# How to use Serial Communications Block (SCB) in XMC7000 family

## About this document

### Scope and purpose

AN234127 demonstrates how to configure and use a Serial Communications Block (SCB) in XMC7000 family MCUs with three serial interface protocols: SPI, UART, and $I^2C$.

### Intended audience

This document is intended for anyone who uses the SCB feature in XMC7000 family MCUs.

### Associated part family

XMC7000 family XMC7100/XMC7200 series of XMC™ industrial microcontrollers

# Table of contents

**About this document** ..... **1**
**Table of contents** ..... **2**
**1 Introduction** ..... **3**
1.1 Features ..... 3
**2 General description** ..... **4**
**3 Common settings** ..... **5**
**4 SPI setting procedure example** ..... **6**
4.1 Master mode ..... 6
4.1.1 Configure ports ..... 8
4.1.2 Configure SPI interface registers ..... 9
4.1.3 Bit rate setting ..... 11
4.1.4 Use case ..... 11
4.2 Slave mode ..... 16
4.2.1 Configure ports ..... 19
4.2.2 Configure SPI interface registers ..... 19
4.2.3 Use case ..... 21
**5 UART setting procedure example** ..... **25**
5.1 UART mode ..... 25
5.1.1 Configure ports ..... 28
5.1.2 Configure UART interface registers ..... 28
5.1.3 Baud rate setting ..... 31
5.1.4 Use case ..... 31
**6 I²C setting procedure example** ..... **35**
6.1 Master mode ..... 35
6.1.1 Configure ports ..... 39
6.1.2 Configure I²C interface registers ..... 39
6.1.3 Bit rate setting ..... 41
6.1.4 Use case ..... 41
6.2 Slave mode ..... 46
6.2.1 Configure ports ..... 50
6.2.2 Configure I²C interface registers ..... 50
6.2.3 Use case ..... 52
**References** ..... **58**
**Glossary** ..... **59**
**Revision history** ..... **60**
**Disclaimer** ..... **61**

# 1 Introduction

This application note describes how to use a Serial Communications Block (SCB) in Infineon XMC7000 family XMC7100/XMC7200 series MCUs. The SCB is used for serial communication with other devices; it supports three serial communication protocols: SPI, UART, and $I^2C$.

This application note explains the functioning of the SCB, initial configuration, and data communication operations with use cases. To understand the functionality described and terminology used in this application note, see the “Serial Communications Block (SCB)” chapter of the architecture reference manual.

## 1.1 Features

The SCB supports the following features:

- Standard SPI Master and Slave functionality with Motorola, Texas Instruments, and National Semiconductor protocols
- Standard UART functionality with Smart Card reader and IrDA protocols
- Standard $I^2C$ Master and Slave functionality
- Only SCB[0] is DeepSleep-capable
- EZ mode for SPI and $I^2C$ Slaves allows for operation without CPU intervention
- CMD_RESP mode for SPI and $I^2C$ Slaves allows for operation without CPU intervention, and available only in DeepSleep-capable SCB
- Low-power (DeepSleep) mode of operation for SPI and $I^2C$ Slaves (using external clocking), only available on DeepSleep-capable SCB
- DeepSleep wakeup on $I^2C$ Slave address match or SPI Slave selection; only available on DeepSleep-capable SCB
- Trigger outputs for connection to DMA
- Multiple interrupt sources to indicate status of FIFOs and transfers

# 2 General description

The SCB supports three serial communication protocols: SPI, UART, and I²C. Only one of the protocols is supported by an SCB at any given time.

Figure 1 shows the block diagram of the SCB.

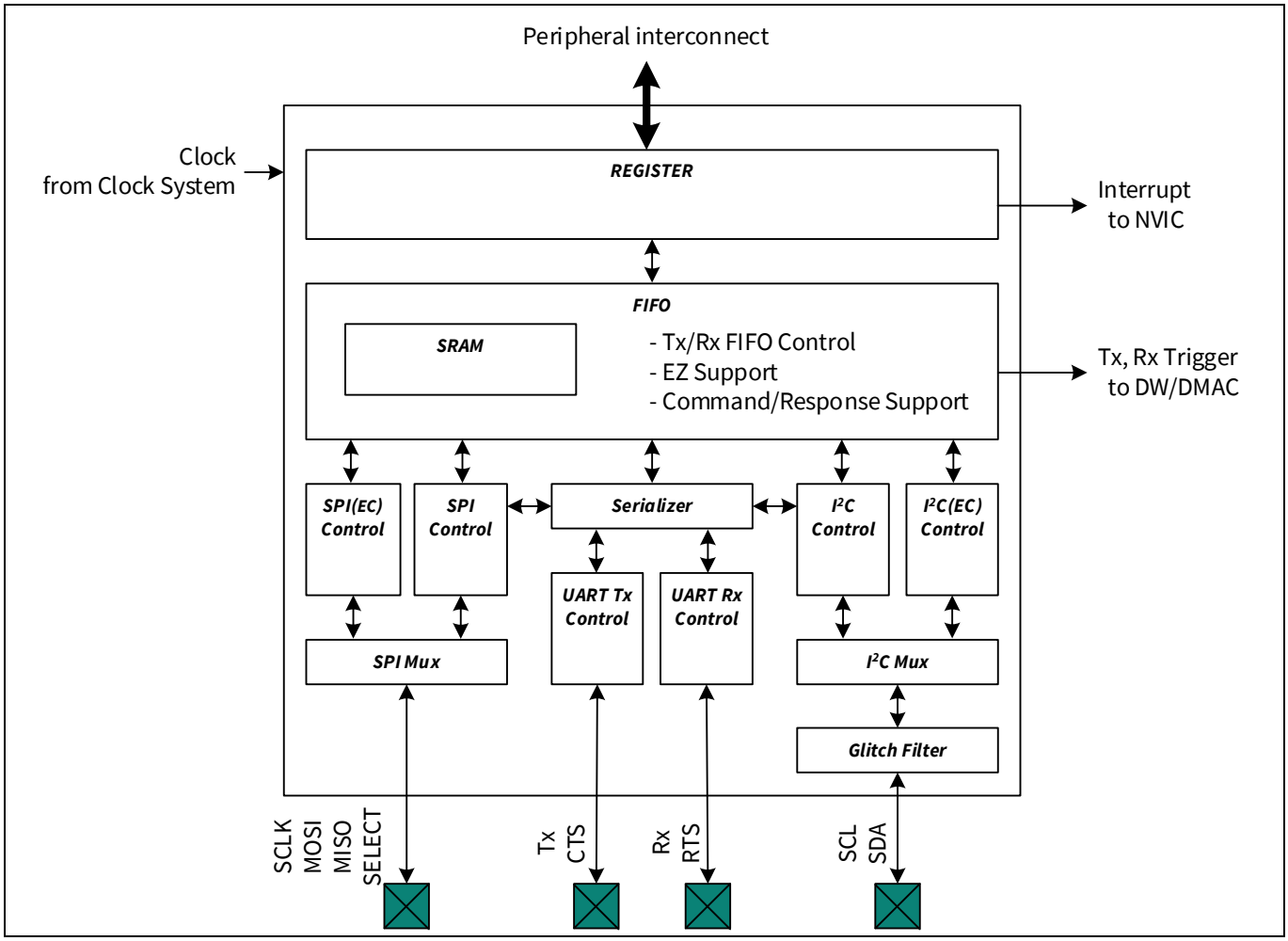


**Figure 1 SCB block diagram**

The SCB consists of registers, FIFO, and a control block for each protocol function (SPI, UART, and I²C). Registers are used as software interfaces for SCB settings and generated interrupts by each event. The FIFO consists of SRAM (256-byte) and has three modes Tx/Rx FIFO (128x8-bit/64x16-bit/32x32-bit), EZ (256x8-bit), and Command/Response (256x8-bit). Each protocol function control block works as a transmitting and receiving controller. SPI (all SCBs) and I²C (SCB[0]) support externally clocked (EC) mode in Slave mode.

# 3 Common settings

Figure 2 shows the flow of the general settings of the SCB. Specific settings for each protocol are described in later sections.

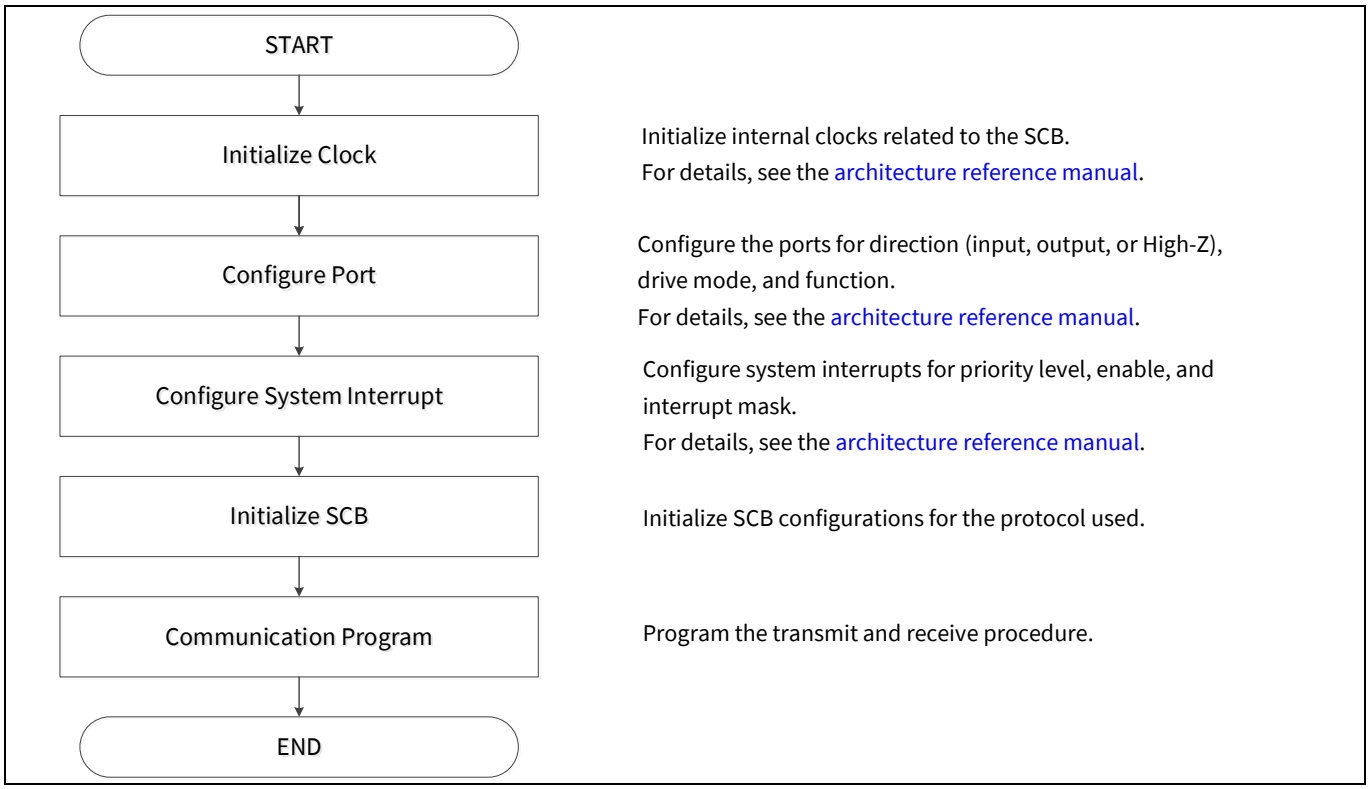


**Figure 2 SCB general settings flow**

The *Initialize Clock* block sets the peripheral clock (PCLK) input to the SCB. This setting is configured by the clock system in the MCU.

The *Configure Port* block configures the connection of external pins used for communication with the SCB. The *Configure System Interrupt* block configures the interrupt settings used for communicating as a transmission interrupt or a receive interrupt.

For details, see the architecture reference manual of each block. The *Initialize SCB* block initializes the registers of the SCB for the protocol used. The *Communication Program* block describes the control procedure when the SCB transmits or receives data.

# 4 SPI setting procedure example

This section shows an example of using SPI. The SCB supports SPI Master mode and SPI Slave mode with Motorola, Texas Instruments, and National Semiconductor protocols. See the architecture reference manual for details of each protocol.

## 4.1 Master mode

This example configures SCB in Motorola SPI Master mode and transmits one word (16-bits) of data and receives one word of data from the SPI Slave.

**Use case**

- SCB Mode = Motorola SPI Master mode
- SCB Channel = 1
- PCLK (Peripheral Clock) = 4 MHz
- Bit rate = 1 Mbps
- Tx/Rx data length = 16 bits
- Tx/Rx FIFO = Yes (16-bit FIFO data elements)
- Rx interrupt = Enable
- Used ports
  - SCLK: SCB1_CLK (P18.2)
  - MOSI: SCB1_MOSI (P18.1)
  - MISO: SCB1_MISO (P18.0)
  - SELECT: SCB1_SEL0 (P18.3)
- MOSI data is driven on a falling edge of SCLK.
- MISO data is captured on a falling edge of SCLK after half a SPI SCLK period from a rising edge.

Figure 3 shows the example of connection between the SCB and another SPI device.

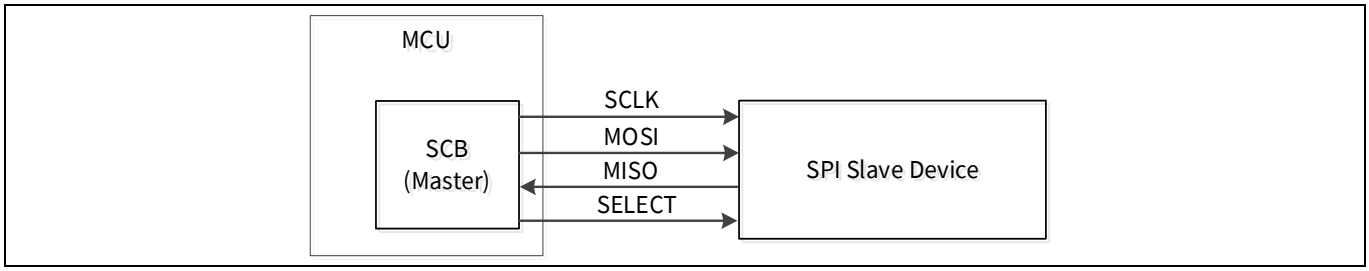


**Figure 3 Example of SPI (Master mode) communication**

In SPI mode, SCLK, MOSI, MISO, and SELECT signals are connected to another Slave device. In Master mode, SCLK and MOSI are the outputs, while MISO is the input. SELECT is used as an indication of valid data period for the Slave device. Up to four SELECT signals can be assigned.

### SPI setting procedure example

Figure 4 shows the setting procedure and operation example for Master mode.

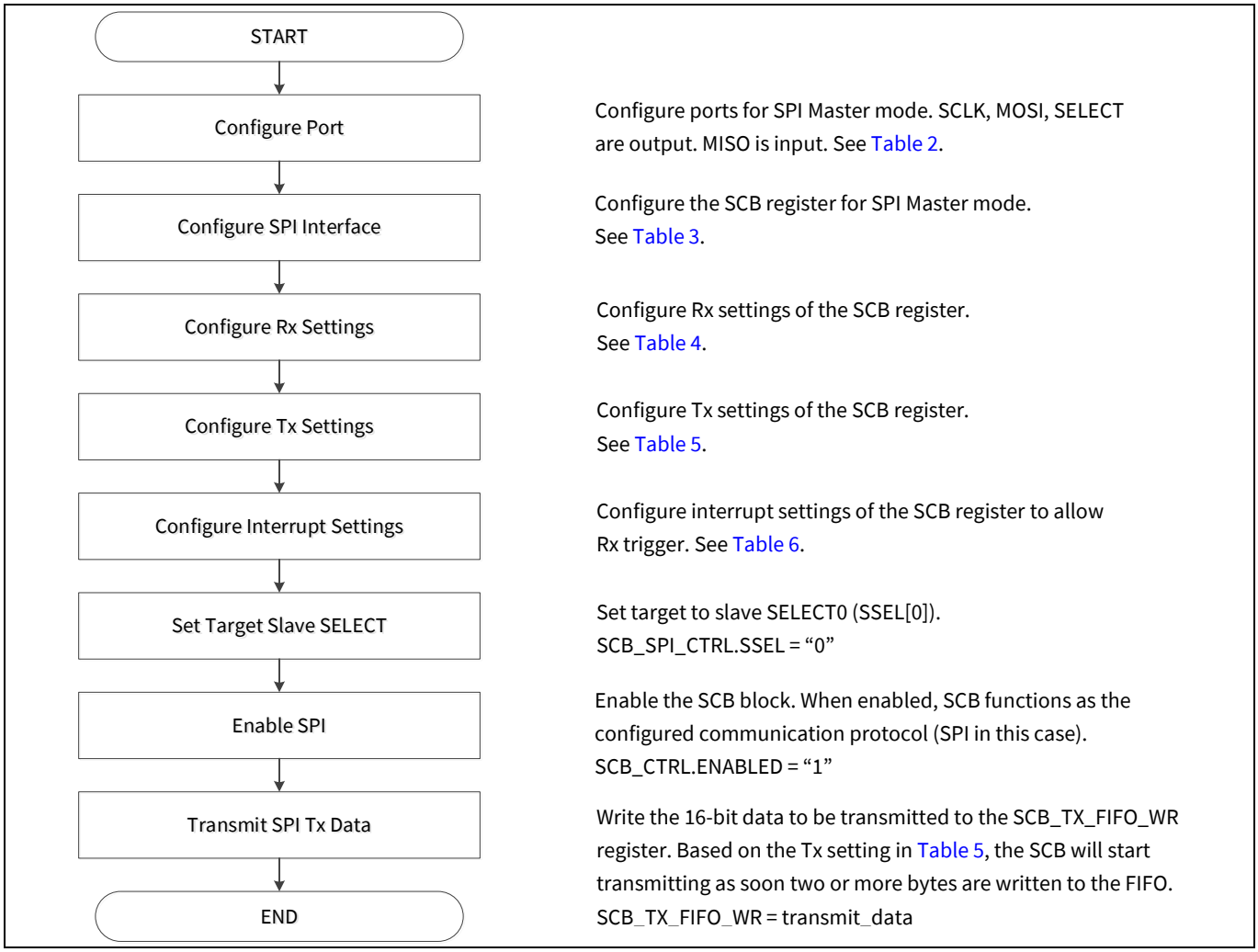


**Figure 4 SPI Master mode operation**

After the common settings such as ports and SPI interface configurations, enable the SCB block by setting the enable bit (SCB_CTRL.ENABLED). The software sets the transmission data to the Tx FIFO; then the SPI controller transmits the data to the Slave device. If the number of bytes received exceeds the Rx FIFO threshold level (SCB_RX_FIFO_CTRL.TRIGGER_LEVEL), the SPI controller notifies the receive interrupt to the CPU. The software can then read the received data from the Rx FIFO.

### SPI setting procedure example

Figure 5 shows an example of SPI receiver interrupt handling.

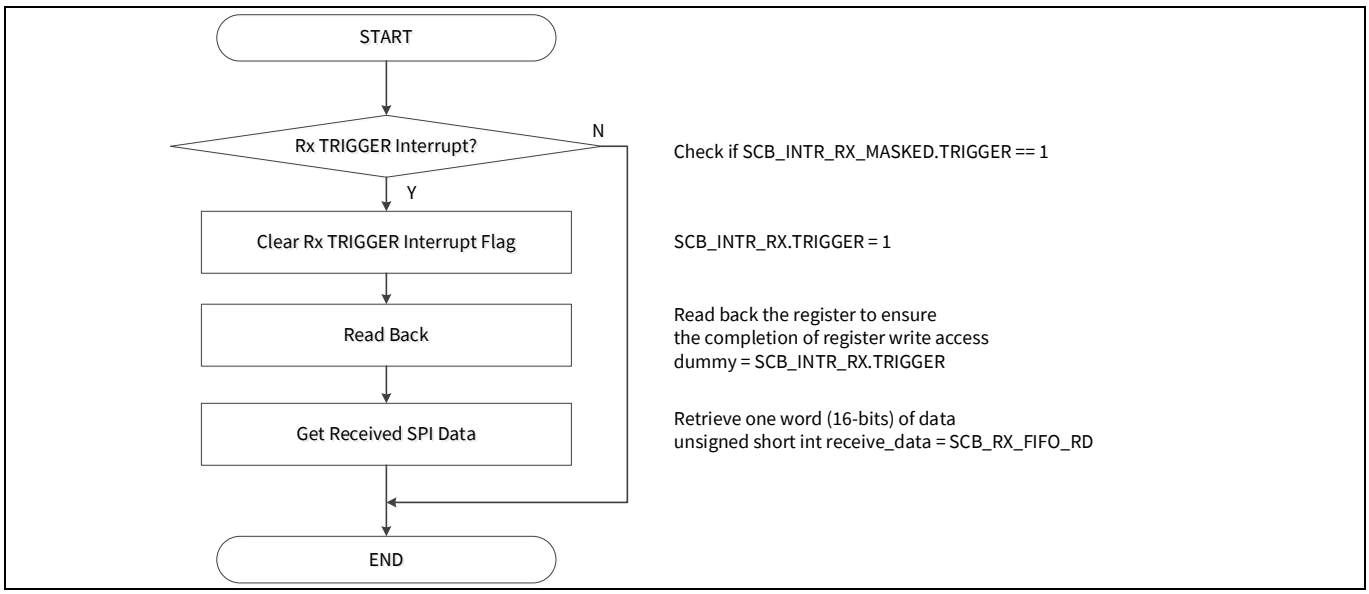


**Figure 5 SPI Master mode interrupt**

When the SPI controller stores the data in the Rx FIFO, the SPI controller can notify a receive interrupt to the CPU. When the CPU detects a receive interrupt, the software can get the received data from the Rx FIFO. If the cause of an interrupt is not TRIGGER, you should check for other interrupt causes or clear all interrupts.

*Note: Procedures for causes other than the TRIGGER interrupt should be handled by the application.*

## 4.1.1 Configure ports

This section explains an example of the port setting used in SPI Master mode. In this mode, SCLK, MOSI, MISO, and SELECT are used as interface signals. Each signal is assigned to the port number as follows:

**Table 1 Port settings**

| Signal | Port number |
|---|---|
| SCLK | SCB1_CLK (P18.2) |
| MOSI | SCB1_MOSI (P18.1) |
| MISO | SCB1_MISO (P18.0) |
| SELECT | SCB1_SEL0 (P18.3) |

Table 2 shows an example of the port configuration in SPI Master mode.

SCLK, MOSI, and SELECT are configured for the output port, and MISO is configured for the input port with the GPIO_PRT18_CFG.DRIVE_MODEx register and GPIO_PRT18_CFG.IN_ENx register. The pin functions are determined by the HSIOM_PRT18.PORT_SEL0.IOx_SEL register.

**SPI setting procedure example**

**Table 2 SPI (Master mode): Example of port configuration**

| Register | Port configuration | | | | Remark |
|---|---|---|---|---|---|
| | SCLK (*x*=2) | MOSI (*x*=1) | MISO (*x*=0) | SELECT (*x*=3) | |
| GPIO_PRT18_ CFG.DRIVE_MODEx | 6 | 6 | 0 | 6 | 0: High-Z.<br>6: Strong Drive Output. |
| GPIO_PRT18_ CFG.IN_ENx | 0 | 0 | 1 | 0 | 0: Input buffer disabled.<br>1: Input buffer enabled. |
| HSIOM_PRT18.PORT_ SEL0.IOx_SEL | 19 | 19 | 19 | 19 | 19: ACT#7. For the SCB function of the pin, see the datasheet. |

*Note: Bits that are not listed in Table 2 have default values. For default values, see the respective registers in the registers reference manual.*

### 4.1.2 Configure SPI interface registers

This section explains an example of setting SPI registers used in SPI Master mode. The following SPI registers are used: interface configuration register, Rx and Tx control register, and interrupt register.

Table 3 shows an example of the SPI interface configuration in SPI Master mode. These registers configure the SPI interface in communication mode or clock and data. In this case, this SPI interface behaves in Motorola mode, SPI Master mode, and 16-bit FIFO data elements.

**Table 3 SPI (Master mode): Example of SPI interface configuration**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_CTRL | MODE | 1 | SPI mode |
| | OVS | 3 | Four oversampling (OVS + 1).<br>This OVS bit is used for determining the bit rate as shown in section 4.1.3 Bit rate setting. |
| | MEM_WIDTH | 1 | 16-bit FIFO data elements |
| SCB_SPI_CTRL | MODE | 0 | SPI_MOTOROLA mode |
| | MASTER_MODE | 1 | Master mode |
| | SSEL_POLARITY0 | 0 | SELECT pin to the Slave is LOW/‘0’ active. |
| | CPHA | 0 | When the clock is inactive, the level is LOW. MOSI is driven on the falling edge of SCLK. MISO is captured on the rising edge of SCLK. |
| | CPOL | 0 | |
| | LATE_MISO_SAMPLE | 1 | The alternate clock edge is used (which comes half a SPI SCLK period later). |
| | SSEL_CONTINUOUS | 0 | SCLK is generated when the SPI Master is enabled and data is transmitted. |
| | SELECT_PRECEDE | 0 | Used only in SPI Texas Instruments mode. |
| | SCLK_CONTINUOUS | 0 | Disables the Master to generate a continuous SCLK regardless of whether there is data to send. |

*Note: Bits that are not listed in Table 3 have default values. For default values, see the respective registers in the registers reference manual.*

### SPI setting procedure example

Table 4 shows an example of the Rx configuration in SPI Master mode. These registers configure the receiver control settings. In this case, the received data format is most significant bit (MSb) first, and the data width is 16 bits.

**Table 4 SPI (Master mode): Example of Rx configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_RX_CTRL | MSB_FIRST | 1 | MSb first |
| | MEDIAN | 0 | Disables the digital 3-tap median filter to be applied to the input of Rx FIFO to filter glitches on the line. |
| | DATA_WIDTH | 15 | Width of Rx data = 16 |
| SCB_RX_FIFO_CTRL | TRIGGER_LEVEL | 1 | Interrupt occurs when there are more than two entries in the Rx FIFO. |

*Note: Bits that are not listed in Table 4 have default values. For default values, see the respective registers in the registers reference manual.*

Table 5 shows an example of the Tx configuration in SPI Master mode. These registers configure the transmitter control settings. In this case, the transmitter data format is MSb first, and the data width is 16 bits.

**Table 5 SPI (Master mode): Example of Tx configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_TX_CTRL | MSB_FIRST | 1 | MSb first |
| | DATA_WIDTH | 15 | Width of Tx data = 16 |

*Note: Bits that are not listed in Table 5 have default values. For default values, see the respective registers in the registers reference manual.*

Table 6 shows an example of the interrupt configuration in SPI Master mode. These registers configure the interrupt mask settings. In this case, when the Rx trigger occurs, the receiver interrupt occurs and the SCB notifies the CPU with the interrupt controller.

**Table 6 SPI (Master mode): Example of interrupt configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_INTR_TX_MASK | – | 0 | Tx interrupt is masked. |
| SCB_INTR_RX_MASK | TRIGGER | 1 | Allow Rx trigger for interrupt. |
| SCB_INTR_M_MASK | – | 0 | Master interrupt is masked. |
| SCB_INTR_S_MASK | – | 0 | Slave interrupt is masked. |
| SCB_INTR_SPI_EC_MASK | – | 0 | External clock interrupt is masked. |

*Note: Bits that are not listed in Table 6 have default values. For default values, see the respective registers in the registers reference manual.*

### 4.1.3 Bit rate setting

The bit rate setting is valid only in Master mode. The formula of bit rate calculation is as follows:

```
Bit rate [bps] = Input Clock [Hz] / OVS
       OVS : SCB_CTRL.OVS + 1
```

In this case, bit rate is calculated as follows:

```
       Bit rate = Input Clock [Hz] / OVS  = PCLK(4MHz) / (3+1)  = 1 [Mbps]
```

For more details, see the architecture reference manual.

### 4.1.4 Use case

This section explains an example of the SPI Master mode.

You can configure these parameters (SPI ports, Data rate, etc.) in the **Peripherals** tab of the **Device Configurator**; based on these, ModusToolbox™ software automatically generates the corresponding code.

1. Double-click the Device Configurator on the left Quick Panel as shown in Figure 6 or you can also double-click the *design.modus* file to open the Device Configurator in your project, *\bsps\TARGET_APP_KIT_XMC72_EVK\config\design.modus*.

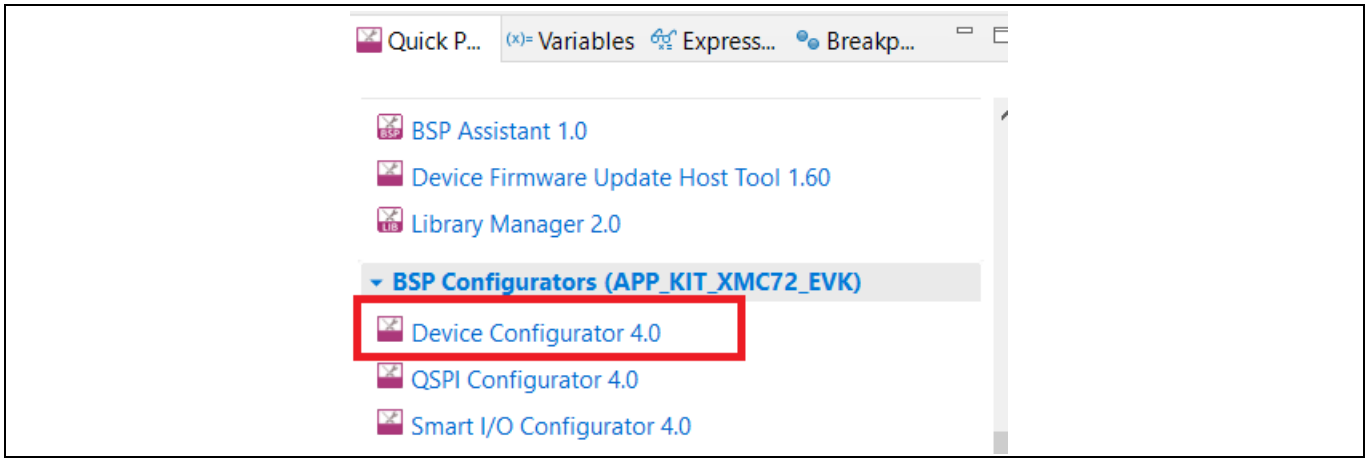


**Figure 6 Device Configurator in ModusToolbox™**

2. Configure SCB mode and channel, data rate and ports in the **Peripherals** tab of the Device Configurator as shown in Figure 7.
   - Select SCB1 channel to SPI mode
   - Mode: Master
   - Data Rate: 1024
   - Clock: 16 bit Divider 0 clk
   - SCLK: P18[2]
   - MOSI: P18[1]
   - MISO: P18[0]
   - SS0: P18[3]

## SPI setting procedure example

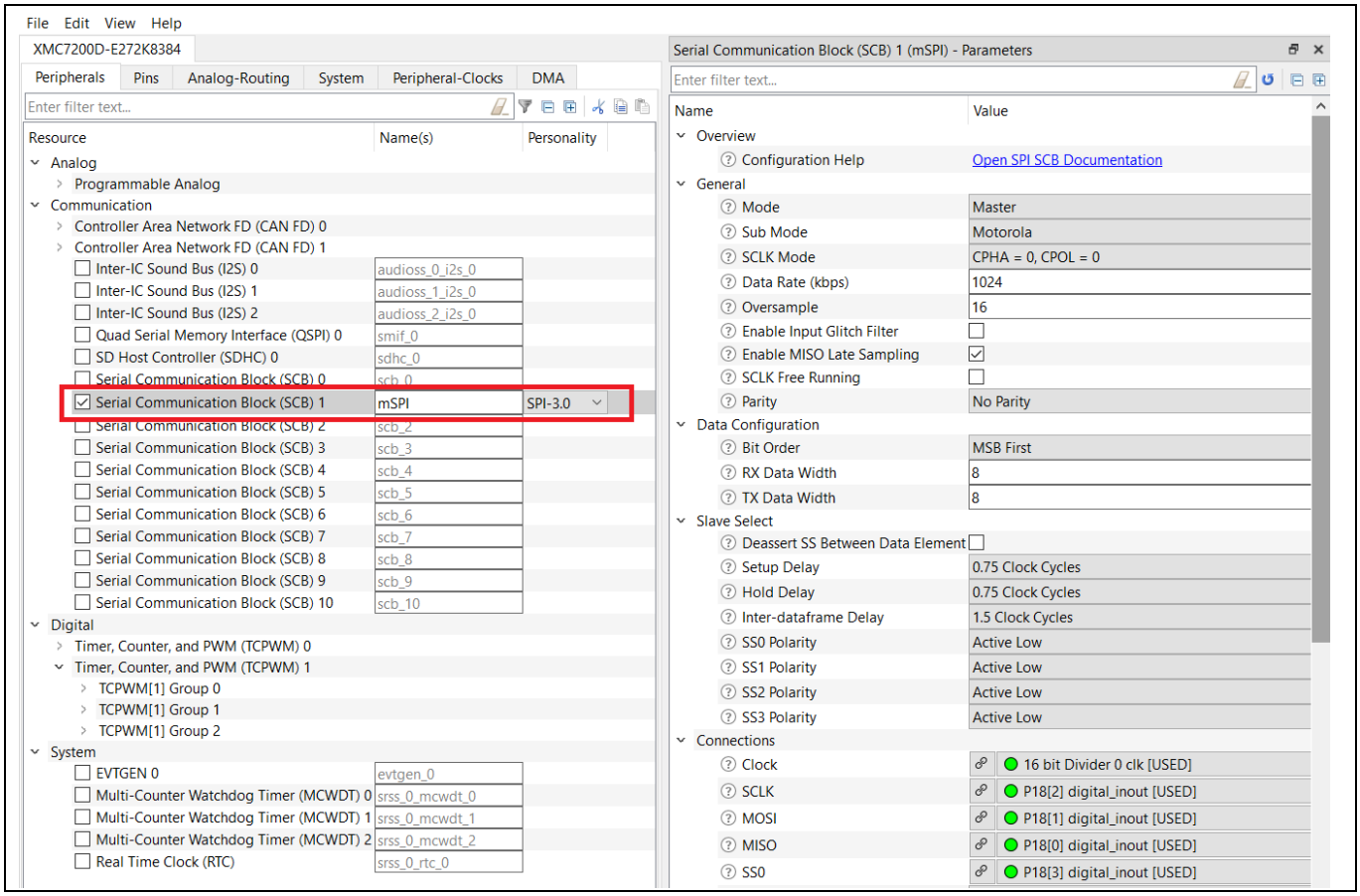


**Figure 7 SCB configuration in ModusToolbox™**

**Code Listing 1 Example of SPI Master configuration using Peripheral Driver Library (PDL)**

```
const cy_stc_sysint_t mSPI_SCB_IRQ_cfg =
{
        .intrSrc = ((NvicMux3_IRQn << 16) | mSPI_IRQ),
        .intrPriority = 2UL
};
void mSPI_Interrupt(void)
{
    Cy_SCB_SPI_Interrupt(mSPI_HW, &mSPI_context);
}
uint32 sendPacket(uint32 *txBuffer, uint32 transferSize)
{
    cy_en_scb_spi_status_t errorStatus;

    /* Initiate SPI Master write and read transaction. */
    errorStatus = Cy_SCB_SPI_Transfer(mSPI_HW, txBuffer, NULL, transferSize,
&mSPI_context);
    return (errorStatus);
}
```

## SPI setting procedure example

**Code Listing 1 Example of SPI Master configuration using Peripheral Driver Library (PDL)**

```
int main(void)
{
    cy_rslt_t result;
    cy_en_scb_spi_status_t initStatus;
    cy_en_sysint_status_t sysSpistatus;

uint32  txBuffer[NUMBER_OF_ELEMENTS];  /* Buffer to hold command packet to be sent
to the slave by the master */
       uint32 cmd = LED_OFF;
       uint32 status = 0;

    /* Initialize the device and board peripherals */
    result = cybsp_init();
    /* Board init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Initialize retarget-io for uart logs */
    result = cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,
                                 CY_RETARGET_IO_BAUDRATE);
    /* Retarget-io init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Initialize the user LED */
    result = cyhal_gpio_init(CYBSP_USER_LED, CYHAL_GPIO_DIR_OUTPUT,
                             CYHAL_GPIO_DRIVE_STRONG, CYBSP_LED_STATE_OFF);
    /* User LED init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* \x1b[2J\x1b[;H - ANSI ESC sequence for clear screen */
    printf("\x1b[2J\x1b[;H");

    printf("*************** "
```

## SPI setting procedure example

**Code Listing 1 Example of SPI Master configuration using Peripheral Driver Library (PDL)**

```
            "PDL: SPI Master "
            "*************** \r\n\n");

    printf("Configuring SPI master...\r\n");

    /* Init SPI master */
    /* Configure SPI block */
    initStatus = Cy_SCB_SPI_Init(mSPI_HW, &mSPI_config, &mSPI_context);
       /* If the initialization fails, return failure status */
       if(initStatus != CY_SCB_SPI_SUCCESS)
       {
              handle_error();
       }

       /* Set active slave select to line 0 */
       Cy_SCB_SPI_SetActiveSlaveSelect(mSPI_HW, CY_SCB_SPI_SLAVE_SELECT0);

       /* Enable SPI master block. */
       Cy_SCB_SPI_Enable(mSPI_HW);

       /* Hook interrupt service routine */
       sysSpistatus = Cy_SysInt_Init(&mSPI_SCB_IRQ_cfg, &mSPI_Interrupt);
       if(sysSpistatus != CY_SYSINT_SUCCESS)
       {
              return(INIT_FAILURE);
       }
       /* Enable interrupt in NVIC */
       NVIC_EnableIRQ((IRQn_Type) mSPI_SCB_IRQ_cfg.intrSrc);

       /* Initialization completed */

    /* Enable interrupts */
    __enable_irq();

    for (;;)
    {
       /* Toggle the current LED state */
              cmd = (cmd == LED_ON) ? LED_OFF : LED_ON;
              /* Form the command packet */
              txBuffer[PACKET_SOP_POS] = PACKET_SOP;
```

## SPI setting procedure example

**Code Listing 1 Example of SPI Master configuration using Peripheral Driver Library (PDL)**

```
                txBuffer[PACKET_CMD_POS] = cmd;
                txBuffer[PACKET_EOP_POS] = PACKET_EOP;

                /* Pass the command packet to the master along with the number of
bytes to be
                 * sent to the slave.*/
                status = sendPacket(txBuffer, SIZE_OF_PACKET);
                printf("SPI Master Send Packet complete \n");

            /* If no error wait till master sends data in the Tx FIFO */
            if(status == CY_SCB_SPI_SUCCESS)
            {
                status = checkTranferStatus(SIZE_OF_PACKET);

            }

                /* Check whether the slave succeeded in receiving the required
number of bytes
                 * and in the right format */
                if(status == TRANSFER_COMPLETE)
                {
                        /* Communication succeeded. Update the LED. */
                        //printf("SPI Salve Read Complete \n");
                        UpdateLED(txBuffer[PACKET_CMD_POS]);
                }
                else
                {
                        /* Communication failed */
                        printf("SPI Slave Read failed \n");
                        handle_error();
                }

                /* Give delay before initiating the next command */
                Cy_SysLib_Delay(CMD_DELAY);

    }
}
```

## 4.2 Slave mode

This example sets the Motorola SPI Slave mode so that the Master transmits two half-words of data to the Slave, and then the Slave receives two half-words of data from the Master.

**Use case**

- SCB Mode = Motorola SPI Slave mode
- SCB Channel = 1
- PCLK = 4 MHz
- Bit rate = 1 Mbps
- Tx/Rx data length = 16 bits
- Tx/Rx FIFO = Yes (16-bit FIFO data elements)
- Tx/Rx interrupts = Enable
- Used ports
  - SCLK: SCB1_CLK (P18.2)
  - MOSI: SCB1_MOSI (P18.1)
  - MISO: SCB1_MISO (P18.0)
  - SELECT: SCB1_SEL0 (P18.3)
- MISO data is driven on a falling edge of SCLK.
- MOSI data is captured on a rising edge of SCLK.

Figure 8 shows the example of a connection between the SCB and another SPI device.

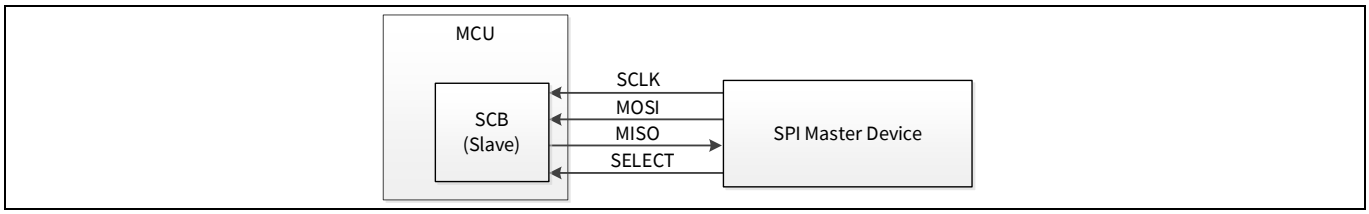


**Figure 8 Example of SPI (Slave mode) communication connection**

In SPI mode, SCLK, MOSI, MISO, and SELECT signals connect to another SPI Master device. In Slave mode, SCLK, MOSI, and SELECT are input ports, and MISO is the output port. SELECT indicates when valid data is transmitted from the SPI Master device or SPI Slave device.

Figure 9 shows the setting procedure and operation example for Slave mode.

### SPI setting procedure example

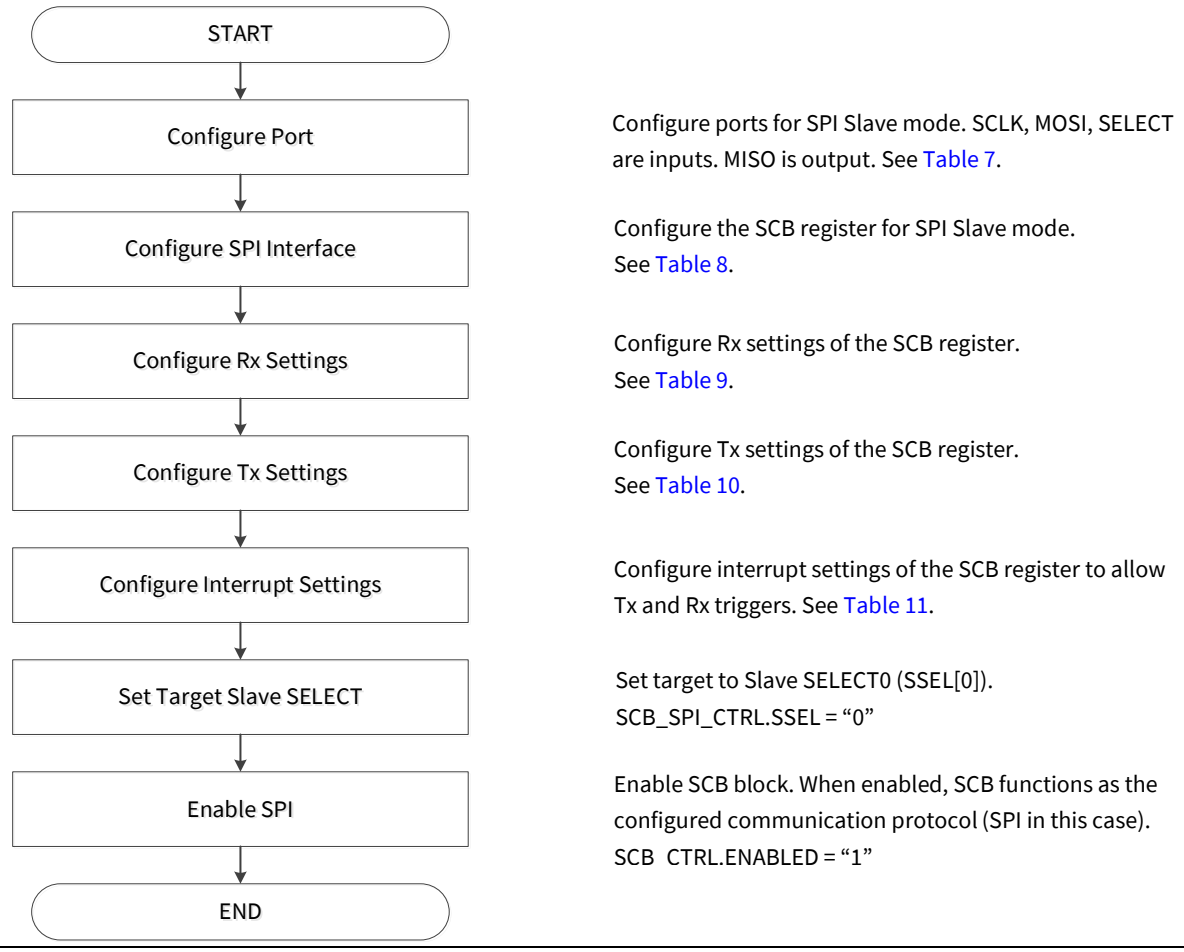


**Figure 9 SPI Slave mode operation**

### SPI setting procedure example

The SCB is configured in SPI Slave mode by these procedures. Received and transmitted interrupts are configured with each threshold level of the FIFO. After the SPI controller is enabled (SCB_CTRL.ENABLE = “1”), the SPI controller waits for the interrupt trigger. If the SPI controller receives an interrupt, the interrupt handler is called as shown in Figure 10, which shows an example of the SPI controller receiver interrupt handling.

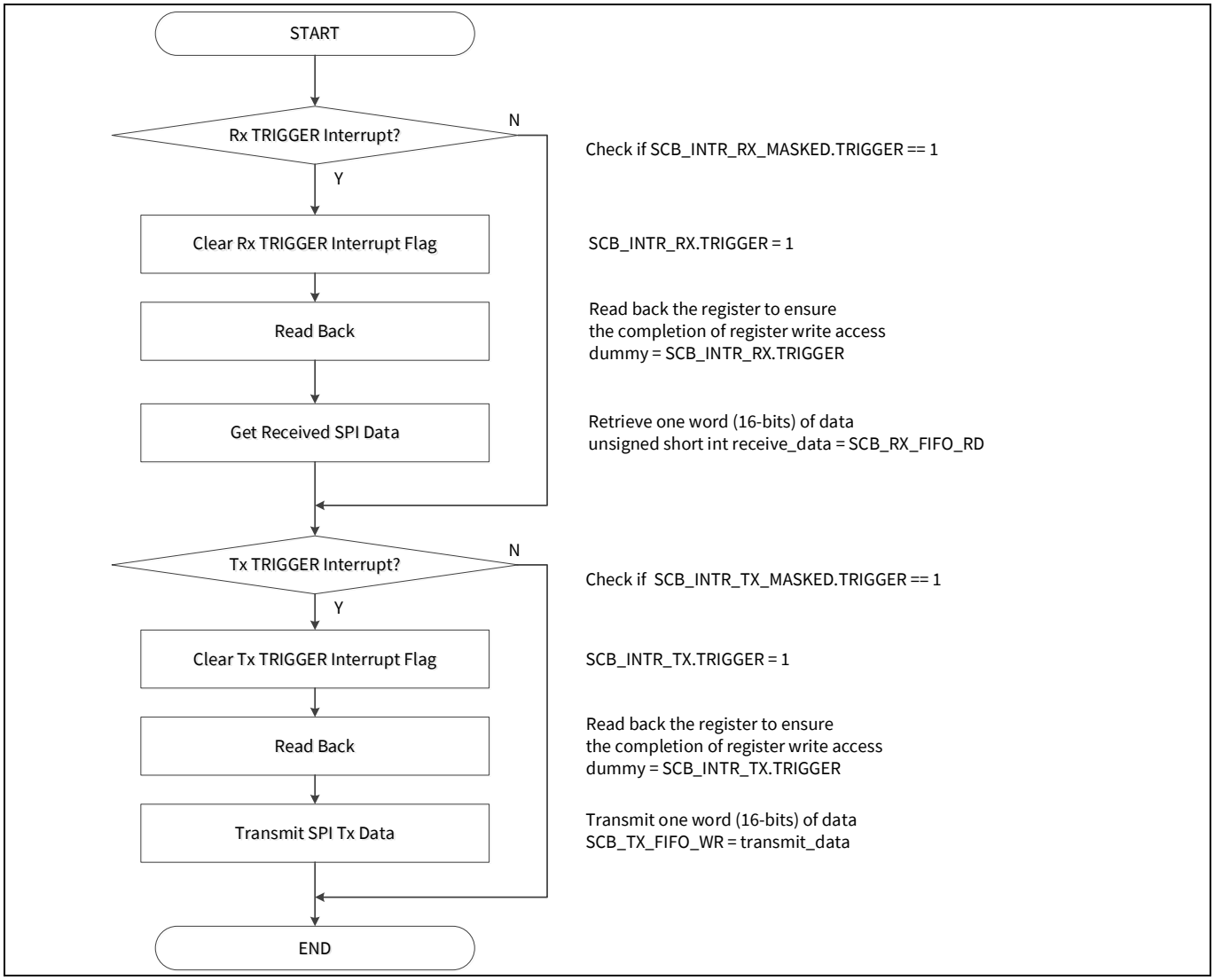


**Figure 10 SPI Slave mode interrupt**

In this case, when more than two bytes of data is stored in the Rx FIFO, the SPI controller can notify a receive interrupt to the CPU. After the CPU detects a receive interrupt, if SCB_INTR_RX.TRIGGER is “1”, the software can get two bytes of the received data from the Rx FIFO. On the other hand, when there is no entry data in Tx FIFO, the SPI controller can also notify a transmission interrupt to the CPU. If SCB_INTR_TX.TRIGGER is “1”, the software can transmit the data via the Tx FIFO.

*Note: Procedures for causes other than the TRIGGER interrupt should be handled by the application.*

### 4.2.1 Configure ports

This section explains an example of the port setting used in SPI Slave mode. In this mode, SCLK, MOSI, MISO, and SELECT are used as interface signals. Each signal is assigned to the port number as follows:

- SCLK : SCB1_CLK (P18.2)
- MOSI : SCB1_MOSI (P18.1)
- MISO : SCB1_MISO (P18.0)
- SELECT : SCB1_SEL0 (P18.3)

Table 7 shows an example of the port configuration in SPI Slave mode.

SCLK, MOSI, and SELECT are configured for input ports, and MISO is configured for the output port with the GPIO_PRT18_CFG.DRIVE_MODEx register and the GPIO_PRT18_CFG.IN_ENx register. The pin functions are determined by the HSIOM_PRT18.PORT_SEL0.IOx_SEL register.

**Table 7 SPI (Slave mode): Example of port configurations**

| Register | Port configuration | | | | Remark |
|---|---|---|---|---|---|
| | SCLK (*x*=2) | MOSI (*x*=1) | MISO (*x*=0) | SELECT (*x*=3) | |
| GPIO_PRT18_ CFG.DRIVE_MODEx | 0 | 0 | 6 | 0 | 0: High-Z<br>6: Strong Drive Output |
| GPIO_PRT18_ CFG.IN_ENx | 1 | 1 | 0 | 1 | 0: Input buffer disabled<br>1: Input buffer enabled |
| HSIOM_PRT18.PORT_ SEL0.IOx_SEL | 19 | 19 | 19 | 19 | 19: ACT#7. For the SCB function of the pin, see the datasheet. |

*Note: Bits that are not listed in the table above have their default values. For default values, refer to the respective registers in the registers reference manual.*

### 4.2.2 Configure SPI interface registers

This section explains an example of the SPI registers setting in SPI Slave mode. The following SPI registers are used: Interface configuration register, Rx and Tx control register, and Interrupt register.

Table 8 shows an example of the SPI interface configuration in SPI Slave mode. These registers configure the SPI interface with a communication mode or a format of clock and data. In this case, the SPI interface behaves in Motorola mode, SPI Slave mode, and 16-bit FIFO data elements.

**Table 8 SPI (Slave mode): Example of SPI interface configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_CTRL | MODE | 1 | SPI mode |
| | OVS | 15 | No effect in Slave mode. |
| | MEM_WIDTH | 1 | 16-bit FIFO data elements |
| SCB_SPI_CTRL | MODE | 0 | SPI_MOTOROLA mode |
| | MASTER_MODE | 0 | Slave mode |
| | SSEL_POLARITY0 | 0 | SELECT pin to the Slave is active LOW. |
| | CPHA | 0 | |

**SPI setting procedure example**

| Register | Bit | Value | Remark |
|---|---|---|---|
| | CPOL | 0 | When the clock is inactive, the level is LOW. MOSI is driven on the falling edge of SCLK. MISO is captured on the rising edge of SCLK. |

*Note: Bits that are not listed in Table 8 have default values. For default values, see the respective registers in the registers reference manual.*

Table 9 shows an example of the Rx configuration in SPI Slave mode. These registers configure the receiver control settings. In this case, the received data format is MSb first, and the data width is 16 bits.

**Table 9 SPI (Slave mode): Example of Rx configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_RX_CTRL | MSB_FIRST | 1 | MSb first |
| | MEDIAN | 0 | Disables the digital 3-tap median filter to be applied to the input of the Rx FIFO to filter glitches on the line. |
| | DATA_WIDTH | 15 | Width of Rx data = 16 |
| SCB_RX_FIFO_CTRL | TRIGGER_LEVEL | 1 | Interrupt occurs when there are more than two entries in the Rx FIFO. |

*Note: Bits that are not listed in Table 9 their default values. For default values, see the respective registers in the registers reference manual.*

Table 10 shows an example of the Tx configuration in SPI Slave mode. These registers configure the transmitter control settings. In this case, the transmitter data format is MSb first, and the data width is 16 bits.

**Table 10 SPI (Slave mode): Example of Tx configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_TX_CTRL | MSB_FIRST | 1 | MSb first |
| | DATA_WIDTH | 15 | Width of Tx data = 16 |
| SCB_TX_FIFO_CTRL | TRIGGER_LEVEL | 1 | Interrupt occurs when Tx FIFO is empty. |

*Note: Bits that are not listed in Table 10 have their default values. For default values, refer the respective registers in the registers reference manual.*

Table 11 shows an example of the interrupt configuration in SPI Slave mode. These registers configure the interrupt mask settings. In this case, when an Rx trigger occurs, the interrupt occurs and the SCB notifies the CPU with the interrupt controller.

**Table 11 SPI (Slave mode): Example of interrupt configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_INTR_TX_MASK | TRIGGER | 1 | Allow interrupt to be triggered based on the TRIGGER_LEVEL setting of SCB_TX_FIFO_CTRL. |

**SPI setting procedure example**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_INTR_RX_MASK | TRIGGER | 1 | Allow interrupt to be triggered based on the TRIGGER_LEVEL setting of SCB_RX_FIFO_CTRL. |
| SCB_INTR_M_MASK | – | 0 | Master interrupt mask is masked. |
| SCB_INTR_S_MASK | – | 0 | Slave interrupt mask is masked. |
| SCB_INTR_SPI_EC_MASK | – | 0 | Externally clocked SPI interrupt is masked. |

*Note: Bits that are not listed in Table 11 have default values. For default values, see the respective registers in the registers reference manual.*

### 4.2.3 Use case

This section explains an example of the SPI Slave mode.

You can configure these parameters (SPI ports, data rate, etc.) in the **Peripherals** tab of the **Device Configurator**; based on these, ModusToolbox™ software automatically generates the corresponding code.

1. Double-click the Device Configurator on the left Quick Panel as shown in Figure 11 or you can also double-click the *design.modus* file to open the Device Configurator in your project, *\bsps\TARGET_APP_KIT_XMC72_EVK\config\design.modus*.

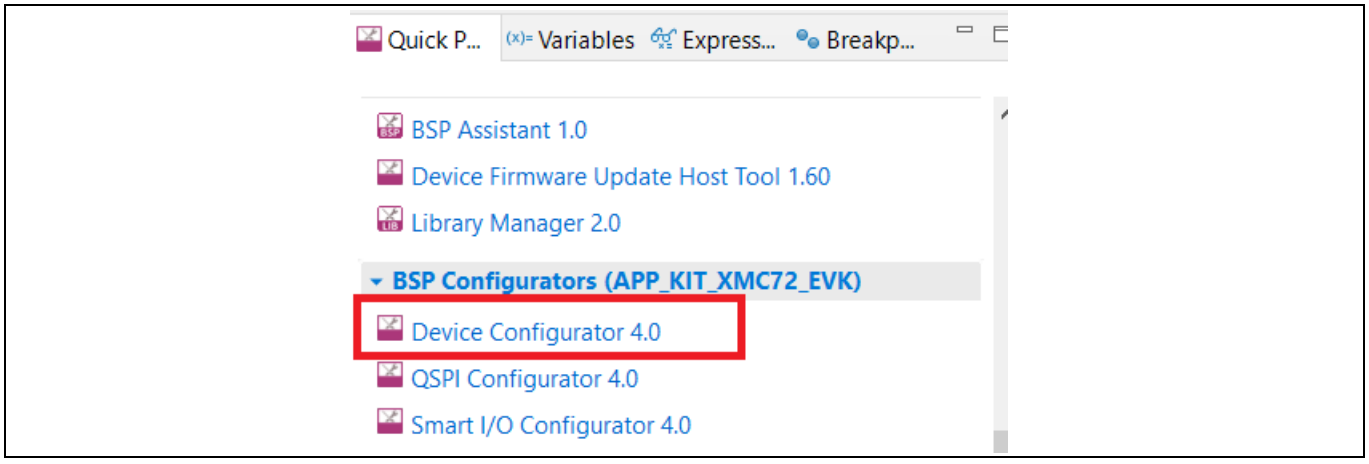


**Figure 11 Device Configurator in ModusToolbox™**

2. Configure SCB mode and channel, data rate and ports in the **Peripherals** tab of the Device Configurator as shown in Figure 12.
   - Select SCB1 channel to SPI mode
   - Mode: Slave
   - Data Rate: 1024
   - Clock: 16 bit Divider 0 clk
   - SCLK: P18[2]
   - MOSI: P18[1]
   - MISO: P18[0]
   - SS0: P18[3]

## SPI setting procedure example

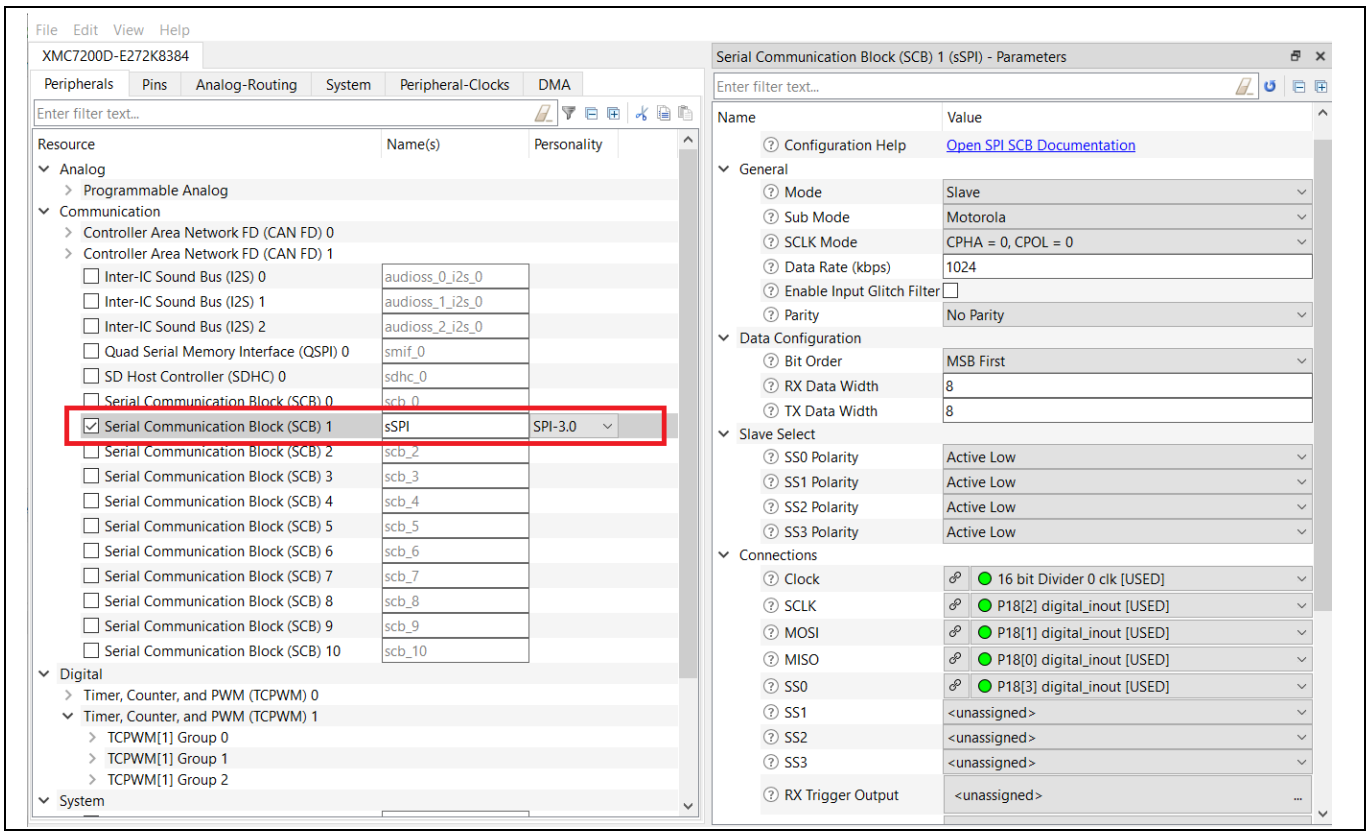


**Figure 12 SCB configuration in ModusToolbox™**

**Code Listing 2 Example of SPI Slave configuration using PDL**

```
uint32 readPacket(uint32 *rxBuffer, uint32 transferSize)
{
      uint32 slaveStatus;

      /* Wait till all the bytes are received */
    while(Cy_SCB_SPI_GetNumInRxFifo(sSPI_HW) < transferSize)
    {

    }

      /* Read RX FIFO */
      Cy_SCB_SPI_ReadArray(sSPI_HW, rxBuffer, transferSize);

      /* Check start and end of packet markers */
      if ((rxBuffer[PACKET_SOP_POS] == PACKET_SOP) && (rxBuffer[PACKET_EOP_POS] ==
PACKET_EOP))
      {
            /* Data received correctly */
            slaveStatus = TRANSFER_COMPLETE;
      }
      else
      {
            /* Data was not received correctly */
            slaveStatus = TRANSFER_FAILURE;
      }

    return slaveStatus;
```

## SPI setting procedure example

**Code Listing 2 Example of SPI Slave configuration using PDL**

```
}
int main(void)
{
    cy_rslt_t result;
    cy_stc_scb_spi_context_t sSPI_context;
    cy_en_scb_spi_status_t initStatus;
      uint32  rxBuffer[NUMBER_OF_ELEMENTS]; /* Buffer to save the received data by
the slave */
      uint32 status = 0;

    /* Initialize the device and board peripherals */
    result = cybsp_init();
    /* Board init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Initialize retarget-io for uart logs */
    result = cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,
                                 CY_RETARGET_IO_BAUDRATE);
    /* Retarget-io init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Initialize the user LED */
    result = cyhal_gpio_init(CYBSP_USER_LED, CYHAL_GPIO_DIR_OUTPUT,
                             CYHAL_GPIO_DRIVE_STRONG, CYBSP_LED_STATE_OFF);
    /* User LED init failed. Stop program execution */
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* \x1b[2J\x1b[;H - ANSI ESC sequence for clear screen */
    printf("\x1b[2J\x1b[;H");

    printf("************** "
           "PDL: SPI Slave "
           "************** \r\n\n");

    printf("Configuring SPI slave...\r\n");
    /* Init SPI slave */

      /* Configure the SPI block */
      initStatus = Cy_SCB_SPI_Init(sSPI_HW, &sSPI_config, &sSPI_context);

      /* If the initialization fails, return failure status */
      if(initStatus != CY_SCB_SPI_SUCCESS)
      {
            return(INIT_FAILURE);
      }

      /* Set active slave select to line 0 */
      Cy_SCB_SPI_SetActiveSlaveSelect(sSPI_HW, CY_SCB_SPI_SLAVE_SELECT0);
```

## SPI setting procedure example

**Code Listing 2 Example of SPI Slave configuration using PDL**

```
      /* Enable the SPI Slave block */
      Cy_SCB_SPI_Enable(sSPI_HW);

      /* Initialization completed */

    /* Enable interrupts */
    __enable_irq();

    for (;;)
    {

            /* The below code is for slave function. It is implemented in this same
code
             * example so that the master function can be tested without the need
of one
             * more kit. */
            printf("SPI Master Transfer Status Complete \n");
            /* Get the bytes received by the slave */
            status = readPacket(rxBuffer, SIZE_OF_PACKET);
            /* Check whether the slave succeeded in receiving the required number
of bytes
                        * and in the right format */

            if(status == TRANSFER_COMPLETE)
            {
                  /* Communication succeeded. Update the LED. */
                  printf("SPI Salve Read Complete \n");
                  UpdateLED(rxBuffer[PACKET_CMD_POS]);

            }
            else
            {
                  /* Communication failed */
                  printf("SPI Slave Read failed \n");
                  handle_error();

            }

            /* Give delay before initiating the next command */
            Cy_SysLib_Delay(CMD_DELAY);

    }
}
```

# 5 UART setting procedure example

SCB features standard UART and multi-processor mode, Smart Card (ISO 7816) reader, IrDA. See the architecture reference manual for details of each protocol. In this section, the procedure to set Standard UART is explained as an example.

## 5.1 UART mode

This sample shows the usage of the SCB in standard UART mode. In this use case, after the respective registers are configured, the SCB transmits one byte of data to another device, and waits for an Rx data from another device.

**Use case**

- SCB Mode = Standard UART
- SCB Channel = 3
- PCLK = 40 MHz
- Baud Rate = 115,200 bps
- Data width = 8 bits
- Parity = None
- Stop Bits = 1
- Flow Control = None
- Tx/Rx FIFO = Yes
- Rx interrupt = Enable
- Used ports
  - Tx: SCB3_TX (P13.1)
  - Rx: SCB3_RX (P13.0)

An example Tx-Rx connection between the SCB and the external UART device is shown in Figure 13. In this example, flow control signals RTS and CTS are not used.

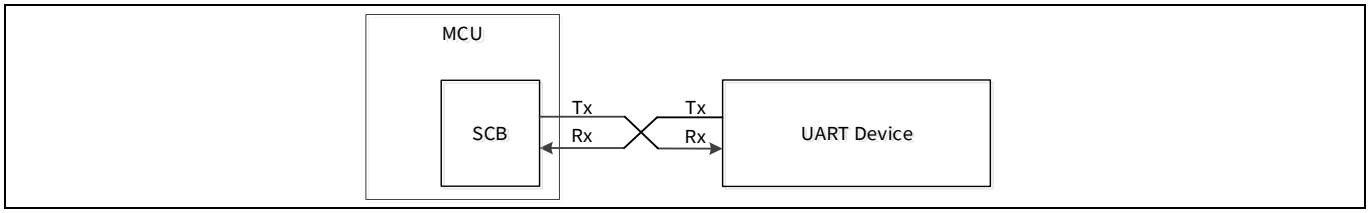


**Figure 13 Example of UART communication connection**

### UART setting procedure example

Figure 14 shows the setting procedure and operation example for UART.

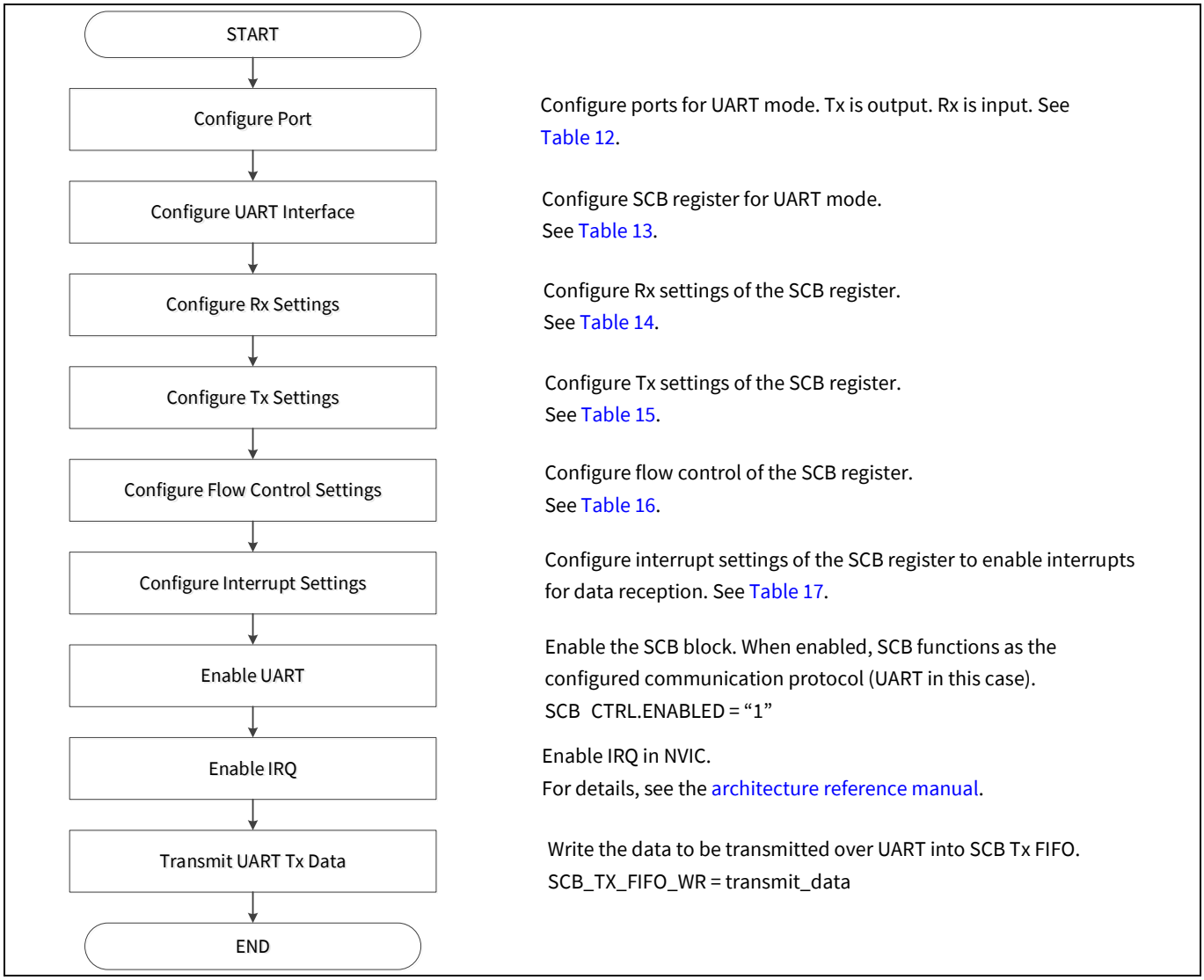


**Figure 14 UART operation**

After the general configuration (clock, port, interrupt, and so on), in this case, SCB registers are configured for UART. The software then transmits the data over UART by writing to the Tx FIFO. If the number of received data exceeds the threshold of Rx FIFO (SCB_RX_FIFO_CTRL.TRIGGER_LEVEL), the receive interrupt occurs and the interrupt handler would be called.

### UART setting procedure example

Figure 15 shows an example of UART interrupt handling.

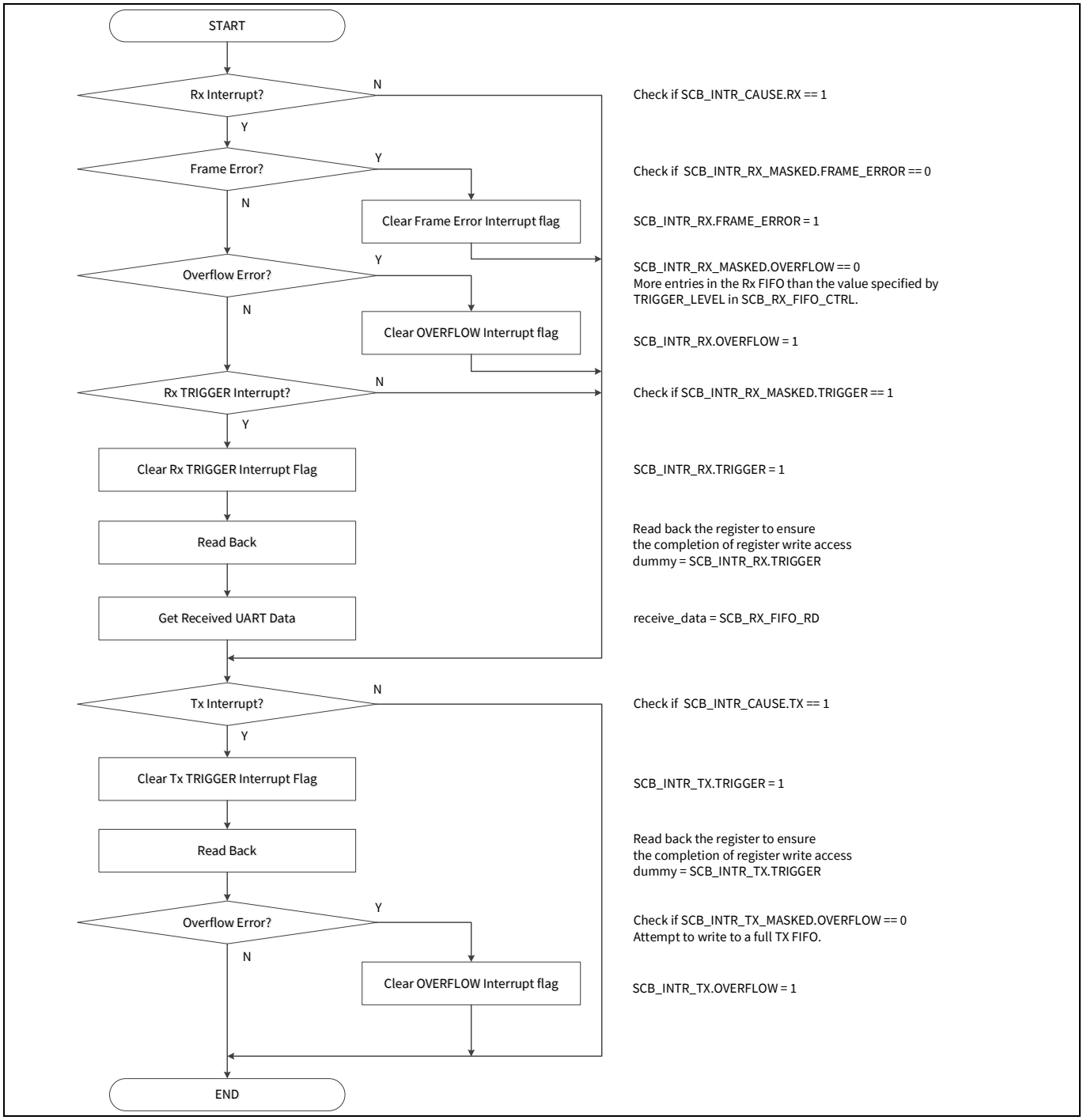


**Figure 15 UART: Example of interrupt flowchart**

Figure 15 shows the interrupt procedure that will be executed when a Tx/Rx interrupt occurs. If an Rx interrupt occurs, received errors are checked, the interrupt flag is cleared, and the received data is read. If a Tx interrupt occurs, the interrupt flag is cleared, and the overflow error is checked.

*Note: The error procedures should be handled by the application.*

### 5.1.1 Configure ports

This section provides an example of the port setting used in UART mode. In this mode, Tx and Rx are used as interface signals. Each signal is assigned to the port number as follows:

- Tx: SCB3_TX (P13.1)
- Rx: SCB3_RX (P13.0)

Table 12 shows an example of port configuration in UART mode.

Tx is configured for the output port, and Rx is configured for the input port with the GPIO_PRT13_CFG.DRIVE_MODEx register and GPIO_PRT13_CFG.IN_ENx register. Pin functions are determined by the HSIOM_PRT13.PORT_SEL0.IOx_SEL register.

**Table 12 UART: Example of port configurations**

| Register | Port configuration | | Remark |
|---|---|---|---|
| | Tx (*x*=1) | Rx (*x*=0) | |
| GPIO_PRT13_CFG.DRIVE_MODEx | 6 | 0 | 0: High-Z<br>6: Strong Drive Output |
| GPIO_PRT13_CFG.IN_ENx | 0 | 1 | 0: Input buffer disabled<br>1: Input buffer enabled |
| HSIOM_PRT13.PORT_SEL0.IOx_SEL | 17 | 17 | 17: ACT#5, for SCB function of the pin, see the datasheet. |

*Note: Bits that are not listed in Table 12 have default values. For default values, see the respective registers in the registers reference manual.*

### 5.1.2 Configure UART interface registers

This section explains an example of the UART registers setting in standard UART mode. The following UART registers are used: interface configuration register, Rx and Tx control register, and interrupt register.

Table 13 shows an example of the UART interface configuration. These registers configure the UART control settings in communication mode or as an oversampling value. In this case, the UART interface behaves in standard UART mode, 16 oversampling, and 8-bit FIFO data elements.

**Table 13 UART: Example of UART interface configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_CTRL | MODE | 2 | UART mode |
| | OVS | 15 | 16 oversampling (OVS + 1) |
| | MEM_WIDTH | 0 | 8-bit FIFO data elements |
| SCB_UART_CTRL | MODE | 0 | Standard UART mode |

*Note: Bits that are not listed in Table 13 have default values. For default values, see the respective registers in the registers reference manual.*

### UART setting procedure example

Table 14 shows an example of the Rx configuration in UART mode. These registers configure the receiver control settings. In this case, the received data format is LSb first, the data width is 8 bits, stop bit is 1, and data parity is none.

The received interrupt threshold (SCB_RX_FIFO_CTRL.TRIGGER_LEVEL) is set to “0”, which means that when one byte is received, an Rx interrupt should occur.

**Table 14 UART: Example of Rx configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_UART_RX_CTRL | POLARITY | 0 | Non-inversion |
| | MP_MODE | 0 | Multi-processor mode disabled |
| | DROP_ON_PARITY_ ERROR | 0 | Even if parity check fails, the received data is sent to the Rx FIFO. |
| | DROP_ON_FRAME_ERROR | 0 | Even if an error is detected in a start or stop period, the received data is sent to the Rx FIFO. |
| | STOP_BITS | 1 | Stop bits is STOP_BITS + 1 (= 2 of half bits). |
| | PARITY | 0 | Parity not used |
| | PARITY_ENABLED | 0 | Parity disabled |
| SCB_RX_CTRL | MSB_FIRST | 0 | Least significant bit (LSb) first |
| | MEDIAN | 0 | Median filter disabled |
| | DATA_WIDTH | 7 | DATA_WIDTH + 1 (=8) is the expected number of bits in the received data frame. |
| SCB_RX_MATCH | ADDR | 0 | Slave device address = 0 |
| | MASK | 0 | Slave device address mask = 0 |
| SCB_RX_FIFO_CTRL | TRIGGER_LEVEL | 0 | When one byte received, the Rx interrupt should occur. |

*Note: Bits that are not listed in Table 14 have default values. For default values, see the respective registers in the registers reference manual.*

Table 15 shows an example of the Tx configuration in UART mode. These registers configure the transmitter control settings. In this case, the transmitter data format is LSb first, the data width is 8 bits, stop bit is 1, and data parity is none. The transmitted interrupt threshold (SCB_RTX_FIFO_CTRL.TRIGGER_LEVEL) is set to “0”, which means that when one byte transmitted, a Tx interrupt should occur.

**Table 15 UART: Example of Tx configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_UART_TX_CTRL | RETRY_ON_NACK | 0 | Not used |
| | STOP_BITS | 1 | Stop bits is STOP_BITS + 1 (= 2 of halve bit). |
| | PARITY | 0 | Parity not used |
| | PARITY_ENABLED | 0 | Parity disabled |

### UART setting procedure example

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_TX_CTRL | MSB_FIRST | 0 | Least significant bit (LSb) first |
| | DATA_WIDTH | 7 | DATA_WIDTH + 1 (=8) is the number of bits in a transmitted data frame. |
| | OPEN_DRAIN | 0 | Normal operation mode |
| SCB_TX_FIFO_CTRL | TRIGGER_LEVEL | 0 | When one byte transmitted, Tx interrupt should occur. |

*Note: Bits that are not listed in Table 15 have default values. For default values, see the respective registers in the registers reference manual.*

Table 16 shows an example of the flow control configuration in UART mode. These registers configure the data flow control settings. In this use case, CTS and RTS are not used.

**Table 16 UART: Example of flow control configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_UART_FLOW_CTRL | CTS_ENABLED | 0 | CTS disabled |
| | CTS_POLARITY | 0 | Not used |
| | RTS_POLARITY | 0 | Not used |
| | TRIGGER_LEVEL | 0 | Not used |

*Note: Bits that are not listed in Table 16 have default values. For default values, see the respective registers in the registers reference manual.*

Table 17 shows an example of the interrupt configuration in UART mode. These registers configure the interrupt mask settings. Rx interrupt trigger, Tx interrupt trigger, Tx transmission done, and various alarms can be configured to be enabled. In this case, when the Tx TRIGGER and UART_DONE interrupts occur, the SCB notifies to the CPU with the interrupt controller.

**Table 17 UART: Example of interrupt configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_INTR_RX_MASK | – | 0xB61 | The following interrupt factors are enabled:<br>TRIGGER, OVERFLOW, UNDERFLOW, FRAME_ERROR, PARITY_ERROR, BREAK_DETECT. |
| SCB_INTR_TX_MASK | – | 0x221 | The following interrupt factors are enabled:<br>TRIGGER, OVERFLOW, UART_DONE. |

*Note: Bits that are not listed in Table 17 have their default values. For default values, refer the respective registers in the registers reference manual.*

### 5.1.3 Baud rate setting

The baud rate calculation formula is as follows:

```
Baud rate [bps] = Input Clock [Hz] / OVS
     OVS : SCB_CTRL.OVS + 1
```

For example, the following shows how to calculate a real UART baud rate from an ideal UART baud rate:

**Calculation example**

```
CLK_PERI frequency = 40 [MHz]
target UART baud rate(Bit rate) = 115,200 [bps]
OVS = 16 [oversamples]
```

You can use the specified for CLK_PERI frequency, target UART baud rate, and OVS for calculating the real baud rate.

First, calculate the ideal input clock to SCB:

```
Ideal Input Clock = target baud rate * OVS = 115,200 * 16 = 1,843,200 [Hz]
```

Next, calculate the ideal value of the clock divider control register (DIV24.5) required:

```
Ideal DIV24.5 = 40 [MHz] / 1,843,200 [Hz] = 21.7014
```

However, the DIV24.5 register has 24 bits for the integer part and limited 5 bits for the fraction part (based 1/32). Therefore, the real divider value and the real UART baud rate can be calculated as follows:

```
Real DIV24.5 = 21.6875 (integer: 21, fractional: 22/32)
Real UART baud rate = 40 [MHz] / 21. 6875 / 16 = 115,274 [bps]
```

### 5.1.4 Use case

This section explains an example of the UART mode.

You can configure these parameters (UART port, baud rate, etc.)in the **Peripherals** tab of the **Device Configurator**; based on these, ModusToolbox™ software automatically generates the corresponding code.

1. Double-click the Device Configurator on the left Quick Panel as shown in Figure 16. or you can also double-click the *design.modus* file to open the Device Configurator in your project, *\bsps\TARGET_APP_KIT_XMC72_EVK\config\design.modus*.

## UART setting procedure example

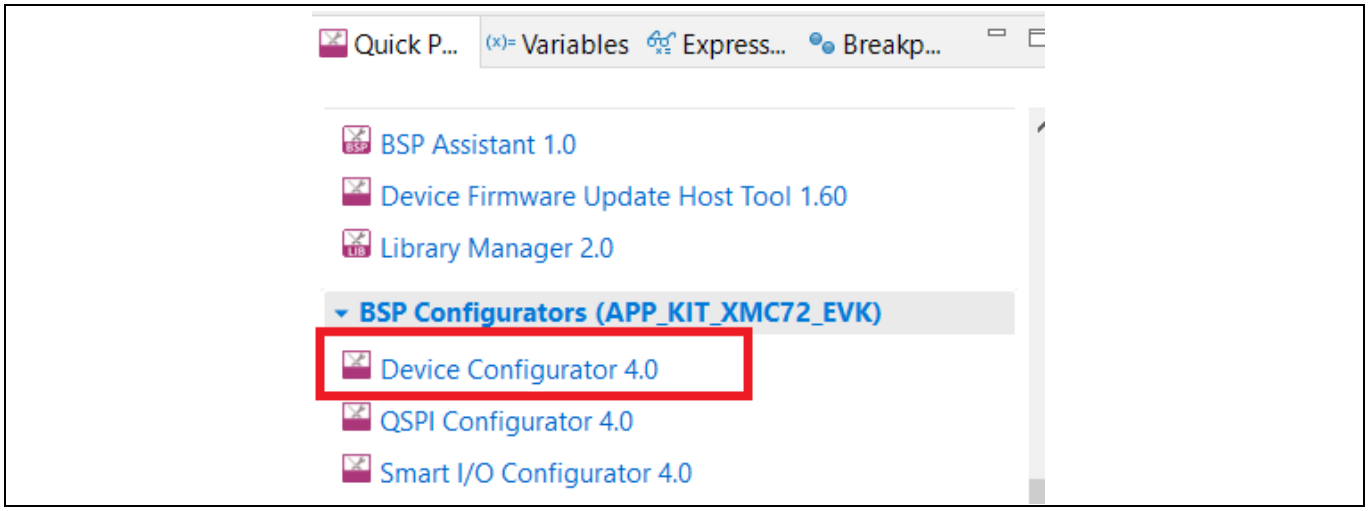


**Figure 16 Device Configurator in ModusToolbox™**

2. Configure SCB mode and channel, baud rate and ports in the **Peripherals** tab of the Device Configurator as shown in Figure 17.
   - Select SCB3 channel to UART mode
   - Data Rate: 115200
   - Clock: 16 bit Divider 0 clk
   - RX: P13[0]
   - TX: P13[1]

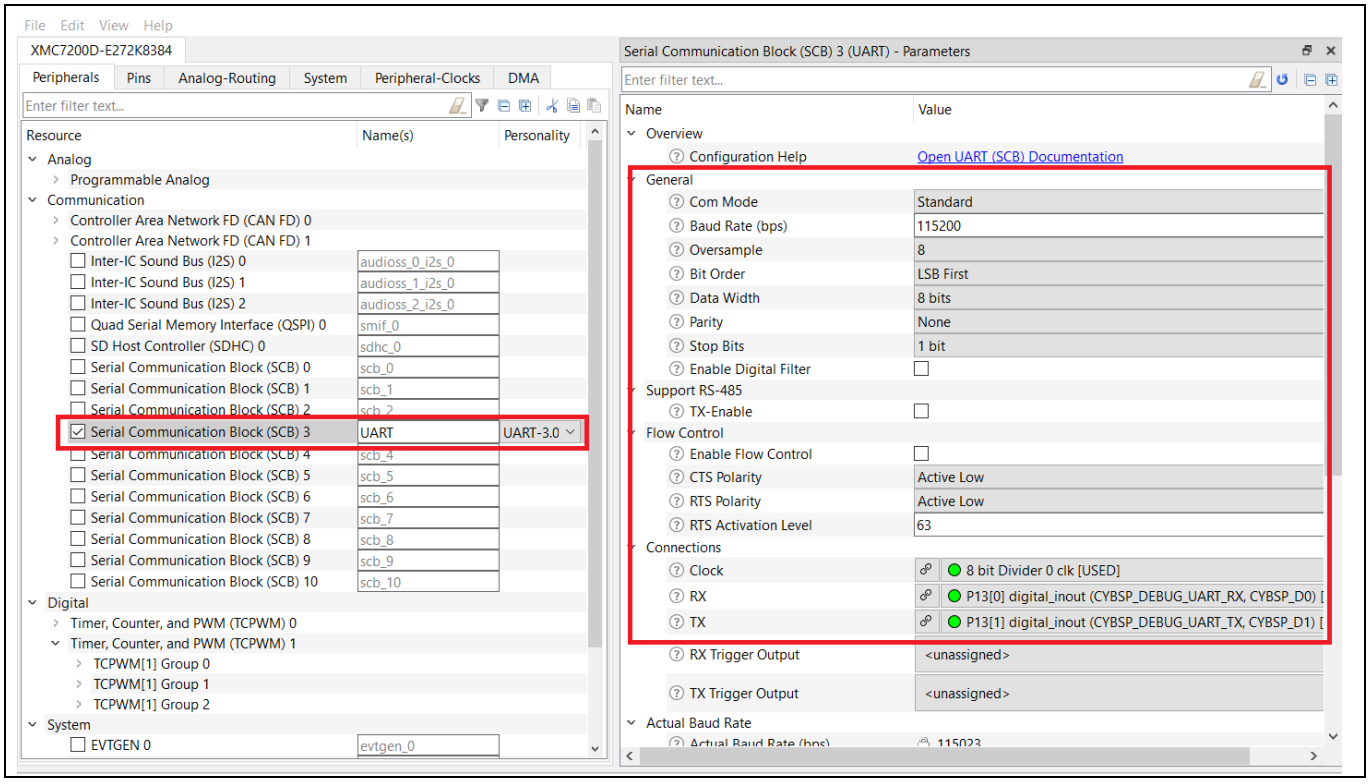


**Figure 17 SCB configuration in ModusToolbox™**

## UART setting procedure example

**Code Listing 3 Example of UART configuration using PDL**

```
cy_stc_sysint_t uart_irq_cfg =
{
    .intrSrc = ((NvicMux3_IRQn << 16) | UART_IRQ),
    .intrPriority = 2UL
};
void Isr_uart_fifo(void)
{
      /*Just call PDL UART intr handler*/
      Cy_SCB_UART_Interrupt(UART_HW, &UART_context);
}

void cb_scb_uart_handle_events (uint32_t event)
{
      if(event == CY_SCB_UART_RECEIVE_DONE_EVENT)
      {
            data_received = 1;
      }
}
int main(void)
{
    cy_rslt_t result;
    cy_en_scb_uart_status_t init_status;
    cy_en_scb_uart_status_t errorStatus;

    /* Initialize the device and board peripherals */
    result = cybsp_init();
    if (result != CY_RSLT_SUCCESS)
    {
        handle_error();
    }

    __enable_irq();

    /* Start UART operation. */
    init_status = Cy_SCB_UART_Init(UART_HW, &UART_config, &UART_context);
    Cy_SCB_SetRxInterruptMask(UART_HW, CY_SCB_UART_RX_TRIGGER);
    Cy_SCB_UART_DisableCts(UART_HW);

    if(init_status!=CY_SCB_UART_SUCCESS)
    {
        handle_error();
    }

    Cy_SCB_UART_RegisterCallback(UART_HW,
(cy_cb_scb_uart_handle_events_t)cb_scb_uart_handle_events, &UART_context);
    Cy_SCB_UART_Enable(UART_HW);
    /* Interrupt Settings for UART */
    Cy_SysInt_Init(&UART_SCB_IRQ_cfg, Isr_uart_fifo);

    /* Enable the interrupt */
    NVIC_EnableIRQ((IRQn_Type) NvicMux3_IRQn);


    /* Transmit header to the terminal. */
      Cy_SCB_UART_PutString(UART_HW,
"\r\n**********************************************************************
****\r\n");
      Cy_SCB_UART_PutString(UART_HW, " UART example using FIFO mode.\r\n");
```

## UART setting procedure example

**Code Listing 3 Example of UART configuration using PDL**

```
      Cy_SCB_UART_PutString(UART_HW, " Type 10 characters you should see them echo
in the terminal.\r\n");
    Cy_SCB_UART_PutString(UART_HW,
"\r\n**************************************************************************
****\r\n");

    /* Initialize flags */

   errorStatus = Cy_SCB_UART_Receive(UART_HW, (void *)uart_buf,
(uint32_t)sizeof(uart_buf), &UART_context);
    if(errorStatus == CY_SCB_UART_SUCCESS)
    {
            for(;;)
            {
                  /* Handle received data from terminal */
                  if(data_received == 1)
                  {
                        /* Get the character from terminal */
                        data_received = 0;
                        Cy_SCB_UART_PutString(UART_HW, (const char*)uart_buf);
                        Cy_SCB_UART_PutString(UART_HW, "\r\n");
                        break;
                  }
            }
      }

    while(!Cy_SCB_IsTxComplete(UART_HW))
    {
    }

      /* Disable the interrupt */
    NVIC_DisableIRQ(UART_SCB_IRQ_cfg.intrSrc);
      Cy_SCB_UART_Disable(UART_HW, &UART_context);
      Cy_SCB_UART_DeInit(UART_HW);
}
```

# 6 I²C setting procedure example

This example shows the usage of the SCB in I²C mode. The SCB supports Master mode, Slave mode, and multi-Master mode. See the architecture reference manual for details of each protocol.

## 6.1 Master mode

In this example, the SCB is configured as an I²C Master and writes one byte data to the Slave (address = 0x08) and reads one byte data from the same Slave. For simplicity, polling method is used in this example instead of interrupts for writing and reading data to/from FIFOs.

**Use case**

- SCB Mode = I²C Master mode
- SCB Channel = 1
- PCLK = 2 MHz
- Bit rate = 100 kbps
- 7-bit Slave address = 0x8 (for another I²C device)
- MSb first
- Tx/Rx FIFO = Yes
- Tx/Rx interrupt = Disabled
- Analog filter is enabled and digital filter = Disabled
- Used ports
  - SCL: SCB0_SCL (P18.2)
  - SDA: SCB0_SDA (P18.1)

Figure 18 shows the example of connection between the SCB and another I²C Slave device.

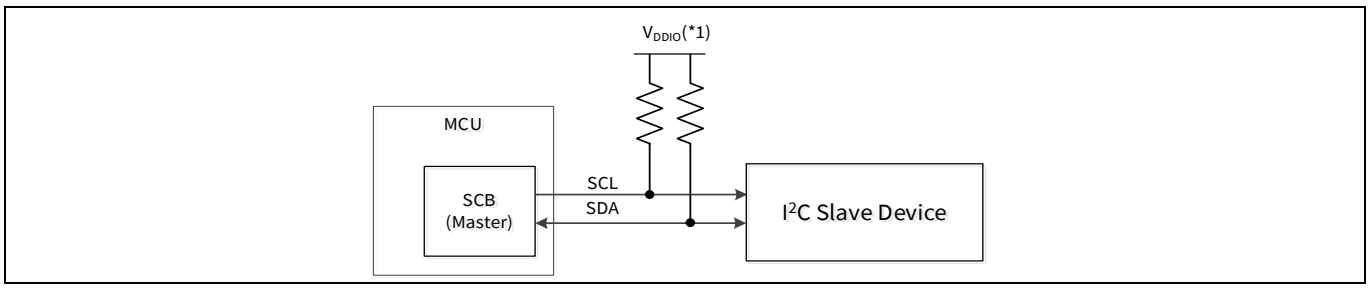


**Figure 18 Example of I²C (Master mode) communication**

*Note: (*1) For $V_{DDIO}$ value, see the datasheet (see References).*

In I²C Master mode, SCL and SDA signals are connected to another I²C Slave device. The Master device outputs the clock (SCL) to the Slave device. The data signal (SDA) is bidirectional. Both SCL and SDA are pulled up to $V_{DDIO}$ via external pull-up register.

### I2C setting procedure example

Figure 19 shows setting procedure and operation example for I²C Master mode.

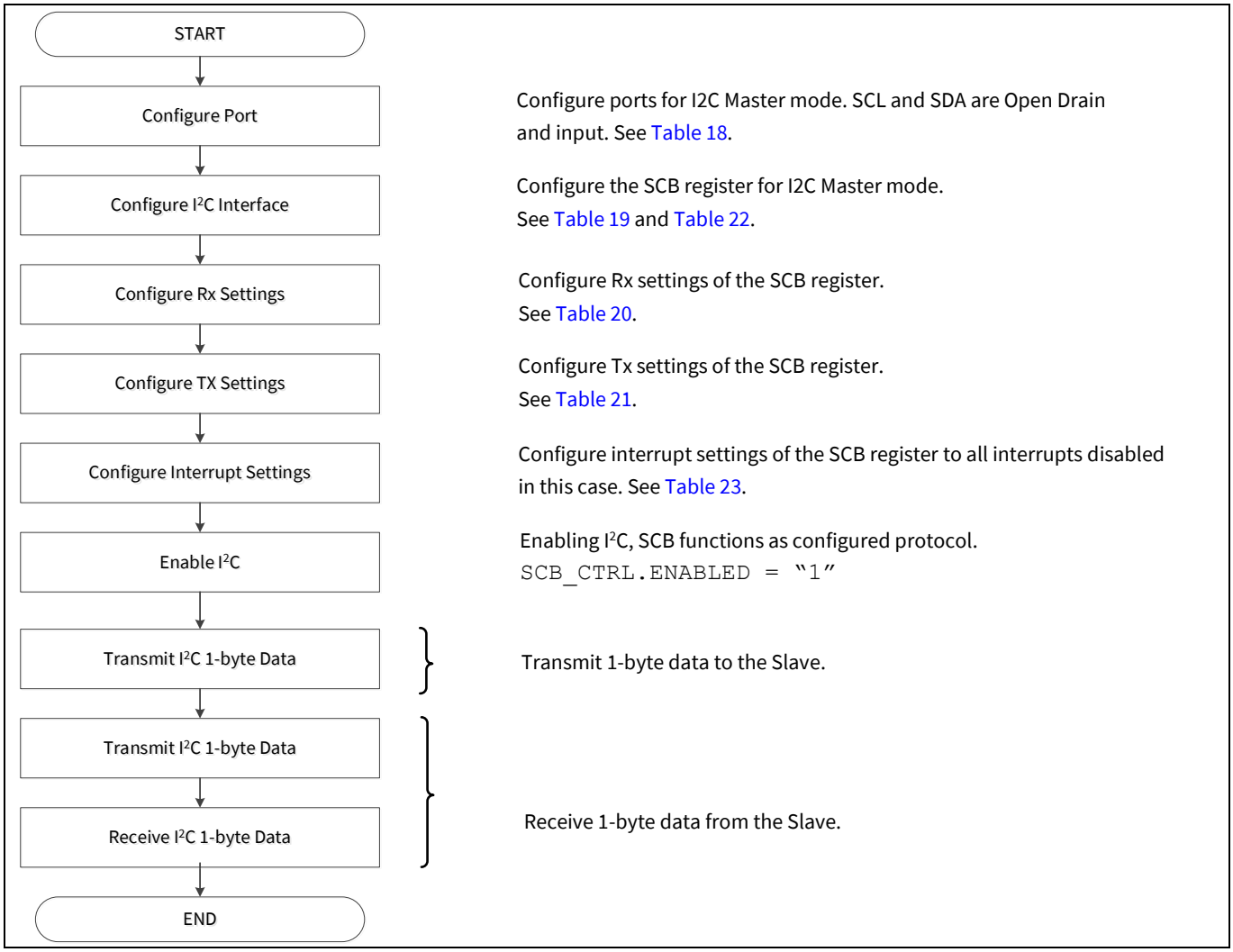


**Figure 19 I²C Master mode operation**

After the Common settings (clock, port, interrupt, and so on), the interface register of the SCB is configured for I²C. If the SCB_CTRL.ENABLE register is set "1", the SCB is ready to transmit and receive the data. Then, the I²C Master writes one byte to the Slave and reads one byte from the Slave.

### I2C setting procedure example

Figure 20 shows a procedure example of I²C Master transmission.

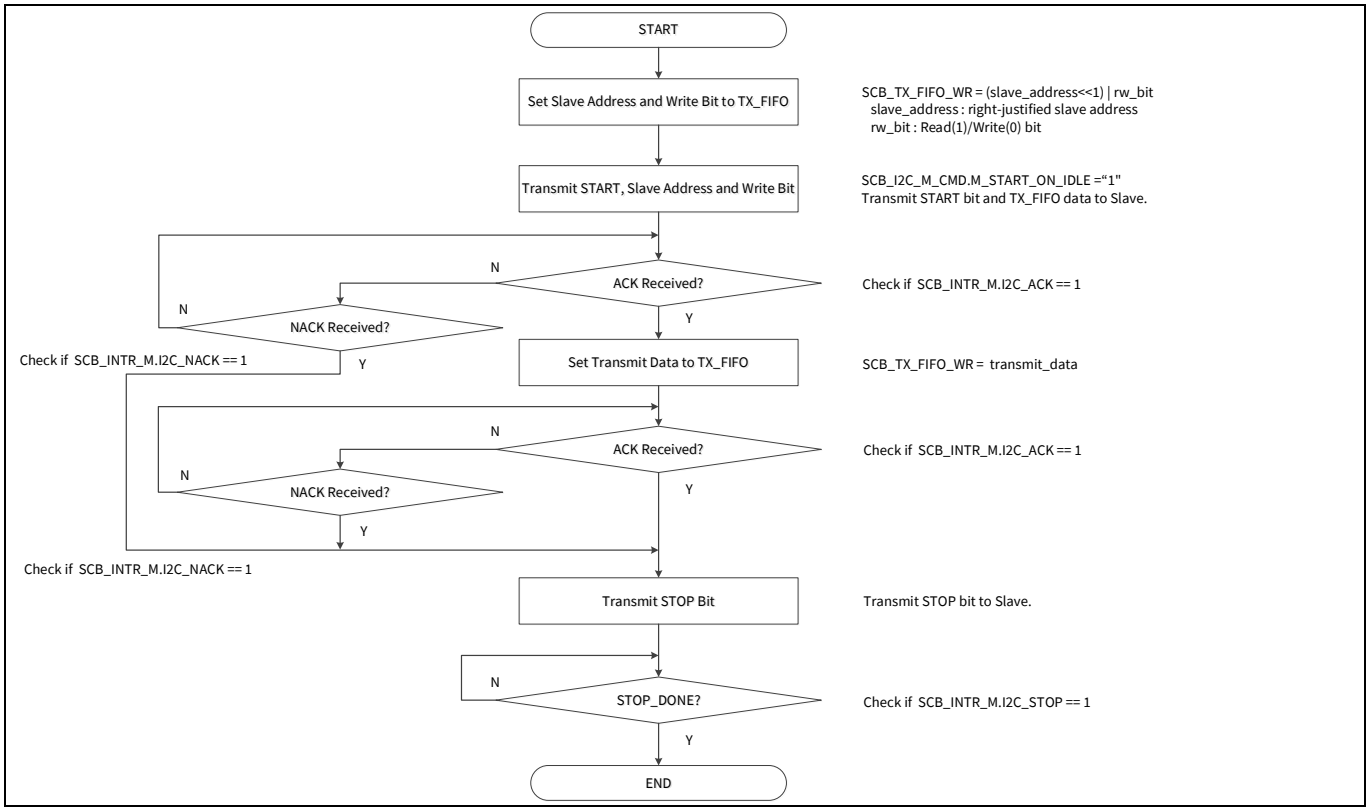


**Figure 20 Example for I²C Master data transmission**

After transmitting the first byte consisting of the Slave address and the write bit, the firmware polls the SCB_INTR_M.I2C_ACK/NACK bit until an ACK/NACK is received from the Slave. If ACK is received, the data byte is transmitted before sending the stop bit. If the NACK is received, a STOP is issued on the I2C bus to terminate the bus transfer.

*Note: Handling of I²C bus errors/arbitration loss is not illustrated in the above example.*

### I2C setting procedure example

Figure 21 shows a procedure example of I²C Master reception.

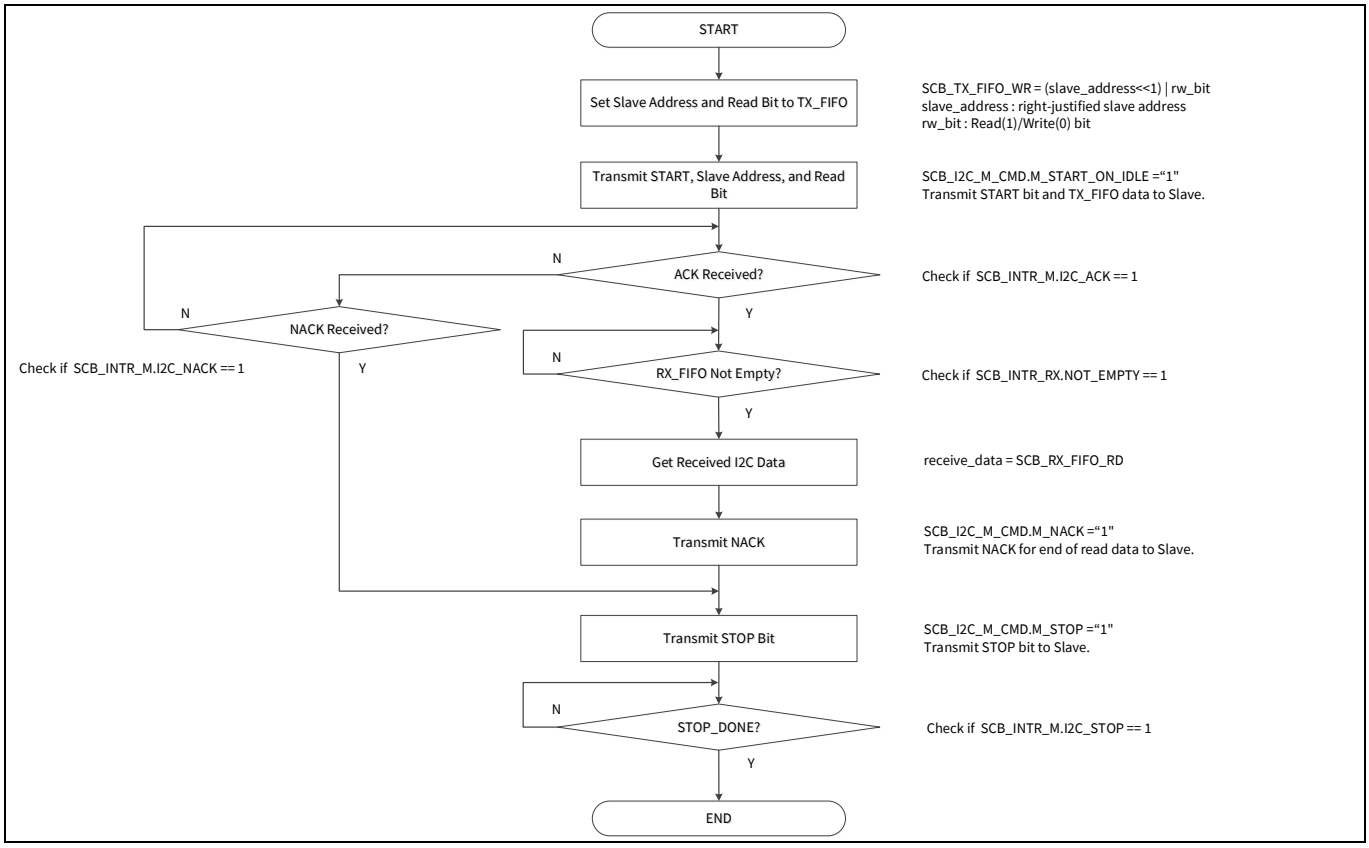


**Figure 21 Example for I²C Master mode reception**

After transmitting the first byte consisting of the Slave address and the read bit, the firmware polls the SCB_INTR_M.I2C_ACK/NACK bit until an ACK/NACK is received from the Slave. Firmware then waits for the data to be available in the Rx FIFO (RX_FIFO not empty), and then transmits NACK and STOP to the Slave to finish the read operation.

*Note: Handling of I²C bus errors/arbitration loss is not illustrated in the above example.*

### 6.1.1 Configure ports

This section explains an example of the port setting used in I²C Master mode. In this mode, SCL and SDA are used as interface signals. Each signal is assigned to the port number as shown follows:

- SCL: SCB0_SCL (P1.0)
- SDA: SCB0_SDA (P1.1)

Table 18 shows an example of the port configuration in I²C Master mode.

SCL and SDA are configured for open-drain and input port with the GPIO_PRT1_CFG.DRIVE_MODEx register and the GPIO_PRT1_CFG.IN_ENx register. The pin functions are determined by the HSIOM_PRT1.PORT_SEL0.IOx_SEL register.

**Table 18 I²C (Master mode): Example of port configurations**

| Register | Port configuration | | Remark |
|---|---|---|---|
| | SCL (*x*=0) | SDA (*x*=1) | |
| GPIO_PRT1_CFG.DRIVE_MODEx | 4 | 4 | 4: Open Drain, Drives Low |
| GPIO_PRT1_CFG.IN_ENx | 1 | 1 | 1: Input buffer enabled |
| HSIOM_PRT1.PORT_SEL0.IOx_SEL | 14 | 14 | 14: DS#2 |

*Note: Bits that are not listed Table 18 have default values. For default values, refer the respective registers in the registers reference manual.*

### 6.1.2 Configure I²C interface registers

This section explains an example of the I²C register setting used in I²C Master mode. The following I²C registers are used: interface configuration register, Rx and Tx control register, and interrupt register.

Table 19 shows an example of the I²C interface configuration in I²C Master mode. These registers determine the behavior of the SCB. In this case, the SCB is configured in I²C Master mode. This is not a multi-Master mode, so the SCB_I2C_CTRL.SLAVE_MODE bit set to disable.

SCB_I2C_CTRL.HIGH_PHASE_OVS and SCB_I2C_CTRL.LOW_PHASE_OVS determine the I²C bit rate. For more details, see 6.1.3 Bit rate setting.

**Table 19 I²C (Master mode): Example of I²C interface configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_CTRL | MODE | 0 | I²C mode |
| | ADDR_ACCEPT | 0 | Slave address not accepted |
| | MEM_WIDTH | 0 | 8-bit FIFO data elements |
| | EC_AM_MODE | 0 | Internally clocked mode |
| SCB_I2C_CTRL | MASTER_MODE | 1 | Master mode |
| | SLAVE_MODE | 0 | Disabled |
| | HIGH_PHASE_OVS | 9 | HIGH phase oversampling factor is 10. |
| | LOW_PHASE_OVS | 9 | LOW phase oversampling factor is 10. |

### I2C setting procedure example

*Note: Bits that are not listed in Table 19 have default values. For default values, see the respective registers in the registers reference manual.*

Table 20 shows an example of the Rx Configuration in I²C Master mode. These registers configure the receiver control settings. In this case, the received data format is MSb first, the data width is 8 bits, and digital median filter is disabled.

**Table 20 I²C (Master mode): Example of Rx configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_RX_CTRL | MSB_FIRST | 1 | MSb first |
| | DATA_WIDTH | 7 | DATA_WIDTH + 1 (=8) is the expected number of bits in received data frame. |
| | MEDIAN | 0 | Digital median filter is disabled. |

*Note: Note: Bits that are not listed in Table 20 have default values. For default values, see the respective registers in the registers reference manual.*

Table 21 shows an example of the Tx configuration in I²C Master mode. These registers configure the transmitter control settings. In this case, the transmitter data format is MSb first, the data width is 8 bits, and OPEN_DRAIN mode is set.

**Table 21 I²C (Master mode): Example of Tx configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_TX_CTRL | MSB_FIRST | 1 | MSb first |
| | DATA_WIDTH | 7 | DATA_WIDTH + 1 (=8) is the number of bits in a transmitted data frame. |
| | OPEN_DRAIN | 1 | Open drain operation mode |

*Note: Bits that are not listed in Table 21 have default values. For default values, see the respective registers in the registers reference manual.*

Table 22 shows an example of the analog filter configuration in I²C Master mode. These registers configure the analog filter settings to remove glitches. In this case, the filters of SDA_IN and SCLK_IN are enabled; SDA_OUT is disabled.

**Table 22 I²C (Master mode): Example of analog filter configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_I2C_CFG | SDA_IN_FILT_SEL | 1 | SDA input filter is enabled. |
| | SCL_IN_FILT_SEL | 1 | SCL input filter is enabled. |
| | SDA_OUT_FILT_SEL | 0 | SDA output filter is disabled. |

*Note: Bits that are not listed in Table 22 have default values. For default values, see the respective registers in the registers reference manual.*

**I2C setting procedure example**

Table 23 shows an example of the interrupt configuration in I²C Master mode. These registers configure the interrupt mask settings. In this case, Tx and Rx interrupts are not used, because the software polls the status bits of the SCB_INTR_M register.

**Table 23 I²C (Master mode): Example of interrupt configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_INTR_I2C_EC_MASK | – | 0 | Externally clocked I²C interrupt request is masked. |
| SCB_INTR_M_MASK | – | 0 | Master interrupt is masked. |
| SCB_INTR_S_MASK | – | 0 | Slave interrupt is masked. |
| SCB_INTR_TX_MASK | – | 0 | Transmitter interrupt is masked. |
| SCB_INTR_RX_MASK | – | 0 | Receiver interrupt is masked. |

*Note: Bits that are not listed in Table 23 have default values. For default values, see the respective registers in the registers reference manual.*

## 6.1.3 Bit rate setting

The bit rate setting is valid only in Master mode. The bit rate calculation formula is as follows:

```
Bit rate [bps] = Input Clock [Hz] / (Low_phase_ovs + High_phase_ovs)
     Low_phase_ovs : SCB_I2C_CTRL.LOW_PHASE_OVS + 1
     High_phase_ovs : SCB_I2C_CTRL.HIGH_PHASE_OVS + 1
```

In this case, bit rate is calculated as follows:

```
       Bit rate     = Input Clock [Hz] / (High_phase_ovs + Low_phase_ovs)
      = PCLK(2MHz) / ((9+1) + (9+1)) = 100 [kbps]
```

For more details, see the architecture reference manual.

## 6.1.4 Use case

This section explains an example of the I2C Master mode.

You can configure these parameters (I2C ports, data rate, etc.) in the **Peripherals** tab of the **Device Configurator**; based on these, ModusToolbox™ software automatically generates the corresponding code.

1. Double-click the Device Configurator on the left Quick Panel as shown in Figure 22 or you can also double-click the *design.modus* file to open the Device Configurator in your project, *\bsps\TARGET_APP_KIT_XMC72_EVK\config\design.modus*.

### I2C setting procedure example

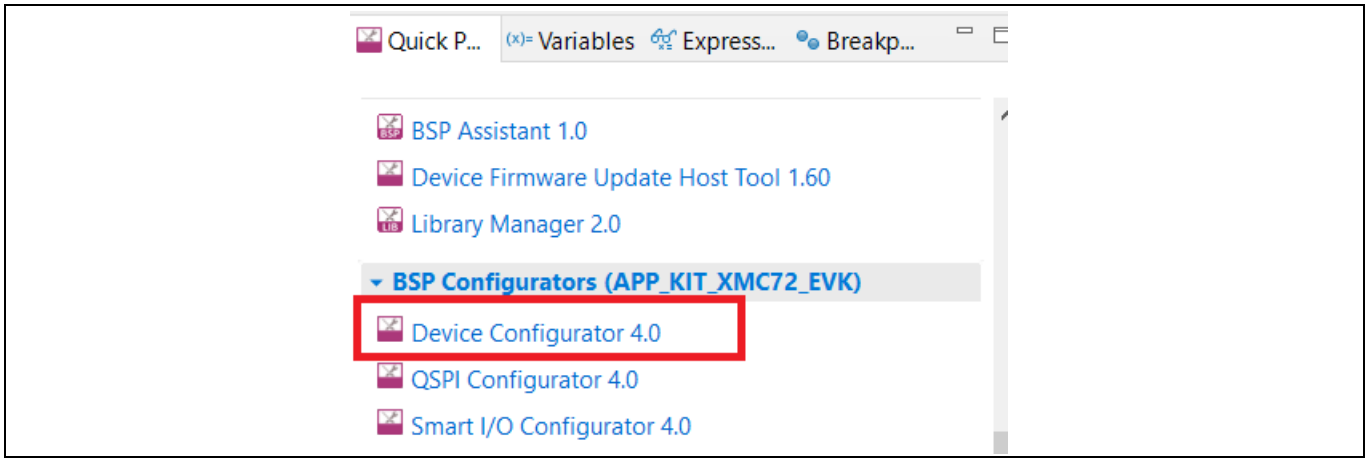


**Figure 22 Device Configurator in ModusToolbox™**

2. Configure SCB mode and channel, data rate and ports in the **Peripherals** tab of the Device Configurator as shown in Figure 23.
   - Select SCB1 channel to I2C mode
   - Mode: Master
   - Data Rate: 100
   - Clock: 16 bit Divider 0 clk
   - SCL: P18[2]
   - SDA: P18[1]

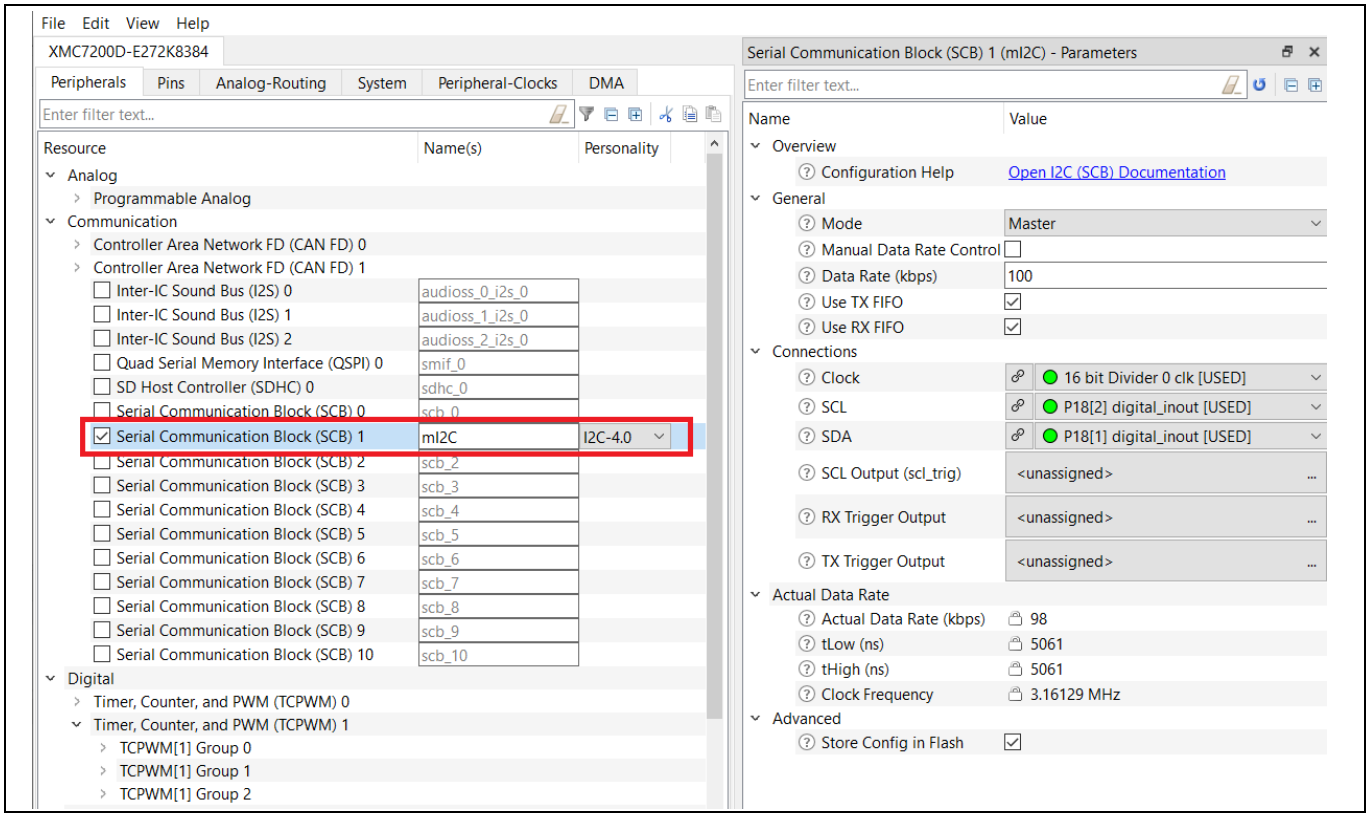


**Figure 23 SCB configuration in ModusToolbox™**

## I2C setting procedure example

**Code Listing 4 Example of basic I2C Master configuration using PDL**

```
cy_stc_sysint_t mI2C_SCB_IRQ_cfg =
{
    .intrSrc = ((NvicMux3_IRQn << 16) | mI2C_IRQ), /* Bit 0-15 of intrSrc is used
to store system interrupt value and bit 16-31 to store CPU IRQ value */
    .intrPriority = 2UL,
};

/* Structure for master transfer configuration */

cy_stc_scb_i2c_master_xfer_config_t masterTransferCfg =
{
    .slaveAddress = I2C_SLAVE_ADDR,
    .buffer       = NULL,
    .bufferSize   = 0U,
    .xferPending  = false
};

cy_stc_scb_i2c_context_t mI2C_context;

void mI2C_Interrupt(void)
{
    Cy_SCB_I2C_Interrupt(mI2C_HW, &mI2C_context);
}

uint8_t WritePacket(uint8_t* buffer, uint32_t bufferSize)
{
    uint8_t status = TRANSFER_ERROR;
    cy_en_scb_i2c_status_t  errorStatus;
    uint32_t masterStatus;
    /* Timeout 1 second (one unit is us) */
    int32_t timeout =  1000000UL;

    /* Setup transfer specific parameters. */
    masterTransferCfg.buffer     = buffer;
    masterTransferCfg.bufferSize = bufferSize;
      masterTransferCfg.slaveAddress = I2C_SLAVE_ADDR;

    /* Initiate write transaction. */
    errorStatus = Cy_SCB_I2C_MasterWrite(mI2C_HW, &masterTransferCfg,
&mI2C_context);
    if(errorStatus == CY_SCB_I2C_SUCCESS)
    {
        /* Wait until master complete read transfer or time out has occurred */
        do
        {
            masterStatus  = Cy_SCB_I2C_MasterGetStatus(mI2C_HW, &mI2C_context);
            Cy_SysLib_DelayUs(CY_SCB_WAIT_1_UNIT);
            timeout--;
        } while ((0UL != (masterStatus & CY_SCB_I2C_MASTER_BUSY)) && (timeout >
0));

        if (timeout <= 0)
        {
            /* Timeout recovery */
            Cy_SCB_I2C_Disable(mI2C_HW, &mI2C_context);
            Cy_SCB_I2C_Enable (mI2C_HW);
        }
        else
```

## I2C setting procedure example

**Code Listing 4 Example of basic I2C Master configuration using PDL**

```
        {
            if ((0u == (MASTER_ERROR_MASK & masterStatus)) &&
                (TX_PACKET_SIZE == Cy_SCB_I2C_MasterGetTransferCount(mI2C_HW,
&mI2C_context)))
            {
                status = TRANSFER_CMPLT;
            }
        }
    }

    return (status);
}

uint8_t ReadStatusPacket(void)
{
    uint8_t status = TRANSFER_ERROR;
    cy_en_scb_i2c_status_t errorStatus;
    uint32_t masterStatus;
    /* Timeout 1 second (one unit is us) */
    uint32_t timeout = 1000000UL;
    uint8_t  rxBuffer[RX_PACKET_SIZE];

    /* Setup transfer specific parameters */
    masterTransferCfg.buffer     = rxBuffer;
    masterTransferCfg.bufferSize = RX_PACKET_SIZE;
      masterTransferCfg.slaveAddress = I2C_SLAVE_ADDR;

    /* Initiate read transaction */
    errorStatus = Cy_SCB_I2C_MasterRead(mI2C_HW, &masterTransferCfg,
&mI2C_context);
    if(errorStatus == CY_SCB_I2C_SUCCESS)
    {
        /* Wait until master complete read transfer or time out has occurred */
        do
        {
            masterStatus  = Cy_SCB_I2C_MasterGetStatus(mI2C_HW, &mI2C_context);
            Cy_SysLib_DelayUs(CY_SCB_WAIT_1_UNIT);
            timeout--;

        } while ((0UL != (masterStatus & CY_SCB_I2C_MASTER_BUSY)) && (timeout >
0));
       if (timeout <= 0)
        {
            /* Timeout recovery */
            Cy_SCB_I2C_Disable(mI2C_HW, &mI2C_context);
            Cy_SCB_I2C_Enable (mI2C_HW);
        }
        else
        {
            /* Check transfer status */
            if (0u == (MASTER_ERROR_MASK & masterStatus))
            {
                /* Check packet structure and status */
                if((PACKET_SOP   == rxBuffer[RX_PACKET_SOP_POS]) &&
                    (PACKET_EOP   == rxBuffer[RX_PACKET_EOP_POS]) &&
                    (STS_CMD_DONE == (rxBuffer[RX_PACKET_STS_POS])))
                {
                    status = TRANSFER_CMPLT;
```

## I2C setting procedure example

**Code Listing 4 Example of basic I2C Master configuration using PDL**

```
                }
            }
        }
    }
    return (status);
}

int main(void)
{
    cy_rslt_t result;
    uint8_t cmd = ON;
    uint8_t  buffer[TX_PACKET_SIZE];
      cy_en_scb_i2c_status_t initStatus;
      cy_en_sysint_status_t sysStatus;

    /* Initialize the device and board peripherals */
    result = cybsp_init();
    /* Board init failed. Stop program execution */
    handle_error(result);

    /* Initialize the retarget-io */
    result = cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,
                                 CY_RETARGET_IO_BAUDRATE);
    /* Retarget-io init failed. Stop program execution */
    handle_error(result);

    /* \x1b[2J\x1b[;H - ANSI ESC sequence for clear screen */
    printf("\x1b[2J\x1b[;H");

    printf("****************** "
           "PDL: I2C Master "
           "****************** \r\n\n");

    /* I2C Master configuration settings */
    printf(">> Configuring I2C Master..... ");

    /* Init I2C master */
    initStatus = Cy_SCB_I2C_Init(mI2C_HW, &mI2C_config, &mI2C_context);
    if(initStatus != CY_SCB_I2C_SUCCESS)
    {
      return I2C_FAILURE;
    }

    /* Hook interrupt service routine. */
    sysStatus = Cy_SysInt_Init(&mI2C_SCB_IRQ_cfg, &mI2C_Interrupt);
    if(sysStatus != CY_SYSINT_SUCCESS)
    {
      return I2C_FAILURE;
    }

    /* Enable interrupt in NVIC. */
    NVIC_EnableIRQ((IRQn_Type) NvicMux3_IRQn);
      /* Enable I2C master hardware. */
      Cy_SCB_I2C_Enable(mI2C_HW);

    printf("Done\r\n\n");

    /* Enable interrupts */
```

### I2C setting procedure example

**Code Listing 4 Example of basic I2C Master configuration using PDL**

```
    __enable_irq();

    for (;;)
    {
        /* create packet to be sent to slave.  */
        buffer[PACKET_SOP_POS] = PACKET_SOP;
        buffer[PACKET_EOP_POS] = PACKET_EOP;
        buffer[PACKET_CMD_POS] = cmd;

        uint8_t res = WritePacket(buffer, TX_PACKET_SIZE);

        /* Send packet with command to the slave. */
            if (TRANSFER_CMPLT == res)
            {
                  printf("Write complete\n");
                  /* Read response packet from the slave. */
                  if (TRANSFER_CMPLT == ReadStatusPacket())
                  {
                        printf("Read Complete\n");
                        /* Next command to be written. */
                        cmd = (cmd == ON) ? OFF : ON;
                  }
                  /* Give 1 Second delay between commands. */
                  Cy_SysLib_Delay(CMD_TO_CMD_DELAY);
            }

    }
}
```

## 6.2 Slave mode

This example sets I²C Slave mode where the Master transmits the write or read data to the Slave SCB. If the Slave receives the data, an interrupt occurs, and the Slave decides whether it should perform the read or write procedure.

**Use case**

- SCB Mode = I²C Slave mode
- SCB Channel = 1
- PCLK = 2 MHz
- Bit rate = 100 kbps
- 7-bit Slave address = 0x8
- Tx/Rx FIFO = Yes
- MSb first
- Data width = 8 bits
- Analog filter is enabled and digital filter = Disabled
- Enabled interrupts:
  - I2C_ARB_LOST (I²C Slave arbitration lost)
  - I2C_STOP (I²C STOP event detected)
  - I2C_ADDR_MATCH (I²C Slave address matching)

### I2C setting procedure example

  - I2C_GENERAL (I²C Slave general call address received)
  - I2C_BUS_ERROR ((I²C Slave bus error detected)
- Used ports
  - SCL: SCB0_SCL (P18.2)
  - SDA: SCB0_SDA (P18.1)

Figure 24 shows the example of the connection between the Slave SCB and another I²C Master device.

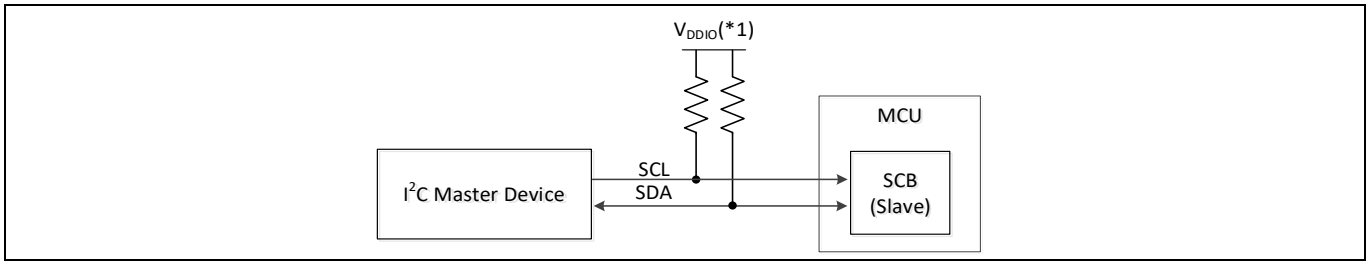


**Figure 24 Example of I²C (Slave mode) communication connection**

*Note: (*1) For $V_{DDIO}$ value, see the datasheet (see References).*

In I²C Slave mode, SCL and SDA signals are connected to another I²C Master device. The Master device outputs the clock (SCL) to the Slave device. The data (SDA) signal is bidirectional. Both SCL and SDA are pulled up to $V_{DDIO}$ via a resistor.

### I2C setting procedure example

Figure 25 shows the setting procedure and operation example for I²C Slave mode.

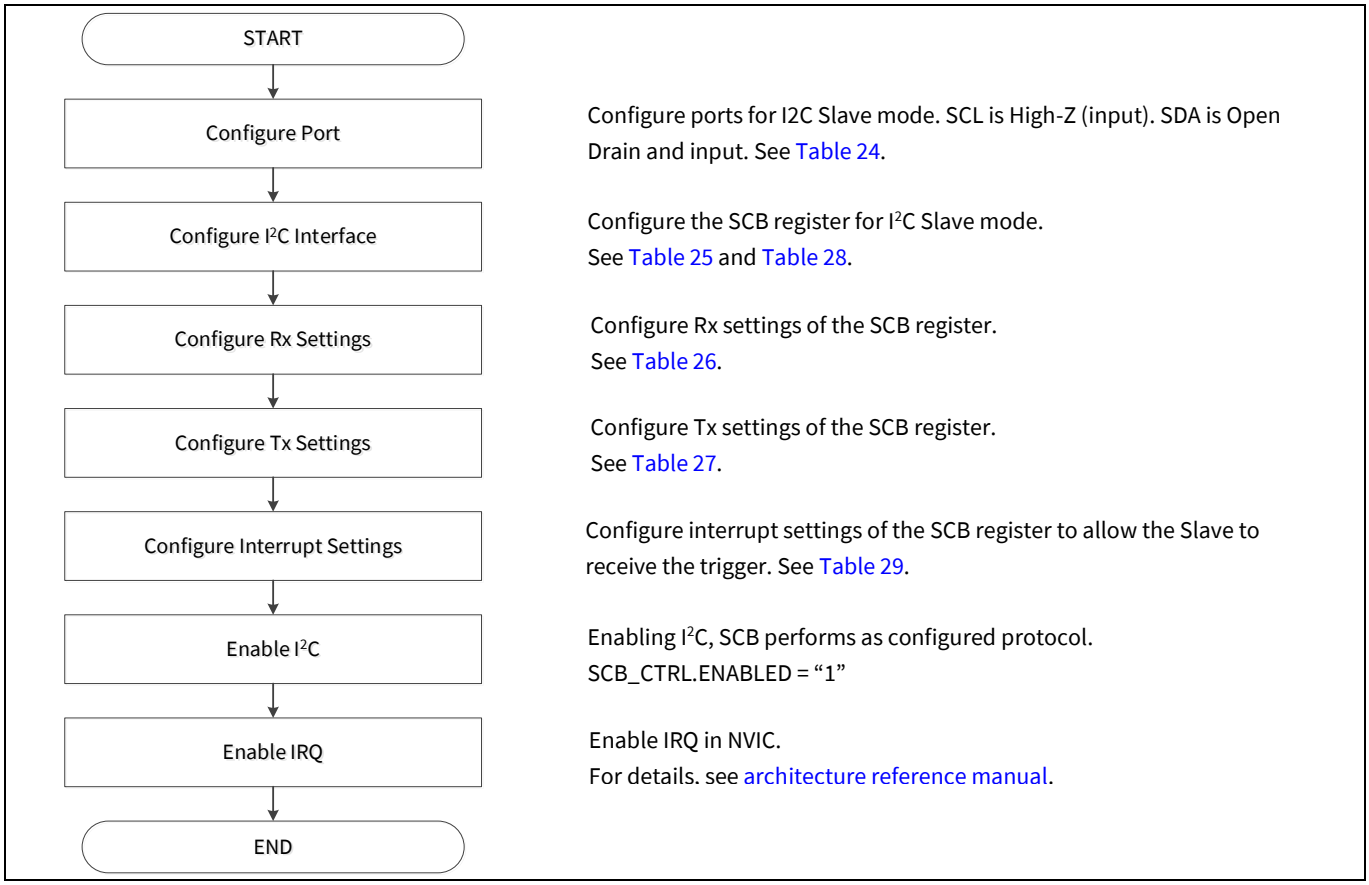


**Figure 25 I²C Slave mode operation**

After the general configuration (clock, port, interrupt, and so on), the interface register of the SCB is configured for I²C. If the SCB_CTRL.ENABLE register is set "1" and interrupt is enabled, the SCB is ready to receive data.

### I2C setting procedure example

Figure 26 shows an example for I²C Slave reception interrupt.

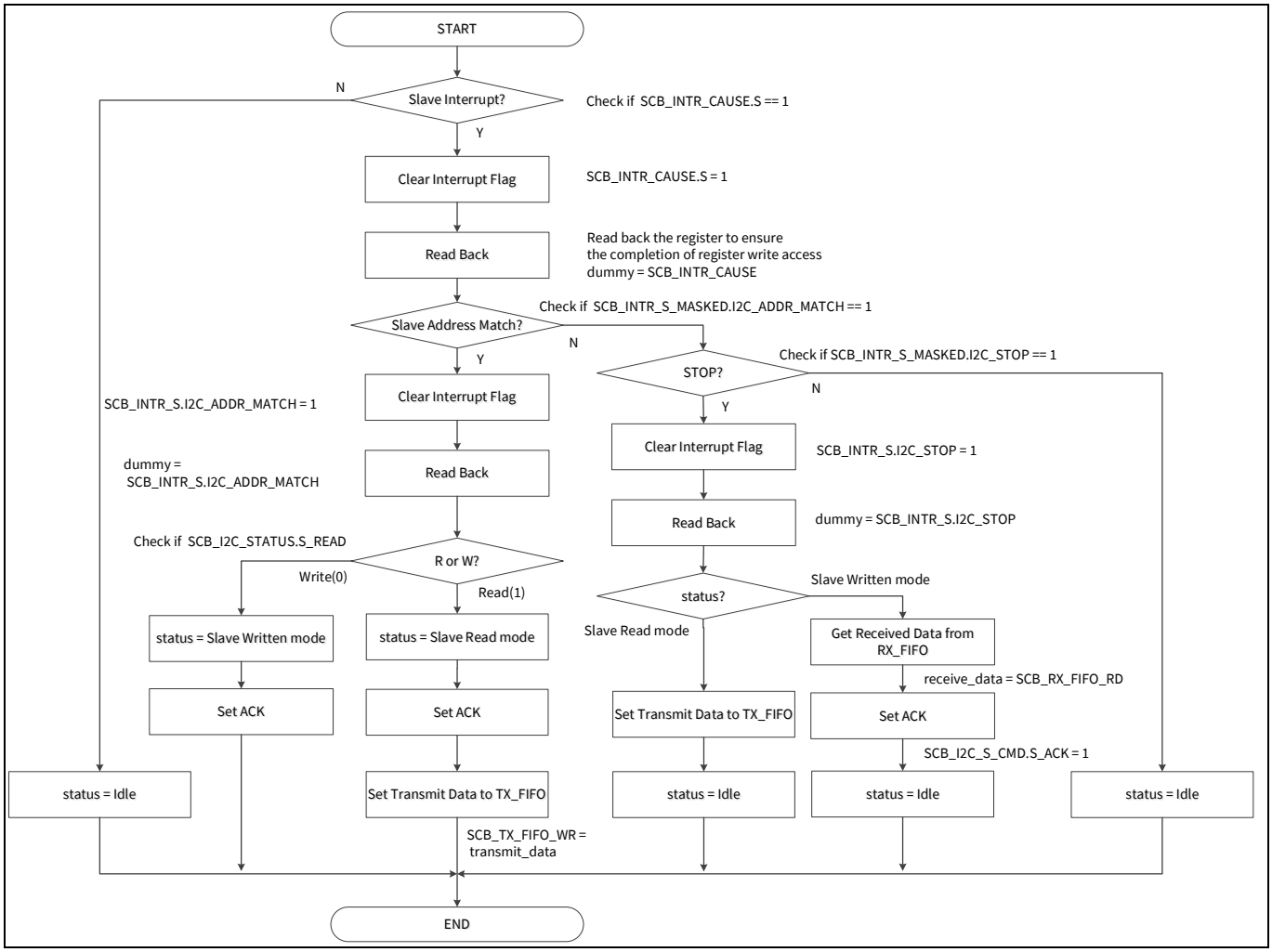


**Figure 26 I²C Slave mode reception interrupt example**

Figure 26 shows the interrupt procedure.

After a Slave interrupt (SCB_INTR_CAUSE.S) occurs, the interrupt flag should be cleared, and the software can check whether the received Slave address matches with the configured Slave address. If the received Slave address matches with the configured Slave address, the received data might be the first byte of the I²C format, and then the software can check whether the direction of the data is write (from Master to Slave) or read (from Slave to Master).

If the received Slave address does not match and STOP interrupt is TRUE, the flow branches based on the status of the first received byte. If the status is Slave Read mode, set the transmit data to the TX_FIFO register as the response data to the Master. If the status is Slave Written mode, the received data may be the written data from the Master; then the software can read out the received data from the RX_FIFO register. If the received Slave address does not match and the STOP interrupt is FALSE, the received data may be invalid.

*Note: Error handling is not explained in Figure 26 and must be handled by the application.*

### 6.2.1 Configure ports

This section explains an example of the port setting used in I²C Slave mode. In this mode, SCL and SDA are used as interface signals. Each signal is assigned to the port number as follows:

- SCL: SCB0_SCL (P1.0)
- SDA: SCB0_SDA (P1.1)

Table 24 shows an example of the port configuration in I²C Slave mode.

SCL and SDA are configured for open-drain and input port with the GPIO_PRT1_CFG.DRIVE_MODEx register and the GPIO_PRT1_CFG.IN_ENx register. The port functions are determined by the HSIOM_PRT1.PORT_SEL0.IOx_SEL register.

**Table 24 I²C (Slave mode): Example of port configurations**

| Register | Port configuration | | Remark |
|---|---|---|---|
| | SCL (*x*=0) | SDA (*x*=1) | |
| GPIO_PRT1_CFG.DRIVE_MODEx | 0 | 4 | 0: High-Z<br>4: Open Drain, Drives Low |
| GPIO_PRT1_CFG.IN_ENx | 1 | 1 | 1: Input buffer enabled |
| HSIOM_PRT1.PORT_SEL0.IOx_SEL | 14 | 14 | 14: DS#2 |

*Note: Bits that are not listed in Table 24 have default values. For default values, see the respective registers in the registers reference manual.*

### 6.2.2 Configure I²C interface registers

This section explains an example of the I²C registers setting used in I²C Slave mode. The following registers are used: interface configuration register, Rx and Tx control register, and Interrupt register.

Table 25 shows an example of the I²C interface configuration in I²C Slave mode. These registers determine the behavior of the SCB. In this case, the SCB is configured in the I²C Slave mode by setting SCB_I2C_CTRL.SLAVE_MODE to “1”.

**Table 25 I²C (Slave mode): Example of I²C interface configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_CTRL | MODE | 0 | I²C mode |
| | ADDR_ACCEPT | 0 | Slave address not accepted |
| | MEM_WIDTH | 0 | 8-bit FIFO data elements |
| SCB_I2C_CTRL | MASTER_MODE | 0 | Disabled |
| | SLAVE_MODE | 1 | Slave mode |
| | S_GENERAL_IGNORE | 0 | General call Slave address enabled |

*Note: Bits that are not listed in Table 25 have default values. For default values, see the respective registers in the registers reference manual.*

### I2C setting procedure example

Table 26 shows an example of the Rx configuration in the I²C Slave mode. These registers configure the receiver control settings. In this case, the received data format is MSb first and the data width is 8 bits. The SCB_RX_MATCH.ADDR register should be set to the Slave device address shifted left by 1 bit. The SCB_RX_MATCH.MASK register is set as is.

**Table 26 I²C (Slave mode): Example of Rx configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_RX_CTRL | MSB_FIRST | 1 | MSb first |
| | DATA_WIDTH | 7 | DATA_WIDTH + 1 (=8) is the expected number of bits in the received data frame. |
| SCB_RX_FIFO_CTRL | TRIGGER_LEVEL | 1 | When the receiver FIFO has more entries than the number of this field, a receiver trigger event is generated. |
| | CLEAR | 0 | Not cleared. |
| | FREEZE | 0 | Not frozen. |
| SCB_RX_MATCH | ADDR | 0x10 | Slave device address = 8.<br>Set the Slave address shifted left by 1 bit because only bits 7 down to 1 (of 7 to 0) are used for the Slave address.<br>In this case,<br>Slave address: 1000b (= 8)<br>Set value: 0001 0000b (= 0x10) |
| | MASK | 0xFE | Slave device address mask. This is left-shifted by one bit. |

*Note: Bits that are not listed in Table 26 have default values. For default values, see the respective registers in the registers reference manual.*

Table 27 shows an example of the Tx Configuration in I²C Slave mode. These registers configure the transmitter control settings. In this case, the transmitter data format is MSb first, the data width is 8 bits, and OPEN_DRAIN mode is set.

**Table 27 I²C (Slave mode): Example of Tx configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_TX_CTRL | MSB_FIRST | 1 | MSb first |
| | DATA_WIDTH | 7 | DATA_WIDTH + 1 (= 8) is the number of bits in a transmitted data frame. |
| | OPEN_DRAIN | 1 | Open-drain operation mode. |
| SCB_TX_FIFO_CTRL | TRIGGER_LEVEL | 64 | When the transmitter FIFO has fewer entries than the number of this field, a transmitter trigger event is generated. |

*Note: Bits that are not listed in Table 27 have default values. For default values, see the respective registers in the registers reference manual.*

**I2C setting procedure example**

Table 28 shows an example of the analog filter configuration in I²C Slave mode. These registers configure the analog filter settings to remove glitches. In this case, the filter of SDA_IN and SCLK_IN are enabled; SDA_OUT is disabled.

**Table 28 I²C (Slave mode): Example of analog filter configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_I2C_CFG | SDA_IN_FILT_SEL | 1 | SDA input filter is enabled. |
| | SCL_IN_FILT_SEL | 1 | SCL input filter is enabled. |
| | SDA_OUT_FILT_SEL | 0 | SDA output filter is disabled. |

*Note: Bits that are not listed in Table 28 have default values. For default values, see the respective registers in the registers reference manual.*

Table 29 shows an example of the interrupt configuration in I²C Slave mode. These registers configure the interrupt mask control settings. In this case, arbitration lost, STOP event, Slave address match, I²C general call address receiving, and I²C bus error are set as receiver interrupts.

**Table 29 I²C (Slave mode): Example of interrupt configurations**

| Register | Bit | Value | Remark |
|---|---|---|---|
| SCB_INTR_I2C_EC_MASK | – | 0 | Externally clocked I²C interrupt request is masked. |
| SCB_INTR_M_MASK | – | 0 | Master interrupt is masked. |
| SCB_INTR_S_MASK | – | 0x1D1 | The following interrupt factors are enabled: I2C_ARB_LOST, I2C_STOP, I2C_ADDR_MATCH, I2C_GENERAL, I2C_BUS_ERROR. |
| SCB_INTR_TX_MASK | – | 0 | Transmitter interrupt is masked. |
| SCB_INTR_RX_MASK | – | 0 | Receiver interrupt is masked. |

*Note: Bits that are not listed in Table 29 have default values. For default values, see the respective registers in the registers reference manual.*

## 6.2.3 Use case

This section explains an example of the I2C Slave mode.

You can configure these parameters (I2C ports, data rate, etc.) in the **Peripherals** tab of the **Device Configurator**; based on these, ModusToolbox™ software automatically generates the corresponding code.

1. Double-click the Device Configurator on the left Quick Panel as shown in Figure 27 or you can also double-click *design.modus* file to open the Device Configurator in your project, *\bsps\TARGET_APP_KIT_XMC72_EVK\config\design.modus*)

### I2C setting procedure example

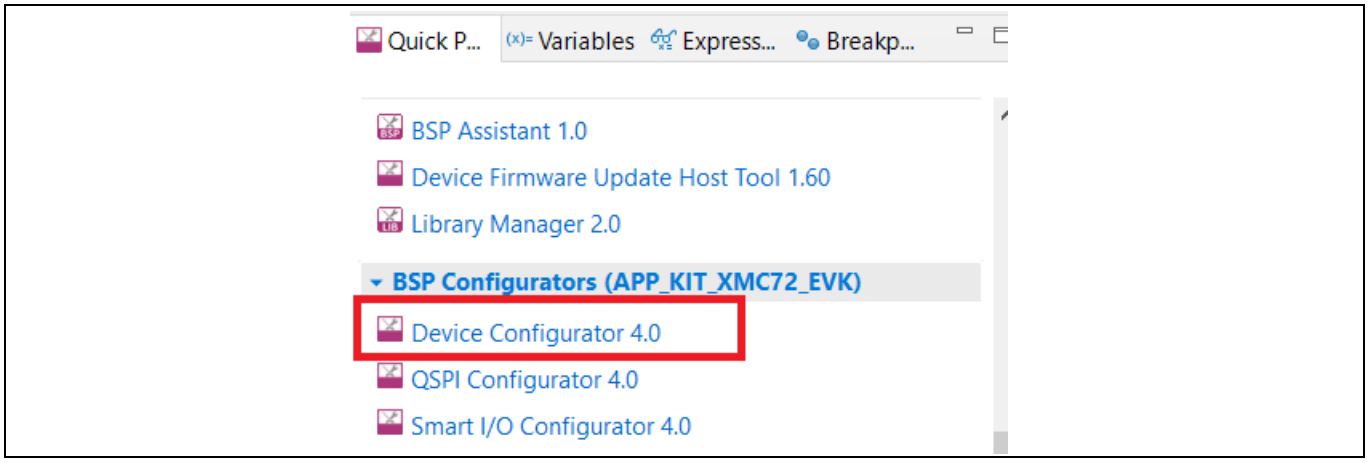


**Figure 27 Device Configuartor in ModusToolbox™**

2. Configure SCB mode and channel, data rate and ports in the **Peripherals** tab of the Device Configurator as shown in Figure 28.
   - Select SCB1 channel to I2C mode
   - Mode: Slave
   - Data Rate: 100
   - Clock: 16 bit Divider 0 clk
   - SCL: P18[2]
   - SDA: P18[1]

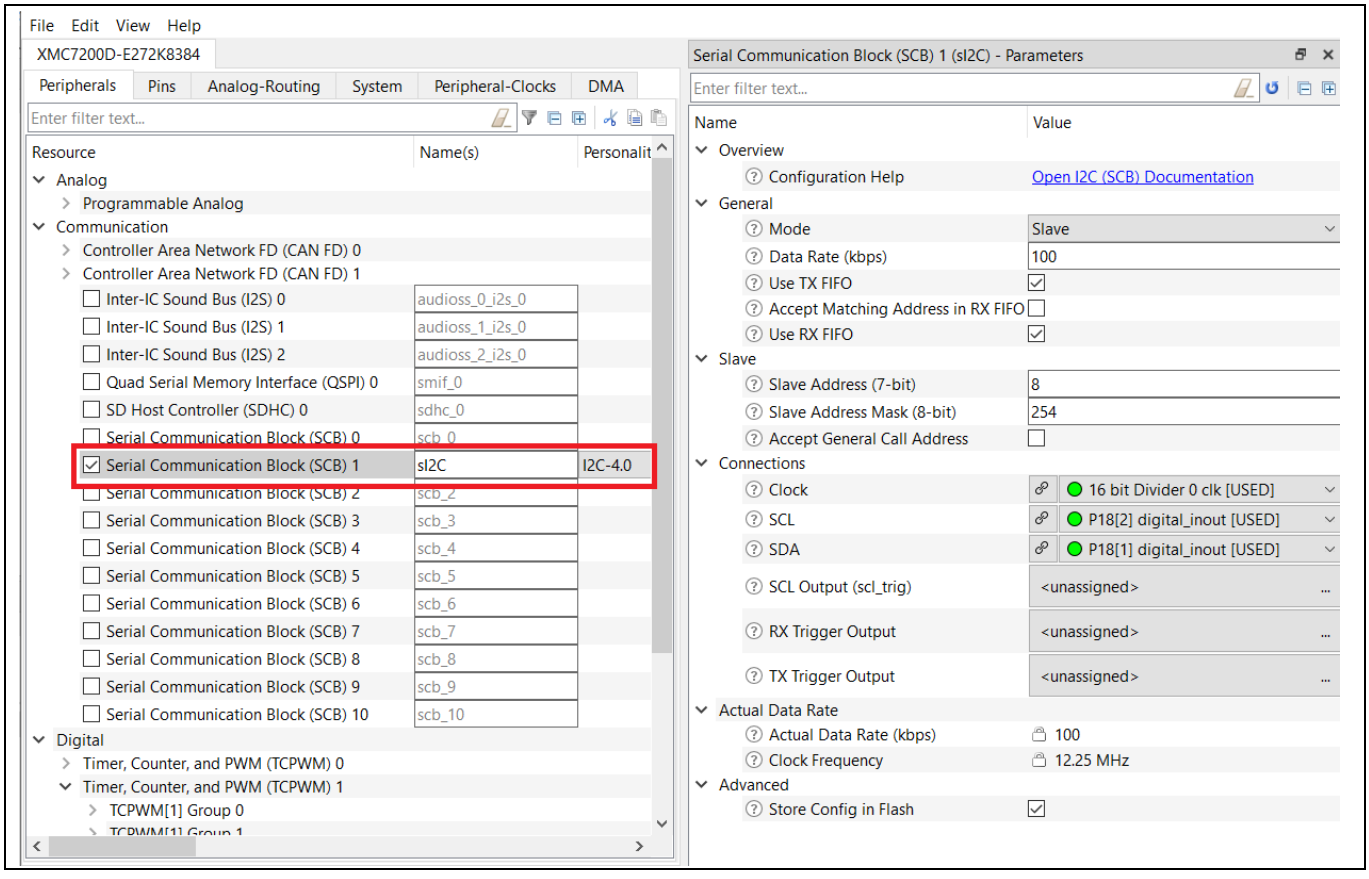


**Figure 28 SCB configuration in ModusToolbox™**

## I2C setting procedure example

**Code Listing 5 Example of basic I2C Slave configuration using PDL**

```
cy_stc_sysint_t mI2C_SCB_IRQ_cfg =
{
    .intrSrc = ((NvicMux3_IRQn << 16) | mI2C_IRQ), /* Bit 0-15 of intrSrc is used
to store system interrupt value and bit 16-31 to store CPU IRQ value */
    .intrPriority = 2UL,
};

/* Structure for master transfer configuration */

cy_stc_scb_i2c_master_xfer_config_t masterTransferCfg =
{
    .slaveAddress = I2C_SLAVE_ADDR,
    .buffer       = NULL,
    .bufferSize   = 0U,
    .xferPending  = false
};

cy_stc_scb_i2c_context_t mI2C_context;

void mI2C_Interrupt(void)
{
    Cy_SCB_I2C_Interrupt(mI2C_HW, &mI2C_context);
}

uint8_t WritePacket(uint8_t* buffer, uint32_t bufferSize)
{
    uint8_t status = TRANSFER_ERROR;
    cy_en_scb_i2c_status_t  errorStatus;
    uint32_t masterStatus;
    /* Timeout 1 second (one unit is us) */
    int32_t timeout =  1000000UL;

    /* Setup transfer specific parameters. */
    masterTransferCfg.buffer     = buffer;
    masterTransferCfg.bufferSize = bufferSize;
      masterTransferCfg.slaveAddress = I2C_SLAVE_ADDR;

    /* Initiate write transaction. */
    errorStatus = Cy_SCB_I2C_MasterWrite(mI2C_HW, &masterTransferCfg,
&mI2C_context);
    if(errorStatus == CY_SCB_I2C_SUCCESS)
    {
        /* Wait until master complete read transfer or time out has occurred */
        do
        {
            masterStatus  = Cy_SCB_I2C_MasterGetStatus(mI2C_HW, &mI2C_context);
            Cy_SysLib_DelayUs(CY_SCB_WAIT_1_UNIT);
            timeout--;
        } while ((0UL != (masterStatus & CY_SCB_I2C_MASTER_BUSY)) && (timeout >
0));

        if (timeout <= 0)
        {
            /* Timeout recovery */
            Cy_SCB_I2C_Disable(mI2C_HW, &mI2C_context);
            Cy_SCB_I2C_Enable (mI2C_HW);
        }
        else
```

## I2C setting procedure example

**Code Listing 5 Example of basic I2C Slave configuration using PDL**

```
        {
            if ((0u == (MASTER_ERROR_MASK & masterStatus)) &&
                (TX_PACKET_SIZE == Cy_SCB_I2C_MasterGetTransferCount(mI2C_HW,
&mI2C_context)))
            {
                status = TRANSFER_CMPLT;
            }
        }
    }

    return (status);
}

uint8_t ReadStatusPacket(void)
{
    uint8_t status = TRANSFER_ERROR;
    cy_en_scb_i2c_status_t errorStatus;
    uint32_t masterStatus;
    /* Timeout 1 second (one unit is us) */
    uint32_t timeout = 1000000UL;
    uint8_t  rxBuffer[RX_PACKET_SIZE];

    /* Setup transfer specific parameters */
    masterTransferCfg.buffer     = rxBuffer;
    masterTransferCfg.bufferSize = RX_PACKET_SIZE;
      masterTransferCfg.slaveAddress = I2C_SLAVE_ADDR;

    /* Initiate read transaction */
    errorStatus = Cy_SCB_I2C_MasterRead(mI2C_HW, &masterTransferCfg,
&mI2C_context);
    if(errorStatus == CY_SCB_I2C_SUCCESS)
    {
        /* Wait until master complete read transfer or time out has occurred */
        do
        {
            masterStatus  = Cy_SCB_I2C_MasterGetStatus(mI2C_HW, &mI2C_context);
            Cy_SysLib_DelayUs(CY_SCB_WAIT_1_UNIT);
            timeout--;

        } while ((0UL != (masterStatus & CY_SCB_I2C_MASTER_BUSY)) && (timeout >
0));
       if (timeout <= 0)
        {
            /* Timeout recovery */
            Cy_SCB_I2C_Disable(mI2C_HW, &mI2C_context);
            Cy_SCB_I2C_Enable (mI2C_HW);
        }
        else
        {
            /* Check transfer status */
            if (0u == (MASTER_ERROR_MASK & masterStatus))
            {
                /* Check packet structure and status */
                if((PACKET_SOP   == rxBuffer[RX_PACKET_SOP_POS]) &&
                    (PACKET_EOP   == rxBuffer[RX_PACKET_EOP_POS]) &&
                    (STS_CMD_DONE == (rxBuffer[RX_PACKET_STS_POS])))
                {
                    status = TRANSFER_CMPLT;
```

## I2C setting procedure example

**Code Listing 5 Example of basic I2C Slave configuration using PDL**

```
                }
            }
        }
    }
    return (status);
}

int main(void)
{
    cy_rslt_t result;
    uint8_t cmd = ON;
    uint8_t  buffer[TX_PACKET_SIZE];
      cy_en_scb_i2c_status_t initStatus;
      cy_en_sysint_status_t sysStatus;

    /* Initialize the device and board peripherals */
    result = cybsp_init();
    /* Board init failed. Stop program execution */
    handle_error(result);

    /* Initialize the retarget-io */
    result = cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,
                                 CY_RETARGET_IO_BAUDRATE);
    /* Retarget-io init failed. Stop program execution */
    handle_error(result);

    /* \x1b[2J\x1b[;H - ANSI ESC sequence for clear screen */
    printf("\x1b[2J\x1b[;H");

    printf("****************** "
           "PDL: I2C Master "
           "****************** \r\n\n");

    /* I2C Master configuration settings */
    printf(">> Configuring I2C Master..... ");

    /* Init I2C master */
    initStatus = Cy_SCB_I2C_Init(mI2C_HW, &mI2C_config, &mI2C_context);
    if(initStatus != CY_SCB_I2C_SUCCESS)
    {
      return I2C_FAILURE;
    }

    /* Hook interrupt service routine. */
    sysStatus = Cy_SysInt_Init(&mI2C_SCB_IRQ_cfg, &mI2C_Interrupt);
    if(sysStatus != CY_SYSINT_SUCCESS)
    {
      return I2C_FAILURE;
    }

    /* Enable interrupt in NVIC. */
    NVIC_EnableIRQ((IRQn_Type) NvicMux3_IRQn);
      /* Enable I2C master hardware. */
      Cy_SCB_I2C_Enable(mI2C_HW);

    printf("Done\r\n\n");

    /* Enable interrupts */
```

## I2C setting procedure example

**Code Listing 5 Example of basic I2C Slave configuration using PDL**

```
    __enable_irq();

    for (;;)
    {
        /* create packet to be sent to slave.  */
        buffer[PACKET_SOP_POS] = PACKET_SOP;
        buffer[PACKET_EOP_POS] = PACKET_EOP;
        buffer[PACKET_CMD_POS] = cmd;

        uint8_t res = WritePacket(buffer, TX_PACKET_SIZE);

        /* Send packet with command to the slave. */
            if (TRANSFER_CMPLT == res)
            {
                  printf("Write complete\n");
                  /* Read response packet from the slave. */
                  if (TRANSFER_CMPLT == ReadStatusPacket())
                  {
                        printf("Read Complete\n");
                        /* Next command to be written. */
                        cmd = (cmd == ON) ? OFF : ON;
                  }
                  /* Give 1 Second delay between commands. */
                  Cy_SysLib_Delay(CMD_TO_CMD_DELAY);
            }

    }
}
```

## References

[1] Datasheet

- XMC7100 XMC7000 microcontroller 32-bit Arm® Cortex®-M7
- XMC7200 XMC7000 microcontroller 32-bit Arm® Cortex®-M7

[2] Architecture reference manual

- XMC7000 MCU family architecture reference manual

[3] Registers reference manual

*Note:* *Contact Technical support to obtain these documents.*

- 002-33817: XMC7100 registers reference manual
- 002-33812: XMC7200 registers reference manual

## Glossary

**CMD_RESP mode**
CMD_RESP (Command Response) mode is similar to EZ mode. The major difference is whether a CPU sets the Slave's base address or a Master device sets it.

**DMA**
direct memory access

**EZ mode**
EZ (easy) mode is the Infineon original communication protocol which is prepared to simplify the Write/Read access between the device in SPI and I2C. During DeepSleep mode, it can communicate with the Master device without CPU intervention.

**FIFO**
First in First Out

**I2C**
inter-integrated circuit
I2C bus is a serial synchronous communication bus corresponding to the multi-Master and the multi-Slave. It is used for low-speed communication between MCUs and peripheral devices. I2C bus is used with two lines of clock (SCK) and data (SDA), and usually pulled-up by resistance.

**IrDA**
IrDA is a standard for optical radio data communication by infrared rays.

**Smart Card**
Smart Card is a card which integrated a circuit to record data and to operate it.

**SPI**
Serial Peripheral Interface
SPI is a synchronous serial communication interface specification used for short distance communication with peripheral devices.

**UART**
Universal Asynchronous Receiver-Transmitter
UART is a receiver-transmitter circuit to convert a serial signal into a parallel signal, and to convert the opposite direction. It is used for low-speed communication between MCU and an external equipment.

## Revision history

| Document revision | Date | Description of changes |
|---|---|---|
| ** | 2021-11-22 | New application note. |
| *A | 2023-08-17 | Added 4.1.4, 4.2.3, 5.1.4, 6.1.4, and 6.2.3 use case sections<br>Updated References |

**Trademarks**
All referenced product or service names and trademarks are the property of their respective owners.

**Edition 2023-08-17**

**Published by**

**Infineon Technologies AG**

**81726 Munich, Germany**

**© 2023 Infineon Technologies AG. All Rights Reserved.**

**Do you have a question about this document?**

**Email:** erratum@infineon.com

**Document reference**

**002-34127 Rev. *A**

**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.

**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.