Interrupt subsystem

XMC1000, XMC4000

About this document

Scope and purpose

This application note provides information on how to configure and enable interrupts in the XMC1000 and XMC4000 microcontrollers, as well as some hints on interrupt handling and improving interrupt performance.

Intended audience

This document is intended for engineers who intend to use interrupts in their application with the XMC1000 and XMC4000 microcontrollers.

Table of contents

1 Using interrupts

An interrupt is a service request signaled by a peripheral, or generated by a software request.

Setting up an interrupt involves the following steps:

- Configure the Interrupt Priority Level (IPL) (optional)
- Define an Interrupt Service Routine (ISR)
- Enable the Interrupt

1.1 Configure the Interrupt Priority Level

In the XMC1000, an interrupt node can be configured to one of 4 priority levels (0 to 3) while in the XMC4000, an interrupt node can be configured to one of 64 priority levels (0 to 63).

In both cases, the level with the highest priority is level 0 and by default, all nodes are configured to level 0. To change the interrupt priority level, the CMSIS function NVIC_SetPriority() can be used.

For example, to configure node 3 in the XMC1200, which is connected to ERU0 service request output 0 (ERU0_SR0), to level 3:

```
//This function configures node 3 to priority level 3
NVIC_SetPriority(ERU0_0_IRQn, 3U);
```

ERU0_0_IRQn is the enumerated constant for node 3 in the XMC1200. The complete list of enumerated constants can be found in the respective device header file “XMCxx00.h”.

1.1.1 Priority grouping in XMC4000

The XMC4000 additionally supports the grouping of priority values into group priority and subpriority fields.

Group priority determines if an interrupt can be preempted by another interrupt, while subpriority determines the order in which pending interrupts of the same group priority are processed.

The available number of group priorities and subpriorities is selected through the field PRIGROUP as shown in Table 1.

Table 1 Priority grouping

PRIGROUP	Number of group priorities	Number of subpriorities
0 or 1	64	1
2	32	2
3	16	4
4	8	8
5	4	16
6	2	32
7	1	64

The following code shows an example of configuring PRIGROUP, and the group and sub-priority values of ERU0_SR0 (node 1 in XMC4000) using CMSIS functions:

```
// This function configures PRIGROUP to 4
NVIC_SetPriorityGrouping(4);
```

```
      …
      // This function configures node 1 to group priority level 7 and
      // subpriority level 0 based on current PRIGROUP.
      NVIC_SetPriority(ERU0_0_IRQn,
NVIC_EncodePriority(NVIC_GetPriorityGrouping(), 7, 0));
```

1.2 Define an Interrupt Service Routine (ISR)

An interrupt node may be assigned to more than one interrupt trigger source. For example in the XMC1200, node 1 (SCU_SR1) triggers an interrupt request whenever there is a standby clock failure or VDDP pre-warning event, assuming these events are enabled for interrupt generation.

When defining an ISR or interrupt handler, the following should be considered:

- Identify the handler name from the list given in the device's startup file "startup_XMCxx00.s".
- The status flag of the highest priority event must be checked first to determine if it is the interrupt trigger source and the processing task executed before proceeding to check for the next highest priority event.

Using the XMC peripheral library for SCU, the ISR for node 1 could take the following form:

```
void SCU_1_IRQHandler(void)
{
      //Check if standby clock failure event status flag is set
      if(((XMC_SCU_INTERUPT_GetEventStatus() &
XMC_SCU_INTERRUPT_EVENT_STDBYCLKFAIL)>>22) == 1U)
      {
      //Write to SRCLR to clear active flag
  XMC_SCU_INTERRUPT_ClearEventStatus(XMC_SCU_INTERRUPT_EVENT_STDBYCLKFAIL);
      … //Process the interrupt
      }
      //Check if VDDP pre-warning event status flag is set
            if(((XMC_SCU_INTERUPT_GetEventStatus() &
XMC_SCU_INTERRUPT_EVENT_VDDPI)>>3) == 1U)

      {
            //Write to SRCLR to clear active flag
         XMC_SCU_INTERRUPT_ClearEventStatus(XMC_SCU_INTERRUPT_EVENT_VDDPI);
            … //Process the interrupt
      }
     return;
}
```

For events that are not enabled for interrupt generation, there is no need to check their status flags within the interrupt handler.

Interrupt handlers are by default allocated to the flash memory by the Integrated Development Environment (IDE) tools, such as Infineon DAVE™ 4 and µKeil Vision.

If an interrupt service request is generated while the flash is busy, and the CPU attempts to vector to the handler location in the flash:

- In the XMC1000, a system bus stall will occur and the CPU execution of the interrupt handler will be carried out only after the flash becomes available again.
- In the XMC4000, a bus error will be generated.

To avoid the above scenarios, interrupt handlers can be assigned to the SRAM, see section 3.3.2.

1.3 Enable the interrupt

To generate an interrupt request to the CPU, both the interrupt node and the underlying interrupt trigger sources must be enabled for interrupt request generation.

Our recommendation is to first clear the interrupt node „pending" status and the event status flags of the interrupt trigger sources, before enabling them. Use the following steps:

- Configure the interrupt node pointer
- Clear the interrupt node pending status
- Enable the interrupt node
- Clear the event status flags
- Enable the interrupt trigger sources

The following example code sequence enables interrupt node 1 (SCU_SR1) in XMC1200, and the standby clock failure and VDDP pre-warning events for interrupt request generation:

```
//This function clears node 1 pending status
NVIC_ClearPendingIRQ(SCU_1_IRQn);
//This function enables node 1 for interrupt request generation
NVIC_EnableIRQ(SCU_1_IRQn);
//Initializes event status flags SBYCLKFI and VDDPI in SRRAW register to 0
XMC_SCU_INTERRUPT_ClearEventStatus(XMC_SCU_INTERRUPT_EVENT_STDBYCLKFAIL);
XMC_SCU_INTERRUPT_ClearEventStatus(XMC_SCU_INTERRUPT_EVENT_VDDPI);
//Enables SBYCLKFI and VDDPI events for interrupt request generation
XMC_SCU_INTERRUPT_EnableEvent(XMC_SCU_INTERRUPT_EVENT_STDBYCLKFAIL);
XMC_SCU_INTERRUPT_EnableEvent(XMC_SCU_INTERRUPT_EVENT_VDDPI);
```

1.3.1 Interrupt node assignment in XMC1400

Only on an XMC1400 device, each NVIC node can be assigned to one of three service request sources (Source A, B or C) or the logical OR of sources A and B.

Therefore, the enabling of interrupts requires an additional step to select the service request source for the NVIC node.

For example, node 9 can be assigned to the following as shown in Figure 1:

- Source A - USIC0_SR0
- Source B – USIC1_SR0
- Source C – ERU0_SR0
- Source A OR B – USIC0_SR0 OR USIC1_SR0

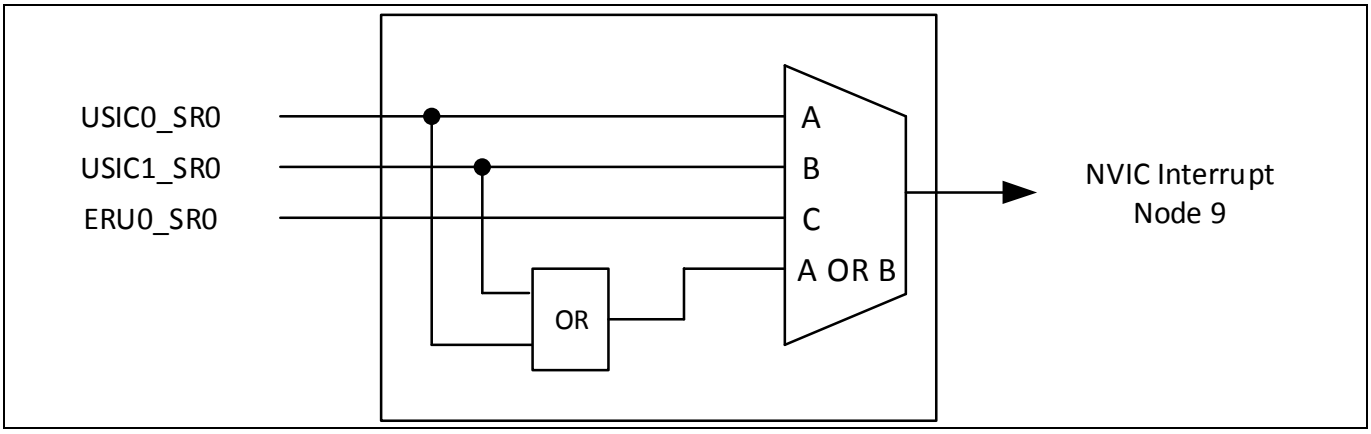

Figure 1 Interrupt node assignment for Node 9

The following code snippet depicts the configuration for node 9 to select ERU0_SR0 as the interrupt source.

```
/* Select Source C for Node-9 */
XMC_SCU_SetInterruptControl(IRQ9_IRQn, XMC_SCU_IRQCTRL_ERU0_SR0_IRQ9);
NVIC_SetPriority(IRQ9_IRQn, 3U); /* Assign lowest priority to Node-9 */
NVIC_EnableIRQ(IRQ9_IRQn); /* Enable Node-9 */
```

2 Processing a pulse interrupt with multiple trigger sources

In the XMC4000, interrupts can be of the type “Level” or “Pulse” while in the XMC1000, all interrupts are of the type “Pulse”. This section discusses the processing of a pulse interrupt with multiple trigger sources.

Assume that an interrupt node is assigned to the service request output of Module n (MOD_n_SR) and there are two events, A and B, that generate pulse-type interrupt requests. The pseudo code of the interrupt handler is given below:

```
void MOD_n_IRQHandler(void)
{
      //Check if event A status flag is set
      {
            //Process the interrupt
      }
      //Check if event B status flag is set
      {
            //Process the interrupt
      }
      return;
}
```

To illustrate how such an interrupt is processed, consider the following three scenarios:

- Events A and B occur before interrupt is serviced
- Event B occurs while servicing interrupt event A
- Event A occurs while servicing interrupt event B

Events A and B occur before interrupt is serviced

Figure 2 below shows the case where events A and B occur before interrupt is serviced. The time instances 1 to 4 are explained below:

1. Event A generates an interrupt pulse at the module service request output and causes the interrupt to switch to a ‘pending’ state.
2. Event B similarly generates an interrupt pulse. However, as the interrupt is already pending due to event A, the second interrupt pulse has no additional effect.
3. The CPU enters the interrupt handler, sees that event A status flag is set and executes the corresponding handler code. The interrupt changes from pending to active state.

The CPU checks for event B status flag and sees that it is also set. Therefore, the event B handler code will also be executed within the same instance of interrupt entry.

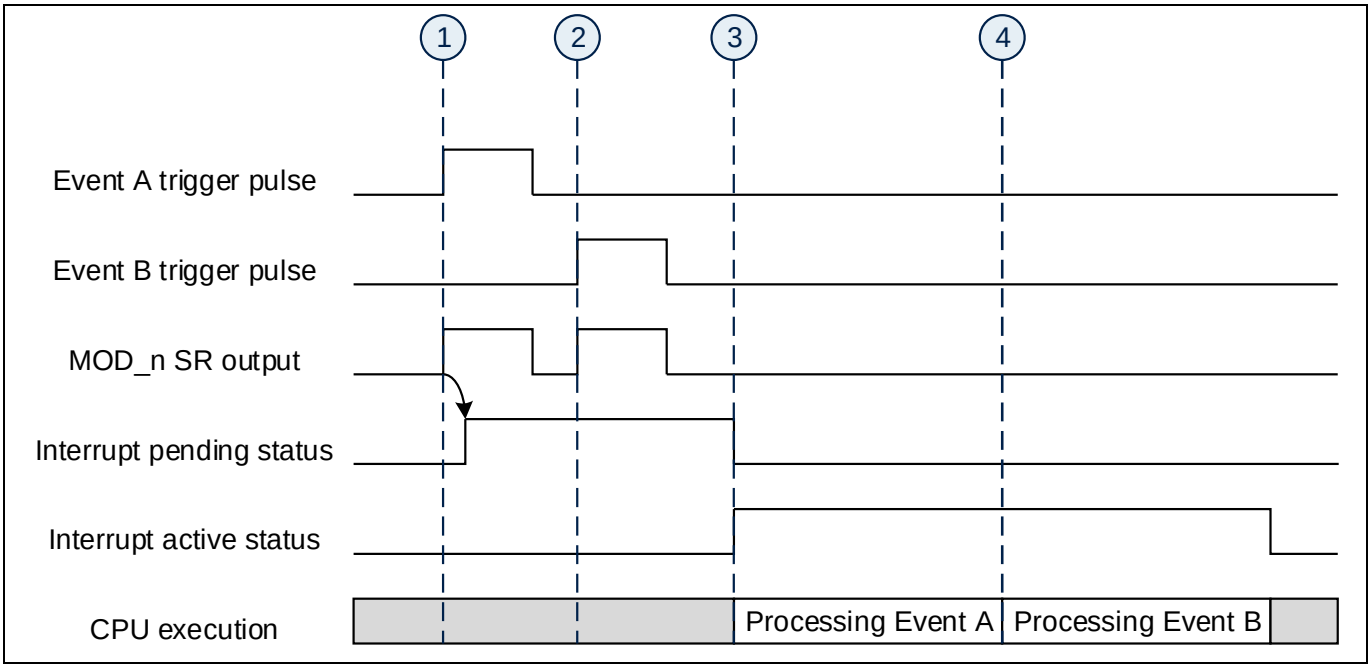

Figure 2 Events A and B occur before interrupt is serviced

Event B occurs while servicing interrupt event A

Figure 3 below shows the case where event B becomes pending while CPU is still servicing the interrupt triggered by event A. The time instances 1 to 5 are explained below:

1. Event A generates an interrupt pulse at the module service request output and causes the interrupt to switch to a ‘pending’ state.
2. CPU enters the interrupt handler, sees that event A status flag is set and executes the corresponding handler code. The interrupt changes from pending to active state.
3. Event B generates an interrupt pulse that changes the interrupt state to be active and pending.
4. The CPU checks for event B status flag and sees that it is also set. Therefore, the event B handler code will also be executed within the same instance of interrupt entry. Once this is done, the interrupt active state is removed.
5. However, as there is still an interrupt pending (due to event B), the CPU enters the interrupt handler again. Since all event status flags have already been cleared, the handler exits without processing any events, i.e. a dummy interrupt.

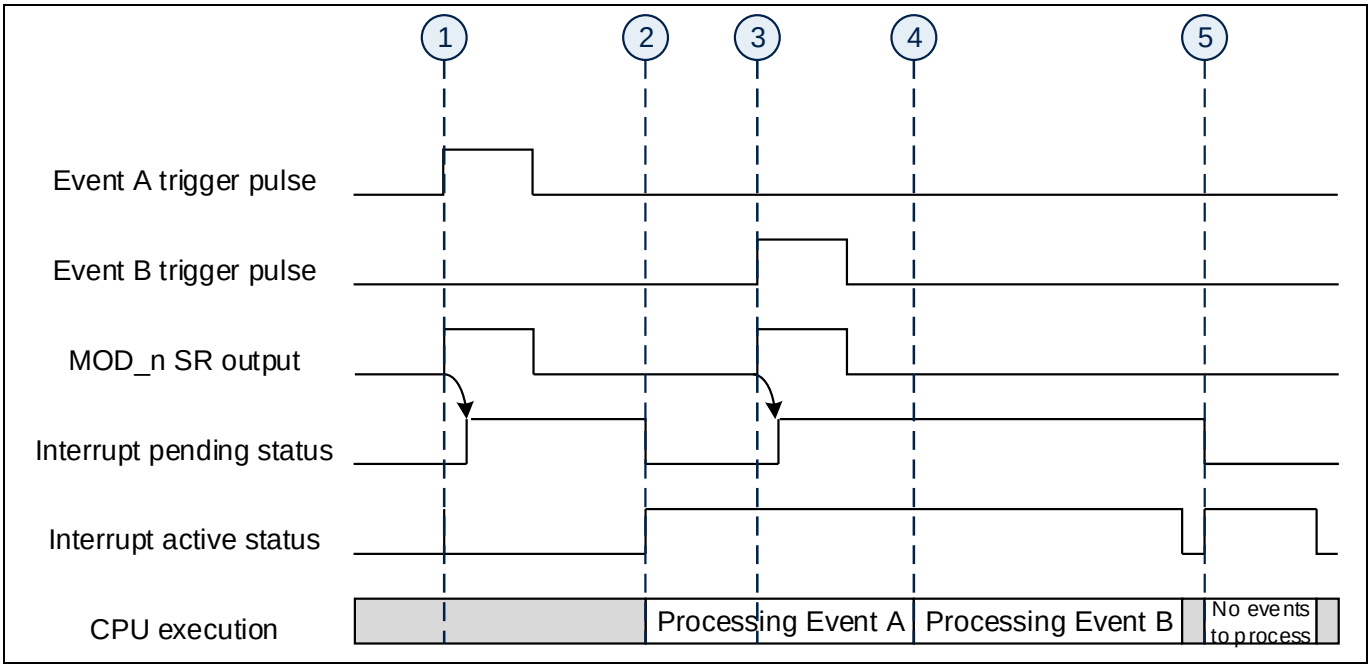

Figure 3 Event B becomes pending while servicing interrupt A

Event A occurs while servicing interrupt event B

Figure 4 below shows the case where event A becomes pending while the CPU is still servicing the interrupt triggered by event B. The time instances 1 to 4 are explained below:

1. Event B generates an interrupt pulse at the module service request output and causes the interrupt to switch to a ‘pending’ state.
2. The CPU enters the interrupt handler and causes the interrupt to change from pending to active state. CPU detects that only event B status flag is set and executes the corresponding handler code.
3. Event A generates an interrupt pulse that changes the interrupt state to be active and pending.
4. After executing the handler code for event B, the CPU exits the interrupt handler. However, as there is still an interrupt pending (due to event A), the CPU enters the interrupt handler again. The CPU detects that event A status flag is set and executes the corresponding handler code.

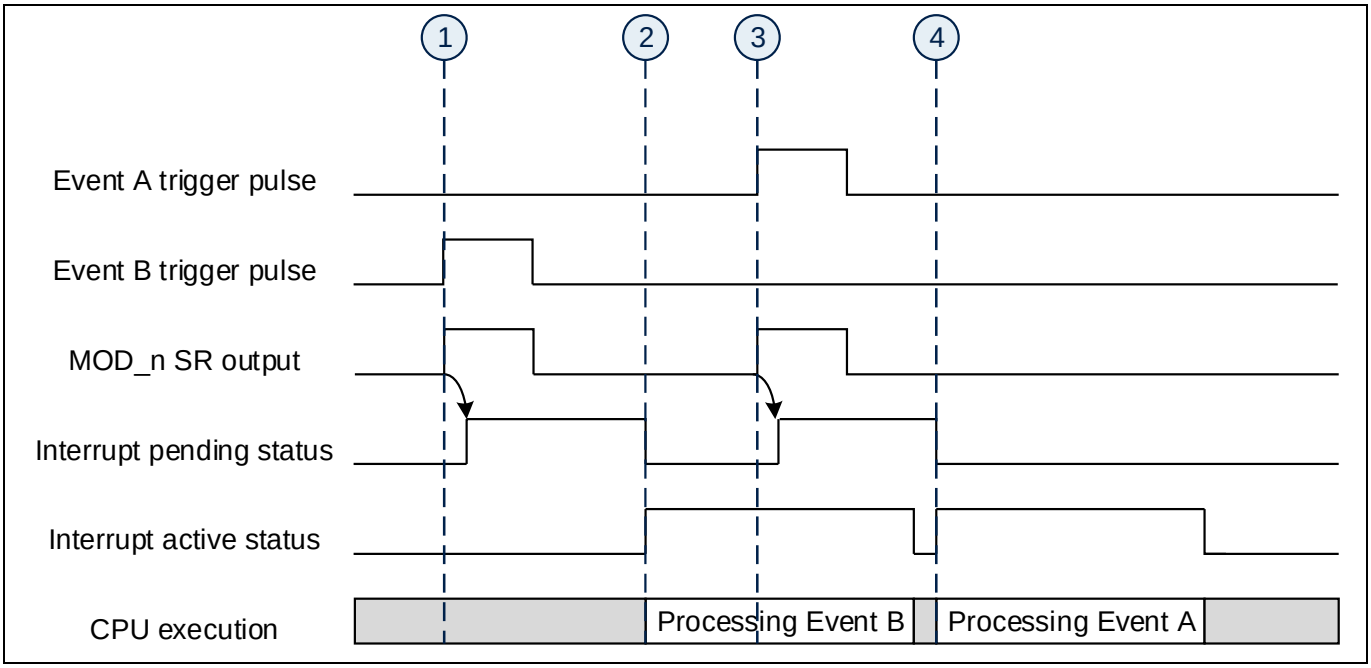

Figure 4 **Event A becomes pending while servicing interrupt B**

3 Interrupt response time

The interrupt response time is defined as the time between the occurrence of an interrupt request event to the execution of the first instruction in the interrupt handler.

In an XMC™ device, the interrupt response time consists of two components as shown in Figure 5:

- Peripheral interrupt generation delay
- Core interrupt latency

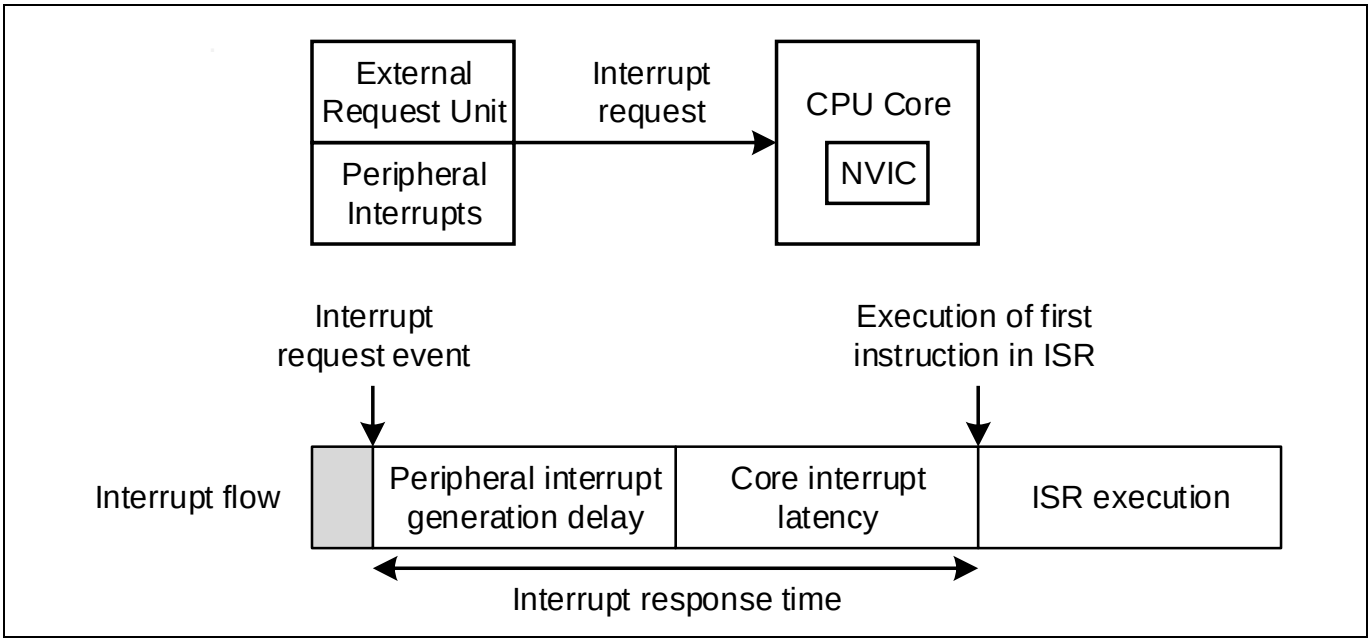

Figure 5 Interrupt flow

3.1 Peripheral interrupt generation delay

Interrupt requests may be triggered either by the on-chip peripherals, or by external inputs through the External Request Unit (ERU).

Peripheral interrupt generation delay refers to the time between the occurrence of an interrupt event to the generation of the interrupt request to NVIC.

This delay is highly dependent on the peripheral itself and also the nature of the interrupt event. For example, an external pin interrupt, which involves edge detection of the pin, is likely to incur more delay compared to an internally generated Capture/Compare Unit 4 (CCU4) period match interrupt.

3.2 Core interrupt latency

Core interrupt latency refers to the time from detection of the interrupt request by NVIC to execution of the first instruction at the interrupt handler.

This latency can be extended due to memory wait states or during interrupt pre-emption. Conversely, in the event of late-arrival or tail-chaining interrupts, the latency will be less.

Excluding the above conditions, the latency in terms of CPU clock cycles is:

- 21 cycles for XMC1000
- 12 cycles for XMC4000

3.3 Improving interrupt performance

There are two methods to improve the overall interrupt performance:

- Enabling the compiler optimization option
- Assigning the Interrupt handler to SRAM

Though these methods do not directly reduce the interrupt response time, which is largely dependent on the hardware - CPU core and peripherals, they improve the interrupt performance through faster code execution of the ISR.

3.3.1 Enabling compiler optimization option

An IDE tool, such as DAVE™4, supports several optimization levels. Selection is through the optimization option in the active project's properties. By default, optimization is turned off.

Enabling the optimization option tells the compiler to generate a more optimized code, which can result in the faster execution of the ISR as shown in Section 3.3.3.

In general, the user can experiment with the different optimization levels to find the one that best fits their application.

Figure 6 shows an active project's properties window in DAVE™4.

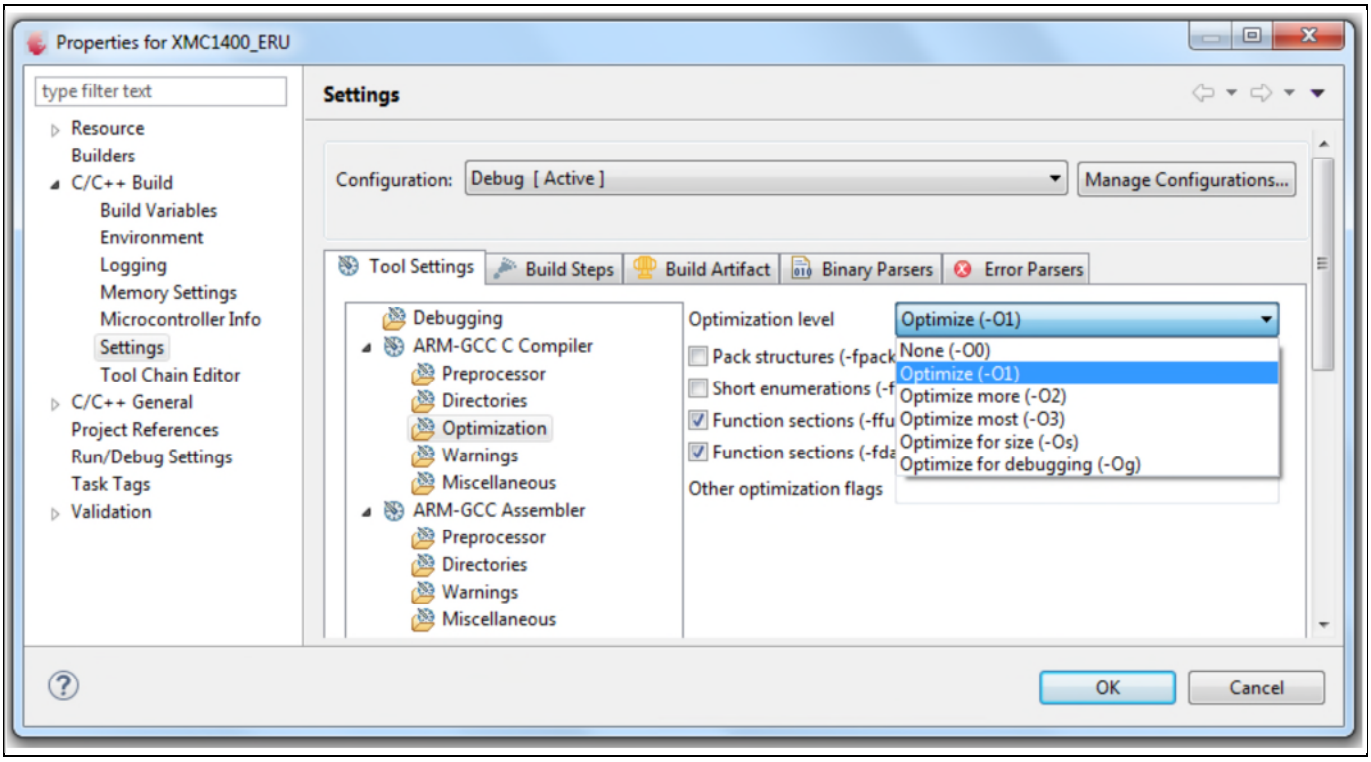

Figure 6 DAVE™4 project properties

3.3.2 Assigning interrupt handler to SRAM

By default, interrupt handlers are assigned to the Flash memory during compilation of the code by the IDE tool. Read accesses to flash memory incur additional wait states unless the device is operating at a very slow clock frequency.

To avoid the performance penalty due to wait states, the interrupt handler can be assigned to the SRAM (PSRAM in XMC4000), which has no wait state.

The linker script file in DAVE™4 defines a dedicated SRAM section (".ram_code") for time-critical code, which can also be used by the interrupt handler.

Therefore, assigning an interrupt handler to SRAM requires only the step to add an interrupt handler declaration with the section attribute as shown in the example code below:

```
/* Interrupt handler declaration */
void IRQ3_Handler(void) __attribute__ ((section (".ram_code")));
/* Interrupt handler definition */
void IRQ3_Handler(void)
{
      …
}
```

3.3.3 Interrupt performance comparison

To illustrate the effect of the two methods, consider the following example use case with an external pin interrupt.

The ERU is used to trigger an interrupt request upon the detection of a rising edge on a defined pin. The very first instruction in the ERU interrupt handler resets the same pin to zero.

For the purpose of this illustration, the interrupt performance is simply measured by the width of the resulting pulse generated on the pin. This includes the ERU interrupt generation delay, the core interrupt latency and the execution time of the instruction to reset the pin.

The results shown in Table 2 are based on an XMC1400 device operating at the maximum frequency of 48 MHz.

Table 2 Width of generated pulse based on XMC1400

Interrupt handler location	Code optimization	Width of generated pulse (CPU clock cycles)
Flash	Off	44
Flash	On (optimization level '-O1')	40
SRAM	Off	37
SRAM	On (optimization level '-O1')	34

4 References

[1] XMC1000 Reference Manual at http://www.infineon.com/xmc1000 Tab: Documents

[2] XMC4000 Reference Manual at http://www.infineon.com/xmc4000 Tab: Documents

Revision history

Major changes since the last revision

Page or reference	Description of change
-	First version V1.0

Edition 2016-04-22

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference
AP32331

IMPORTANT NOTICE

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

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

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (www.infineon.com).

WARNINGS

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

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