

Future Technology Multicore

Single Handler Interrupt Implementation for TC297B

Integration Guide  
(Revision A)

**Author(s):**  
Huy Dinh

**Release Date:**  
 06.07.2018

**ABSTRACT**

Interrupts are hardware signals towards a CPU, which break the current thread and execute a handler functions specific to a given interrupt. Interrupts enable the fast response of a CPU to external events like GPIO state changes or a “conversion ready” event from and ADC. In this project, an interrupt driver for the Infineon AURIX multicore was refinded.

The implementation of the interrupt system at the time this document was created uses a multi-entry interrupt vector table stored in the AURIX ROM. Although this is one of the original implementations suggested by Infineon, this method of handling interrupts is complex and requires a fairly large portion of the ROM memory to be allocated for the vector table. This calls for an implementation where both the level of complexity and the amount of ROM usage can be reduced.

This document introduces the integration of another interrupt handling approach, where only one interrupt vector entry is needed. This entry will be entered for all interrupt requests, calling a dispatching function which will, in turn, forward the interrupt to the corresponding handler functions registered beforehand as callback functions. The logics behind the interrupt signal flows are clearly visible in the dispatching function and the amount of ROM usage is effectively reduced, since there is only one entry in the interrupt vector table. Also introduced is an interrupt configuration method that removes the need to manually manage interrupt priorities using configuration tables for both pre-OS and OS runtime.

LIST OF CONTENTS

[1 Principle of operation 1](#_Toc516093961)

[1.1 Overview 1](#_Toc516093962)

[1.2 Hardware for Interrupts in the AURIX 1](#_Toc516093963)

[1.3 Interrupt Handling in the AURIX CPU 1](#_Toc516093964)

[1.4 Single-entry interrupt vector table 2](#_Toc516093965)

[2 Driver implementation 3](#_Toc516093966)

[2.1 Pre-OS Interrupt implementation 3](#_Toc516093967)

[2.1.1 Single-entry interrupt vector table 3](#_Toc516093968)

[2.1.2 Interrupt handler table 3](#_Toc516093969)

[2.1.3 Common Dispatcher 3](#_Toc516093970)

[2.2 OS runtime 4](#_Toc516093971)

[2.2.1 Interrupt handler table 4](#_Toc516093972)

[3 Driver API 5](#_Toc516093973)

[3.1 RC\_t INT\_preOsSetConfig(INT\_preOsIsrEntry\_t\* pTable); 5](#_Toc516093974)

[3.2 RC\_t INT\_preOsInstallIsrTable(); 5](#_Toc516093975)

[3.3 RC\_t INT\_preOsStop(); 5](#_Toc516093976)

[3.4 RC\_t INT\_osSetConfig(INT\_osIsrEntry\_t\* pTable); 5](#_Toc516093977)

[3.5 RC\_t INT\_osInstallIsrTable(INT\_osIsrTaskId\_t taskId, PxArg\_t argument); 5](#_Toc516093978)

[3.6 RC\_t INT\_osInstall(volatile Ifx\_SRC\_SRCR\* serviceRequestNode, INT\_isr\_t pIsr, INT\_handler\_t handlerType, PxArg\_t argument); 6](#_Toc516093979)

[3.7 RC\_t INT\_osUnInstall(); 6](#_Toc516093980)

[4 Integration into the overall system 7](#_Toc516093981)

[4.1 Integration and configuration of code files 7](#_Toc516093982)

[4.1.1 Folders to be created 7](#_Toc516093983)

[4.1.2 Files to be copied 7](#_Toc516093984)

[4.1.3 Modify the main linker description file main.ld 7](#_Toc516093985)

[4.1.4 Modify CPU initialization configuration inside core\_startup.c. 7](#_Toc516093986)

[4.2 Usage 8](#_Toc516093987)

[4.2.1 Pre-OS 8](#_Toc516093988)

[4.2.2 During OS runtime 9](#_Toc516093989)

[5 Open issues 11](#_Toc516093990)

[6 Changelog 12](#_Toc516093991)

[7 List of abbreviations 13](#_Toc516093992)

[8 list of literature 14](#_Toc516093993)

# Principle of operation

## Overview

An interrupt is a signal sent by a hardware or software module to a CPU/DMA module to stop current operations and either start an Interrupt Service Routine or a DMA service.

The AURIX typically uses a multi-entry interrupt handler vector table in its interrupts handling implementation.

The integration presented in this document allows the interrupt system on the AURIX to run with only one interrupt vector entry located on ROM.

## Hardware for Interrupts in the AURIX

Every interrupt source in the AURIX has one or more Service Request Node (SRN). Each SRN is connected to all Interrupt Control Units (ICU) and has a Service Request Control register that defines basic properties like priority and service provider… An ICU is connected to one service provider (either a CPU or a DMA module). The service provider is on the receiving end and works to handle the interrupt signal.[[1]](#footnote-1)

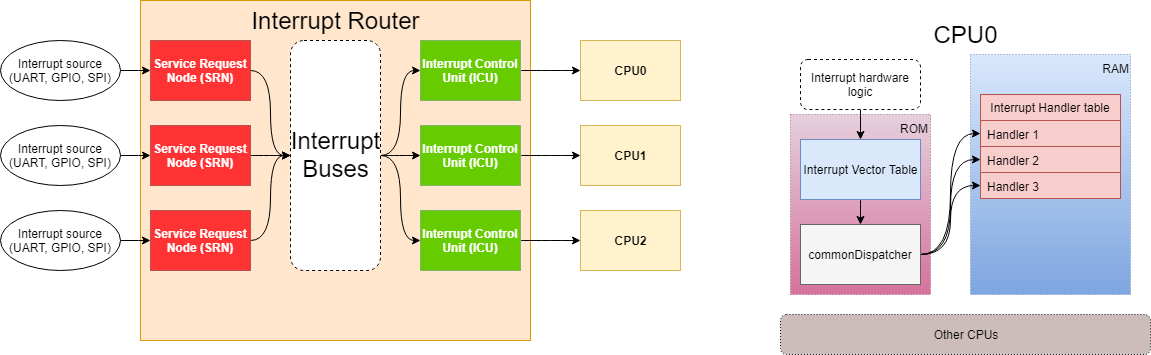


Figure AURIX Interrupt hardware overview

Interrupt arbitration: ICU manages the interrupt system and arbitrates incoming interrupt requests to find the one with the highest priority and to determine whether or not to interrupt the service provider. The interrupt with the highest priority in one arbitration process will be the winning interrupt.

## Interrupt Handling in the AURIX CPU

There is an interrupt vector table stored in ROM, each entry of this table contains a handler for a corresponding interrupt of a certain priority. The Base Interrupt Vector (BIV) register should contain the starting address of this interrupt vector and is used to calculate the table entry point of an interrupt request.

When a CPU receives an interrupt request, it will perform the following preparation steps:[[2]](#footnote-2)

* Save the upper context, update A[11] (Return address) with the current program counter.
* If not using interrupt stack, set A[10] Stack pointer to ISP (Interrupt Stack Pointer) (PSW.IS = 1)
* Set I/O mode to Supervisor (PSW.IO = 0b10), enable memory protection using the interrupt memory protection map (PSW.PRS = 0b00), clear Call Depth Counter (PSW.CDC).
* Disable write to A[0], A[1], A[8] and A[9] (PSW.GW = 0)
* Disable global interrupts.
* Save CCPN (Current Priority Number) to Previous CPU Priority Number (PCPN)
* Save PIPN (Pending Interrupt Priority Number) to CCPN (Current CPU Priority Number)
* Entry address of the interrupt vector table is calculated and entered

At the last step, the entry address is calculated by OR’ing the interrupt masking region on the BIV register with PIPN, the result will be the address where the program would enter on the interrupt vector table.[[3]](#footnote-3)

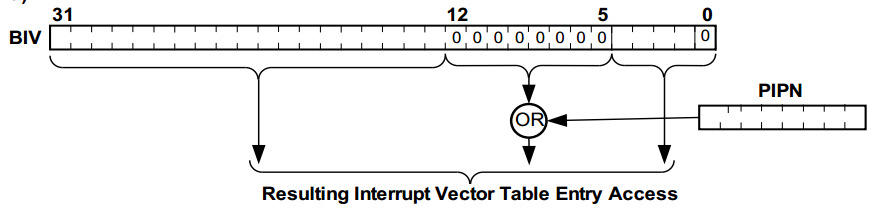


Figure 2 Calculation of the entry address on the interrupt vector table

The entered entry should handle the interrupt and jump back to the re-entry point in the program.

## Single-entry interrupt vector table

By filling the 8-bit interrupt masking region on the BIV register with 1’s, the resulting entry address will always be the same value that BIV is holding regardless of the value stored on PIPN[[4]](#footnote-4). That means interrupt requests from all priorities are directed to the same handler whose address is stored in BIV.

# Driver implementation

## Pre-OS Interrupt implementation

There are 3 main software components inside this pre-OS interrupt implementation: the interrupt vector table (located on Flash), the interrupt handler table (on RAM) and a common dispatcher function. The interrupt vector table has only one entry, which will be called every time an interrupt is received by the CPU. This entry will then call the common dispatcher function to dispatch the interrupt to its suitable handler by reading the interrupt priority.

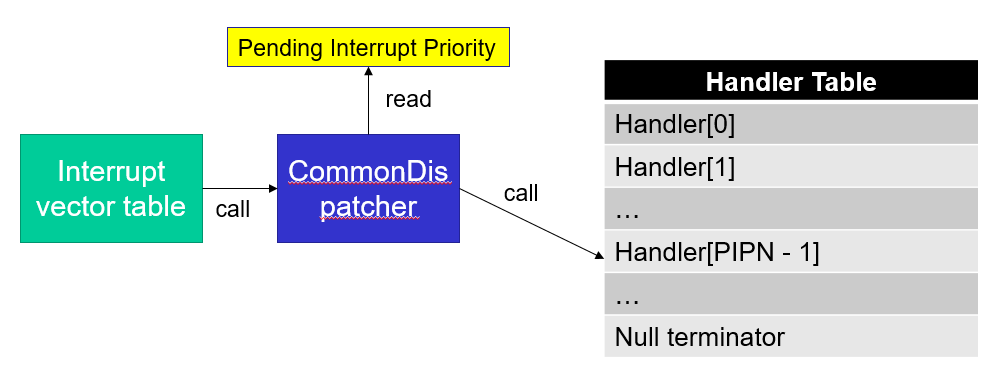


Figure 3 The relationship between the vector table, the common dispatcher and the handler table during pre-OS runtime

### Single-entry interrupt vector table

This implementation of interrupt handling system relies on the interrupt vector table having only one entry that is entered every time the CPU receives an interrupt request. In order for this to happen, the interrupt vector table is located at 0x80003FE0, meaning the interrupt masking region has a value of 255.

This interrupt vector table always calls the Common Dispatcher.

### Interrupt handler table

The interrupt handler table contains all the interrupt handlers whose indices are also the priority of the interrupts they service with an offset of 1 (since the interrupt with priority 0 are reserved). This table is located on RAM and could be found in **Interrupts\_cfg.cpp** as an array of **INT\_preOsIsrEntry\_t** named **INT\_preOsTable**.

Each entry (of type INT\_preOsIsrEntry\_t) contains a pointer to the SRC register of the interrupt source, the interrupt handling function and the service provider (which cpu/dma module should run the interrupt handling function).

### Common Dispatcher

The common dispatcher is a function named commonDispatcher. This function is always called by the interrupt vector table and in turn calls the corresponding handlers inside the interrupt handler table based on the pending interrupt priority.

## OS runtime

Once the operating system (PXROS) is initialized, it takes over the handling of interrupts by moving the interrupt vector table to its own implementation. That renders the common dispatcher and the pre-OS interrupt handler table obsolete.

### Interrupt handler table

As interrupts are already handled by the OS, the OS-runtime interrupt handler table is only used for interrupt configuration/registration. The indices of the entries are also the priority of the corresponding interrupts with an offset of 2 (interrupts with priorities 1 and 0 are reserved). This table is located on RAM and could be found in **Interrupts\_cfg.cpp** as an array of **INT\_osIsrEntry\_t** named **INT\_osTable**.

Each entry (of type **INT\_osIsrEntry\_t**) contains a pointer to the SRC register of the interrupt source, the interrupt handling function and the service provider (which cpu/dma module should run the interrupt handling function) and the type of the handlers (defined by PXROS).

# Driver API

## RC\_t INT\_preOsSetConfig(INT\_preOsIsrEntry\_t\* pTable);

This function is used to select the interrupt handler table to be used for pre-OS runtime. Normally the table to be used should be INT\_preOsTable in Interrupts\_cfg.cpp.

Parameter pTable: pointer to the table to be selected.

Return: RC\_SUCCESS if the operation succeeds, RC\_ERROR\_INVALID\_STATE if the table has already been selected.

## RC\_t INT\_preOsInstallIsrTable();

This function configures the Service Request Nodes specified in the selected interrupt handler table and enable interrupts, effectively installing and enabling pre-OS interrupts.

Return: RC\_SUCCESS if the installation is successful, RC\_ERROR\_NULL if no tables have been selected, RC\_ERROR\_OVERRUN if the table is either not null terminated or too big.

Note: INT\_preOsSetConfig may need to be called beforehand.

## RC\_t INT\_preOsStop();

This functions clears the Service Request Control registers and the interrupt handler table pointer (deselect the table)

Return: RC\_SUCCESS if the operation succeeds, RC\_ERROR\_OVERRUN if the table is either not null terminated or too big.

## RC\_t INT\_osSetConfig(INT\_osIsrEntry\_t\* pTable);

This function selects the interrupt handler table to be used for OS runtime. Normally the table to be used should be INT\_osTable.

Parameter pTable: pointer to the table to be selected.

Return: RC\_SUCCESS if the operation succeeds, RC\_ERROR\_INVALID\_STATE if the table has already been selected.

## RC\_t INT\_osInstallIsrTable(INT\_osIsrTaskId\_t taskId, PxArg\_t argument);

This function configures the Service Request Nodes specified in the selected interrupt handler table and enable interrupts, effectively installing and enabling OS interrupts defined in the table.

Parameter taskId: The task identifier of the task in charge of the interrupt, this should be the same as the task calling this function.

Parameter argument: the argument to be passed to all of the interrupt handlers associated with this task.

Return: RC\_SUCCESS if the installation is successful, RC\_ERROR\_NULL if no tables have been selected, RC\_ERROR\_OVERRUN if the table is either not null terminated or too big.

Note: This function must be called in a task context. INT\_osSetConfig may need to be called beforehand

## RC\_t INT\_osInstall(volatile Ifx\_SRC\_SRCR\* serviceRequestNode, INT\_isr\_t pIsr, INT\_handler\_t handlerType, PxArg\_t argument);

This function configures the service request node and interrupt vector table in a task context.

Parameter serviceRequestNode: the service request node (interrupt source) which needs to be configured.

Parameter pIsr: pointer to the handler function

Parameter handlerType: type of the handler (Check the INT\_handler\_t enum for details)

Return: RC\_SUCCESS of the operation succeeds, RC\_ERROR\_NULL if no tables have been selected, RC\_ERROR in case PXROS handler installation returns an error code, RC\_ERROR\_BAD\_PARAM if SRC has not been found in the table.

Note: This function will be called by INT\_osInstallIsrTable and is usually not called from user side, unless a ISR should be reconfigured during runtime.

## RC\_t INT\_osUnInstall();

This function uninstalls the installed interrupts for OS runtime. Currently not implemented.

# Integration into the overall system

## Integration and configuration of code files

### Folders to be created

These folders need to be created in these respective locations:

* Interrupts in /src\_BSW/drv
* Interrupts in /src\_BSW/drv\_config

### Files to be copied

These files need to be copied to the project in these respective folders:

* Interrupts.c to /src\_BSW/drv/Interrupts
* Interrupts.h to /src\_BSW/drv/Interrupts
* Interrupts\_cfg.cpp to /src\_BSW/drv\_config/Interrupts
* Interrupts\_cfg.h to /src\_BSQ/drv\_config/Interrupts

### Modify the main linker description file main.ld

The FLASH memory section for the interrupt vector table needs to be defined by modifying the linker description file main.ld located in **/ld/**

The section below needs to be added to the SECTIONS block in the file, any overlap with other sections needs to be avoided.

.interrupttable\_init\_out (0x80003FE0) : /\*0x80003FE0 \*/

{

\_\_INIT\_INT\_TAB\_BEGIN = . ;

KEEP (\*(.interrupttable\_init\_in));

. = ALIGN(8) ;

\_\_INIT\_INT\_TAB\_END = . ;

} > mem\_flash0}

Snippet 1 The section to be added in main.ld

### Modify CPU initialization configuration inside core\_startup.c.

Add the bolded line below to the declaration section of **core\_startup.c** in /src\_BSW/startup so that the symbol \_\_INIT\_INT\_TAB\_BEGIN is visible within the scope of the file

EXTERN \_FARDATA uint32\_t PXROS\_SYSTEM\_STACK\_CPU1\_;

EXTERN \_FARDATA uint32\_t PXROS\_SYSTEM\_STACK\_CPU2\_;

EXTERN \_FARDATA uint32\_t \_\_INT\_TAB\_BEGIN;

**EXTERN \_FARDATA uint32\_t \_\_INIT\_INT\_TAB\_BEGIN; //Add this line**

EXTERN \_FARDATA uint32\_t \_\_INIT\_TRAPTAB\_BEGIN;

Snippet 2 declaring \_\_INIT\_INT\_TAB\_BEGIN inside core\_startup.c

Then configure CPU0 to use this address as the starting address for its table interrupt by making sure the first **CoreInit\_t** struct inside the array **CPUInit[3]** has **&\_\_INIT\_INT\_TAB\_BEGIN** as its **\*InterruptTable**. This is done by modifying the initialization of this struct like below (bolded line)

/\* CPU configuration Table \*/

\_FARDATA **const** CoreInit\_t CPUInit[3] = {

{

&\_\_clear\_table\_CPU0\_,

&\_\_copy\_table\_CPU0\_,

&PXROS\_SYSTEM\_STACK\_CPU0\_,

&PXROS\_SYSTEM\_STACK\_CPU0\_,

&\_SMALL\_DATA\_,

&\_SMALL\_DATA2\_,

&\_SMALL\_DATA3\_,

&\_SMALL\_DATA4\_CPU0\_,

(uint32\_t)\_\_CSA\_BEGIN,

(uint32\_t)\_\_CSA\_SIZE,

**&\_\_INIT\_INT\_TAB\_BEGIN, //Make sure this is correct**

&\_\_INIT\_TRAPTAB\_BEGIN,

&SCU\_WDTCPU0CON0,

&SCU\_WDTCPU0CON1

},

Snippet 3 Adding \_\_INIT\_INT\_TAB\_BEGIN as interrupt table for CPU0

## Usage

### Pre-OS

To use the interrupt system, the user first needs to define ISR functions that return void and takes a PxArg\_t as parameter. PxArg\_t is defined in pxdef.h.

**void** **RX\_UART\_RX\_Isr**(PxArg\_t inputArg)

{

uint8\_t readByte;

// Clear interrupt flag

MODULE\_ASCLIN3.FLAGSCLEAR.B.RFLC = 1;

// Read the data from the FIFO

UART\_ReadData(*uart6*, \*readByte);

}

Snippet 4 sample UART RX ISR definition

Then, in **Interrupt\_cfg.cpp**, the ISR functions need to be added to the pre-OS interrupt handler table (named **INT\_preOsTable**) with the correct configuration. Note that an extern declaration of the function inside **Interrupt\_cfg.cpp** might be needed so that the function is visible within this file.

**extern** **void** **RX\_UART\_RX\_Isr**(PxArg\_t inputArg);

Snippet extern declaration of RX\_UART\_RX\_Isr inside Interrupt\_cfg.cpp

Add the handler along with the corresponding SRC (Interrupt source) and the service provider (CPU) to an entry before the null terminator. The index of the entry defines the priority of the interrupt with an offset of 1 (For example: interrupt whose entry is at index 0 has a priority of 1).

**const** INT\_preOsIsrEntry\_t INT\_preOsTable[] = {

//Low priority

{&SRC\_ASCLIN3RX, &RX\_UART\_RX\_Isr, *cpu0* },

//High Priority

{0,0,(CpuId\_t)0} // Null terminator – end of table

};

Snippet RX\_UART\_RX\_Isr configured for SRC\_ASCLIN3RX handled by cpu0

This table then needs to be registered and its entries’ hardware enabled by making calls to INT\_preOsSetConfig and INT\_preOsInstallIsrTable.

//Select INT\_preOSTable as the pre-OS handler table

INT\_preOsSetConfig(INT\_preOsTable);

//Install the selected tables and enable preOS interrupts

INT\_preOsInstallIsrTable();

Snippet Selecting and installing a pre-OS handler table

Pre-OS interrupts should now be ready to be used.

### During OS runtime

Before OS interrupts can be configured, it is recommended to reset the hardware/software configuration done during pre-OS. This can be done by calling INT\_preOsStop.

Since during OS runtime, an interrupt has to be associated with a certain task, the driver needs an identifier for each interrupt-enabled task. These identifiers need to be defined inside the enum INT\_osIsrTaskId\_t found in Interrupts\_cfg.h. Add the identifiers for all interrupt-enabled tasks here. Note that these identifiers do not need to be the same as the PXROS task IDs of the corresponding tasks.

**typedef** **enum** {

*INT\_TID\_C0\_INIT*,

*INT\_TID\_C0\_COMMUNICATION*,

*INT\_TID\_C2\_COMMUNICATION*,

*INT\_TID\_C2\_TFT*,

} INT\_osIsrTaskId\_t ;

Snippet Sample task identifiers inside the enum INT\_osIsrTaskId\_t

Similar to pre-OS, handler functions (returns void, takes a PxArg\_t as parameter) will need to be declared and defined, then have an extern declaration inside **Interrupts\_cfg.cpp**

Entries for the interrupts need to be added into the OS interrupt handler table (named **INT\_osTable**, found inside **Interrupts\_cfg.cpp**) before the null terminator. The index of an interrupt entry defines the priority of that interrupt with an offset of 2 (Interrupt at index 0 will have priority 2). Each entry must contain the handler function, the SRC, the handler type and the task identifier.

**extern** **void** **CAN\_RxISR**(PxArg\_t arg);

**extern** **void** **CAN\_TxISR**(PxArg\_t arg);

**extern** **void** **QSPI0\_txISR**(PxArg\_t arg);

**extern** **void** **QSPI0\_ptISR**(PxArg\_t arg);

**extern** **void** **QSPI0\_errorISR**(PxArg\_t arg);

**const** INT\_osIsrEntry\_t INT\_osTable[] = {

//Low Priority

{&SRC\_CANINT0, &CAN\_TxISR, *INT\_FASTCONTEXTHANDLER*, *INT\_TID\_C0\_COMMUNICATION*},

{&SRC\_CANINT1, &CAN\_RxISR, *INT\_FASTCONTEXTHANDLER*, *INT\_TID\_C0\_COMMUNICATION* },

{&SRC\_QSPI0ERR, &QSPI0\_errorISR, *INT\_FASTCONTEXTHANDLER*, *INT\_TID\_C2\_TFT*},

{&SRC\_QSPI0TX, &QSPI0\_txISR, *INT\_FASTCONTEXTHANDLER*, *INT\_TID\_C2\_TFT*},

{&SRC\_QSPI0PT, &QSPI0\_ptISR, *INT\_FASTCONTEXTHANDLER*, *INT\_TID\_C2\_TFT*},

//High Priority

{0,0,(INT\_handler\_t)0,(INT\_osIsrTaskId\_t)0} //Null terminator

};

Snippet Sample extern declarations and entries inside the OS interrupt handler table

For more information regarding the handler type, read the description of the enum INT\_handler\_t.

Now select the table INT\_osTable as the OS task handler table by calling INT\_osSetConfig.

INT\_osSetConfig(INT\_osTable);

Now that the table is configured and selected, the interrupts need to be initialized by making calls to INT\_osInstallIsrTable. This function has to be called inside a task context and thus, needs to be called once in every task whose identifier is present in the OS interrupt handler table. For example:

INT\_osInstallIsrTable(*INT\_TID\_C0\_COMMUNICATION*, NULL);

Will need to be called inside task “Communication” and will install the interrupts associated with this task. Repeat the same process for other interrupt-enabled tasks.

# Open issues

# Changelog

# List of abbreviations

|  |  |  |
| --- | --- | --- |
| API | - | Application Programming Interface |
|  |  |  |
|  |  |  |

# list of literature

|  |  |
| --- | --- |
| [1] | Infineon, *TC29x B-Step User’s Manual v1.3,* 2014. |

**APPENDIX**

1. TC29x B-Step User’s Manual (V1.3 2014-12) 16.1 Overview (page 1607) [↑](#footnote-ref-1)
2. TricoreTM User’s Manual (V1.0, 2012-05) chapter 5.3 Entering an Interrupt Service Routine (ISR) (page 68) [↑](#footnote-ref-2)
3. TC29x B-Step User’s Manual (V1.3 2014-12) Chapter 16.9.1 Use Case Example Interrupt Handler (page 1633) [↑](#footnote-ref-3)
4. TC29x B-Step User’s Manual (V1.3 2014-12) Chapter 16.9.1 Use Case Example Interrupt Handler (page 1633) [↑](#footnote-ref-4)