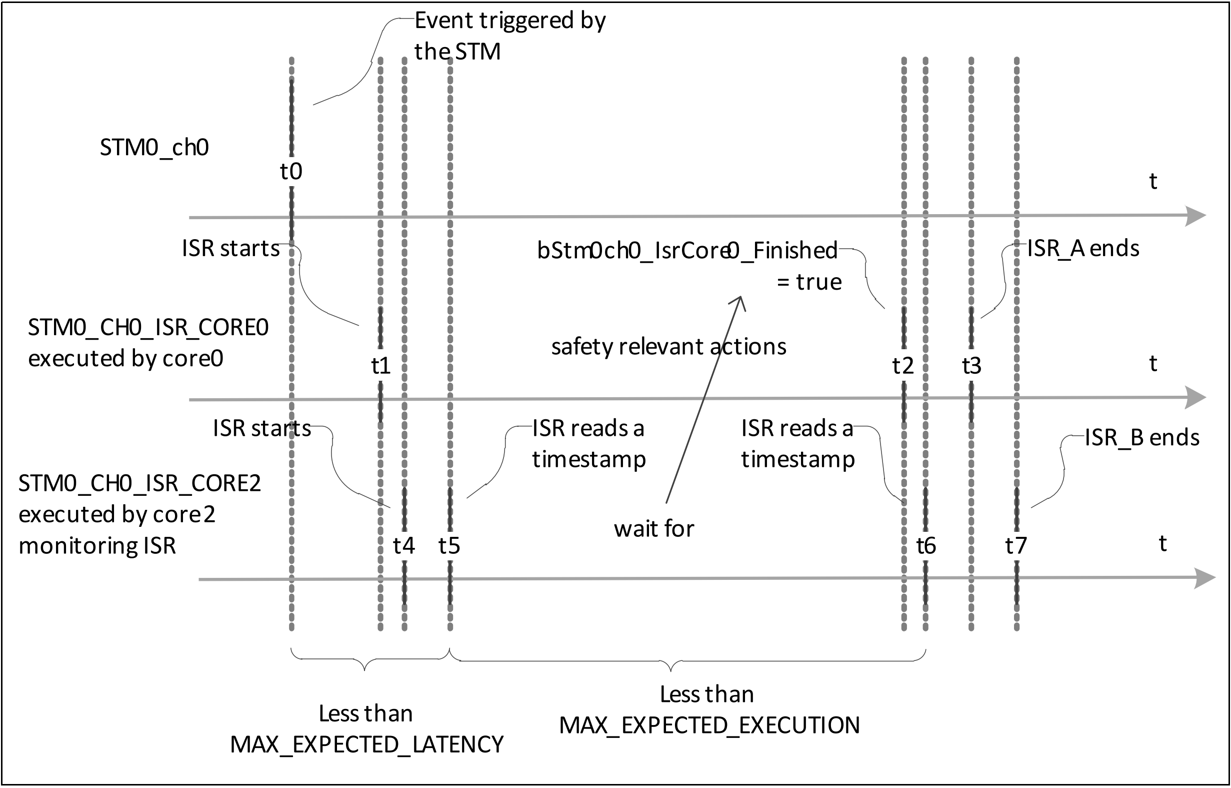
Reference code considers an interrupt sources, i.e. channel #0 of STM0, which triggers interrupt to core 0 and core 2:

- STM0_CH0_ISR_CORE0 executed by core 0 executes the safety relevant actions
- STM0_CH0_ISR_CORE2 monitors the timely execution of STM0_CH0_ISR_CORE0.

The monitoring ISR verifies that the total execution time (including latency) of STM0_CH0_ISR_CORE0 is below a certain threshold. To do this check, core 2 reads some timestamps using an independent free-running timer, i.e. STM2[h].

h. STM2 does not trigger any interrupt, but it is only used at free running timer. STM belongs to a different lake respect STM0 to decrease the probability of CCF.

Figure 14. Schema of the example of the implementation of ISR_DOUBLE_ASSIGNMENT


Event triggered by the STM
STM0_ch0
t0
t
ISR starts
bStm0ch0_IsrCore0_Finished = true
ISR_A ends
STM0_CH0_ISR_CORE0 executed by core0
safety relevant actions
t1
t2
t3
t
ISR starts
ISR reads a timestamp
ISR reads a timestamp
ISR_B ends
STM0_CH0_ISR_CORE2 executed by core2 monitoring ISR
wait for
t4
t5
t6
t7
t
Less than MAX_EXPECTED_LATENCY
Less than MAX_EXPECTED_EXECUTION

The STM0_CH0_ISR_CORE0 executes the safety relevant actions and, at time t_2, sets a shared variable, i.e. bStm0ch0_IsrCore0_Finished, to communicate to core 2 the ISR ends.

Example 7: Functional ISR sets a shared variable

```
void STM0_CH0_ISR_CORE0(void) {
...
   /* execute its safety relevant actions*/
...
   /* This flag indicates that the ISR is ended.*/
   bStm0ch0_IsrCore0_Finished=true;
}
```

The STM0_CH0_ISR_CORE2 executes multiple actions.

At time t_5, it reads a timestamp from the STM2 counter (see *Example 8*) and it verifies that latency is below an expected threshold (see *Example 9*).

Example 8: Monitoring ISR reads a timestamp from an independent timer, i.e. STM2

```
void STM0_CH0_ISR_CORE2(void) {
...
   /* a first timestamp is read from the free running timer STM2 */
   u32InitTimerCounter = STM_2.CNT.R;
...
}
```

Example 9: Monitoring ISR verifies that latency is below an expected threshold

```
void STM0_CH0_ISR_CORE2(void) {
...
   /* interrupt is expected when the STM0 counter reaches the value
      STM_0_CH0_TIMEOUT_INIT. latency is estimated by reading the STM0_CH0
      counter and subtracting the STM_0_CH0_TIMEOUT_INIT.*/
   u32latency =
CounterSubWithOverflow(i)(STM_0.CNT.R,STM_0_CH0_TIMEOUT_INIT);
   if (u32latency>MAX_EXPECTED_LATENCY) {
      error_handling_ISR(); //error handling
   }
...
}
```

Then it waits for the *bStm0ch0_IsrCore0_Finished* set to true by the functional ISR, i.e. time t_2.

At this point it can read another timestamp from the STM2, i.e. time t_6. Subtracting the 2 read timestamps, it can verify the execution time of STM0_CH0_ISR_CORE0 is below a threshold, i.e. MAX_EXPECTED_EXECUTION.

The reference code implements a software timeout to detect not expected situation as for example the *bStm0ch0_IsrCore0_Finished* never set to true (see *Example 10*).

Example 10: Execution time of functional ISR is estimated by monitoring one

```
void STM0_CH0_ISR_CORE2(void) {
...
   do
   /* passed time considering the overflow is measured using the the free
      running timer STM2. A second time stamp is read */
   u32WaitingTime = CounterSubWithOverflow(i)(STM_2.CNT.R,
u32InitTimerCounter);
   /* loop exits if:
      a) interrupt is serviced by Core 0, or
      b) time-out expires */
   while( (bStm0ch0_IsrCore0_Finished==false) &&
(u32WaitingTime<MAX_EXPECTED_EXECUTION));

   /* check if Core 0 executed the ISR in time */
   if (bStm0ch0_IsrCore0_Finished==false) {
      error_handling_ISR();
   } else {
   /* flag is reset for the next cycle */
      bStm0ch0_IsrCore0_Finished=false;
   }
...
}
```

i. This function subtracts 2 values taking into account the counter overflow situation.

3.6 ISR_CHECK_EFFECT

Instead of checking event status flags, priorities and core ID, a different way to detect failures in the INTC is to verify directly the functional effect of a wrong interrupt service routing execution.

This generic mechanism includes checking any functional parameter related to the application to detect failure in the INTC, for example:

- DSPI buffer over-run event in the receiving FIFO may indicate that ISR used to retrieve data out from the receiving FIFO has not been executed as expected, or
- If the application expects a fixed number of interrupt in a FTTI, it is possible to count them and compare it with the expected number, or
- In case of interrupts with a known period it's possible to measure the elapsed time between 2 consecutive interrupts and compare it with the expected period.

Quoting the SM:

Periodically check for effects of lost interrupts (for example, buffer overflow or underflow).

It is possible to use such kind of methodologies to detect different failures modes, e.g. missing/spurious interrupt and late interrupts.

Note: Implementation of this safety mechanism is strictly application dependent. Aim of this section is not to describe all possible implementations, but describes some common ones.

The reference code considers 2 safety relevant periodic ISRs

- *STM0_CH0_ISR_CORE0* triggered by STM0 ch0
- *STM0_CH1_ISR_CORE0* triggered by STM0 ch1

and a monitor one

- *PIT0_Isr* triggered by PIT0.

Each time the *STM0_CH0_ISR_CORE0* (or *STM0_CH1_ISR_CORE0*) is executed a software counter is incremented (see *Example 11*).

Example 11: Safety relevant ISR increases a software counter when executed

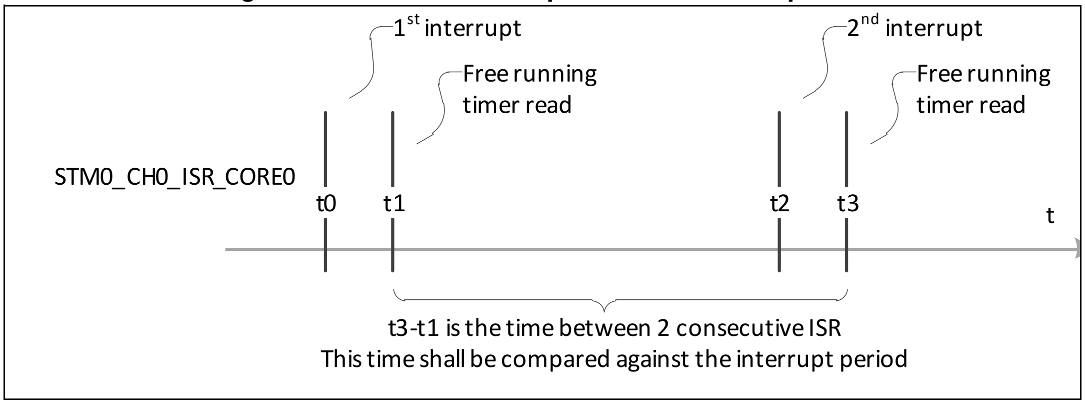
```
void STM0_CH0_ISR_CORE0(void) {
...
   /* this global variable is used to count the number of interrupt in a
      certain period. It is used to implement the ISR_CHECK_EFFECT safety
      mechanism */
   u32NumOf_Stm0ch0core0_ISR++;
...
}

void STM0_CH1_ISR_CORE0(void) {
...
   u32NumOf_Stm0ch1core0_ISR++;
...
}
```

PIT0_Isr is called every FTTI. It reads the software counters to verify how many times, safety relevant ISRs have been executed.

```
void PIT0_Isr(void) {
...
   /* Considering application expects a certain number of STM interrupts in
      a certain period. Missing interrupt can detected by reading, each
      period, the number of executed STM interrupt
      and compare it with the expected one. */
   if (
      EXPECTED_NUMOF_STM0CH0CORE0_ISR != u32NumOf_Stm0ch0core0_ISR ||
      EXPECTED_NUMOF_STM0CH1CORE0_ISR != u32NumOf_Stm0ch1core0_ISR ) {
         error_handling_ISR(); //error handling
   }
...
   /* Counters are re-initialized to 0 */
   u32NumOf_Stm0ch1core0_ISR = 0;
   u32NumOf_Stm0ch0core0_ISR = 0;
...
}
```

Since the 2 safety relevant interrupts are periodic additional checks can be implemented. Two timestamps read from a free running timer can be used to verify the period (see *Figure 15*).

Figure 15. Periodic interrupt can measure its period



1st interrupt
Free running timer read
2nd interrupt
Free running timer read
STM0_CH0_ISR_CORE0
t0
t1
t2
t3
t
t3-t1 is the time between 2 consecutive ISR
This time shall be compared against the interrupt period

In our example the free-running timer is provided by the STM2.

Each time the STM0_CH0_ISR_CORE0[j] is called, it reads a timestamp from a free-running counter. It uses this timestamp with the one from the last ISR invocation to measure the execution period and compares it with the expected one, see *Example 12*.

j. Same mechanism is applied also on the STM0_CH1_ISR_CORE0.

Example 12: Period of safety relevant ISR is measured via a 2 consecutive read of free running counter

```
void STM0_CH0_ISR_CORE0(void) {
...
   /* In case the interrupt to be monitored is periodic its possible measure
      its frequency via a free running timer every time the ISR is called:
         -store the current value of the timer and
         -compare it against the stored value from the last ISR invocation.
      This can detect the spurious/late/missing ISR execution. */
...
   u32CurrentTimeStamp = STM_2.CNT.R;
   u32_MeasuredPeriod = CounterSubWithOverflow(k)(u32CurrentTimeStamp,
u32PreviousTimeStamp);
   if ((u32_MeasuredPeriod < EXPECTED_STM0_CH0_PERIOD_LOW) ||
(u32_MeasuredPeriod > EXPECTED_STM0_CH0_PERIOD_HIGH)) {
      /* measured period not compliant with the expected time */
      error_handling_ISR();
   }
...
}
```

k. This function subtracts 2 values taking into account the counter overflow situation.

4 Summary

This document based on an example describes the recommended software mechanisms to grant the correct execution of the interrupt service routines.

The document cannot describe all possible solutions to implement these software mechanisms because some of them are application dependent. Aim of this AN is to make the user aware of what needs to be implemented in software.

These software mechanisms are not strictly mandatory, but they have been assumed to be implemented by the safety analysis done for SPC57EM80x/SPC574K72x. User may not implement them, but it’s up to him to provide other methods to cover the respective failure modes.

Appendix A Reference documents

1. SPC57EM80xx - 32-bit Power Architecture® based MCU with up to 4 Mbyte Flash and 304 Kbyte RAM memories (RM0314, DocID022530)
2. SPC574Kxx - 32-bit Power Architecture® based MCU for automotive applications (RM0334, DocID023671)
3. 32-bit Power Architecture® based MCU for automotive applications (SPC57EM80E7, SPC57EM80C3, DocID022502)
4. 32-bit Power Architecture® based MCU for automotive applications (SPC574K72E5, SPC574K72E7, DocID023601)
5. Safety application guide for SPC57EM80x family (AN4252, DocID024256)
6. Road vehicles — Functional safety, ISO/FDIS 26262:2011

Appendix B Acronyms

Table 1. Acronyms

Acronym	Name
SoC	System on Chip
FMEDA	Failure Modes, Effects, and Diagnostics Analysis
ASIL	Automotive Safety Integrity Level
SM	Safety Manual
RM	Reference Manual
INTC	Interrupt controller
MCU	Microcontroller
ISR	Interrupt service routine
STM	System Timer Module
DSPI	Deserial Serial Peripheral Interface
PIT	Periodic Interrupt Timer
FTTI	Fault Tolerant Time Interval
CCF	Common Cause Failure

Revision history

Table 2. Document revision history

Date	Revision	Changes
05-Aug-2013	1	Initial release.
17-Sep-2013	2	Updated Disclaimer.