XMC1000

Microcontroller Series
for Industrial Applications

Interrupt Subsystem

✓ Using Interrupts

Device Guide

V1.0 2013-04

Microcontrollers

Edition 2013-04

Published by
Infineon Technologies AG
81726 Munich, Germany

Legal Disclaimer

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics. With respect to any examples or hints given herein, any typical values stated herein and/or any information regarding the application of the device, 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.

Information

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

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office.

Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

Revision History	
Page or Item	**Subjects (major changes since previous revision)**
V1.0, 2013-04	

Trademarks of Infineon Technologies AG

AURIX™, C166™, CanPAK™, CIPOS™, CIPURSE™, EconoPACK™, CoolMOS™, CoolSET™, CORECONTROL™, CROSSAVE™, DAVE™, EasyPIM™, EconoBRIDGE™, EconoDUAL™, EconoPIM™, EiceDRIVER™, eupec™, FCOS™, HITFET™, HybridPACK™, I²RF™, ISOFACE™, IsoPACK™, MIPAQ™, ModSTACK™, my-d™, NovalithIC™, OptiMOS™, ORIGA™, PRIMARION™, PrimePACK™, PrimeSTACK™, PRO-SIL™, PROFET™, RASIC™, ReverSave™, SatRIC™, SIEGET™, SINDRION™, SIPMOS™, SmartLEWIS™, SOLID FLASH™, TEMPFET™, thinQ!™, TRENCHSTOP™, TriCore™.

Other Trademarks

Advance Design System™ (ADS) of Agilent Technologies, AMBA™, ARM™, MULTI-ICE™, KEIL™, PRIMECELL™, REALVIEW™, THUMB™, µVision™ of ARM Limited, UK. AUTOSAR™ is licensed by AUTOSAR development partnership. Bluetooth™ of Bluetooth SIG Inc. CAT-iq™ of DECT Forum. COLOSSUS™, FirstGPS™ of Trimble Navigation Ltd. EMV™ of EMVCo, LLC (Visa Holdings Inc.). EPCOS™ of Epcos AG. FLEXGO™ of Microsoft Corporation. FlexRay™ is licensed by FlexRay Consortium. HYPERTERMINAL™ of Hilgraeve Incorporated. IEC™ of Commission Electrotechnique Internationale. IrDA™ of Infrared Data Association Corporation. ISO™ of INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. MATLAB™ of MathWorks, Inc. MAXIM™ of Maxim Integrated Products, Inc. MICROTEC™, NUCLEUS™ of Mentor Graphics Corporation. Mifare™ of NXP. MIPI™ of MIPI Alliance, Inc. MIPS™ of MIPS Technologies, Inc., USA. muRata™ of MURATA MANUFACTURING CO., MICROWAVE OFFICE™ (MWO) of Applied Wave Research Inc., OmniVision™ of OmniVision Technologies, Inc. Openwave™ Openwave Systems Inc. RED HAT™ Red Hat, Inc. RFMD™ RF Micro Devices, Inc. SIRIUS™ of Sirius Satellite Radio Inc. SOLARIS™ of Sun Microsystems, Inc. SPANSION™ of Spansion LLC Ltd. Symbian™ of Symbian Software Limited. TAIYO YUDEN™ of Taiyo Yuden Co. TEAKLITE™ of CEVA, Inc. TEKTRONIX™ of Tektronix Inc. TOKO™ of TOKO KABUSHIKI KAISHA TA. UNIX™ of X/Open Company Limited. VERILOG™, PALLADIUM™ of Cadence Design Systems, Inc. VLYNQ™ of Texas Instruments Incorporated. VXWORKS™, WIND RIVER™ of WIND RIVER SYSTEMS, INC. ZETEX™ of Diodes Zetex Limited.

Last Trademarks Update 2011-02-24

Table of Contents

Using Interrupts

1 Using Interrupts

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

Setting up an interrupt in a XMC1000 system involves the following steps:

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

1.1 Configure the Interrupt Priority Level

This step is optional.

An interrupt node can be configured to one of four priority levels. The levels are in steps of 64, from 0 to 192, with level 0 having the highest priority. By default, all nodes are configured to level 0.

To change the interrupt priority level, use the CMSIS function NVIC_SetPriority(). For example, to configure node 1, which is connected to SCU service request output 1 (SCU_SR1), to level 192, the following code can be used:

```
//This CMSIS function configures node 1 to priority level 192
NVIC_SetPriority(SCU_1_IRQn, 0xC0);
```

SCU_1_IRQn is the enumerated constant for node 1. The complete list of enumerated constants can be found in the device header file “XMC1x00.h”.

1.2 Define an Interrupt Service Routine (ISR)

An interrupt node may contain more than one interrupt trigger source. For example, 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_XMC1x00.s”.
- Check the status flag of the highest priority event 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.

Note: The startup file handles the interrupt vector address and the method of pointing to the actual interrupt handler location.

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(0U != (SCU_INTERRUPT->SRRAW & SCU_INTERRUPT_SRRAW_SBYCLKFI_Msk))
      {
            //Write to SRCLR to clear active flag
            SCU_INTERRUPT->SRCLR |= SCU_INTERRUPT_SRCLR_SBYCLKFI_Msk;
            … //Process the interrupt
      }
      //Check if VDDP pre-warning event status flag is set
      if (0U != (SCU_INTERRUPT->SRRAW & SCU_INTERRUPT_SRRAW_VDDPI_Msk))
      {
            //Write to SRCLR to clear active flag
```

```
            SCU_INTERRUPT->SRCLR |= SCU_INTERRUPT_SRCLR_VDDPI_Msk;
            … //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™ 3 and µKeil Vision, during a software build.

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, a system bus stall will occur. CPU execution of the interrupt handler will only occur when the Flash is available again.

To avoid the above scenario, interrupt handlers can be relocated to the SRAM. For information on relocating code to SRAM, refer to the respective IDE tool documentation.

1.3 Enable the Interrupt

To generate an interrupt request to 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 (optional)
- 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), 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 to 0
SCU_INTERRUPT->SRCLR |= SCU_INTERRUPT_SRCLR_SBYCLKFI_Msk;
SCU_INTERRUPT->SRCLR |= SCU_INTERRUPT_SRCLR_VDDPI_Msk;
//Enables SBYCLKFI and VDDPI interrupt trigger sources
SCU_INTERRUPT->SRMSK |= SCU_INTERRUPT_SRMSK_SBYCLKFI_Msk;
SCU_INTERRUPT->SRMSK |= SCU_INTERRUPT_SRMSK_VDDPI_Msk;
```

1.4 Processing an 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 can generate an interrupt request. 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 1 shows the case where events A and B occur before the interrupt is serviced. The time instances 1 to 4 are:

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. 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.
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.

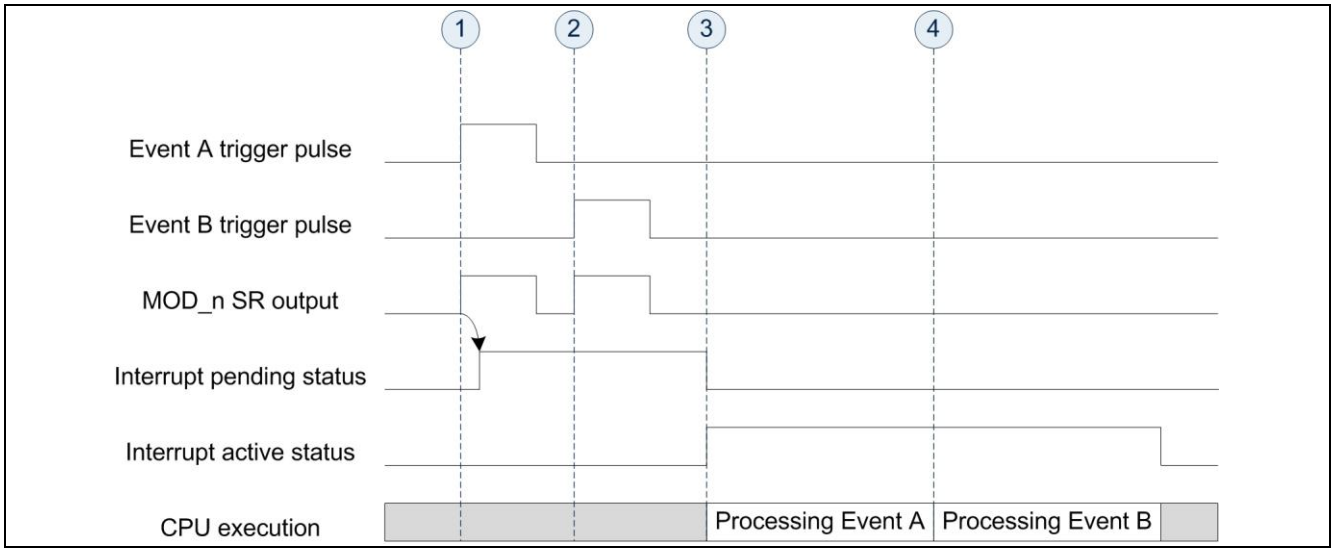

Figure 1 Events A and B occur before interrupt is serviced

Event B occurs while servicing interrupt event A

Figure 2 shows the case where event B becomes pending while the CPU is still servicing the interrupt triggered by event A. The time instances 1 to 5 are:

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.

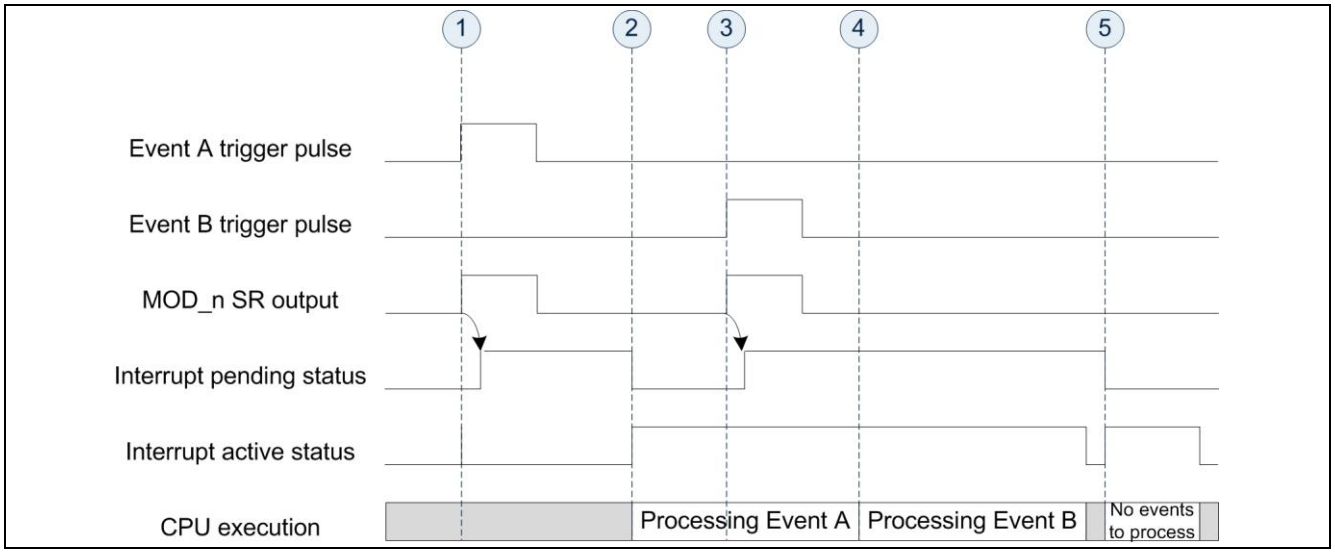

Figure 2 Event B becomes pending while servicing interrupt A

Event A occurs while servicing interrupt event B

Figure 3 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:

1. Event B 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 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.

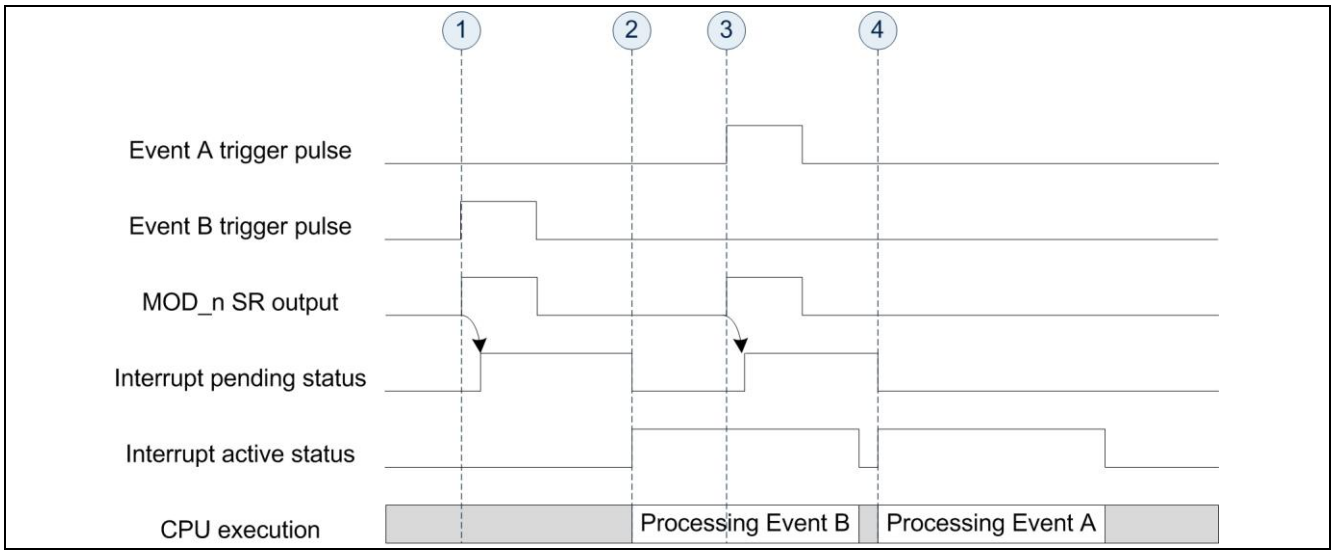

Figure 3 Event A becomes pending while servicing interrupt B

w w w . i n f i n e o n . c o m

Published by Infineon Technologies AG