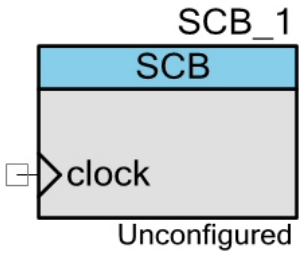
PSoC 4 Serial Communication Block (SCB)

2.0



Features

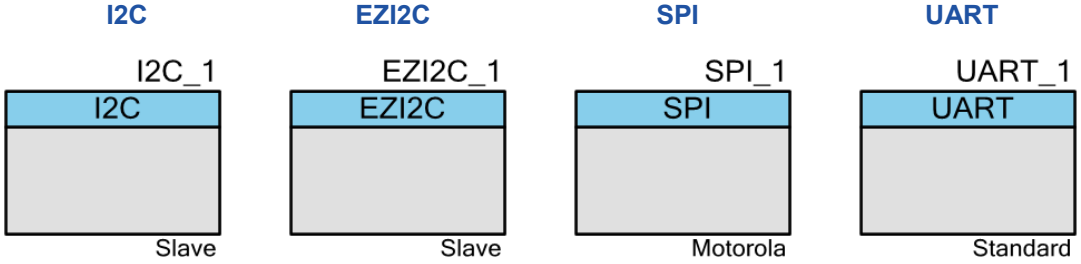
- Industry-standard NXP® I²C bus interface
- Standard SPI Master and Slave interfaces with Motorola, Texas Instruments, and the National Semiconductor's Microwire protocols
- Standard UART TX and RX interfaces with SmartCard reader and IrDA protocols
- EZ I²C mode which emulates a common I²C EEPROM interface
- Supports wakeup from Deep Sleep mode
- Run-time reconfigurable
- I²C Bootloader support

General Description

The PSoC 4 SCB component is a multifunction hardware block that implements the following communication components: I2C, SPI, UART, and EZI2C. Each is available as a pre-configured schematic macro in the PSoC Creator Component Catalog, labeled with “SCB Mode.”

Note PSoC 4000 devices support only I²C modes. The UART or SPI mode choice is not available.

Click on one of the links below to jump to the appropriate section:



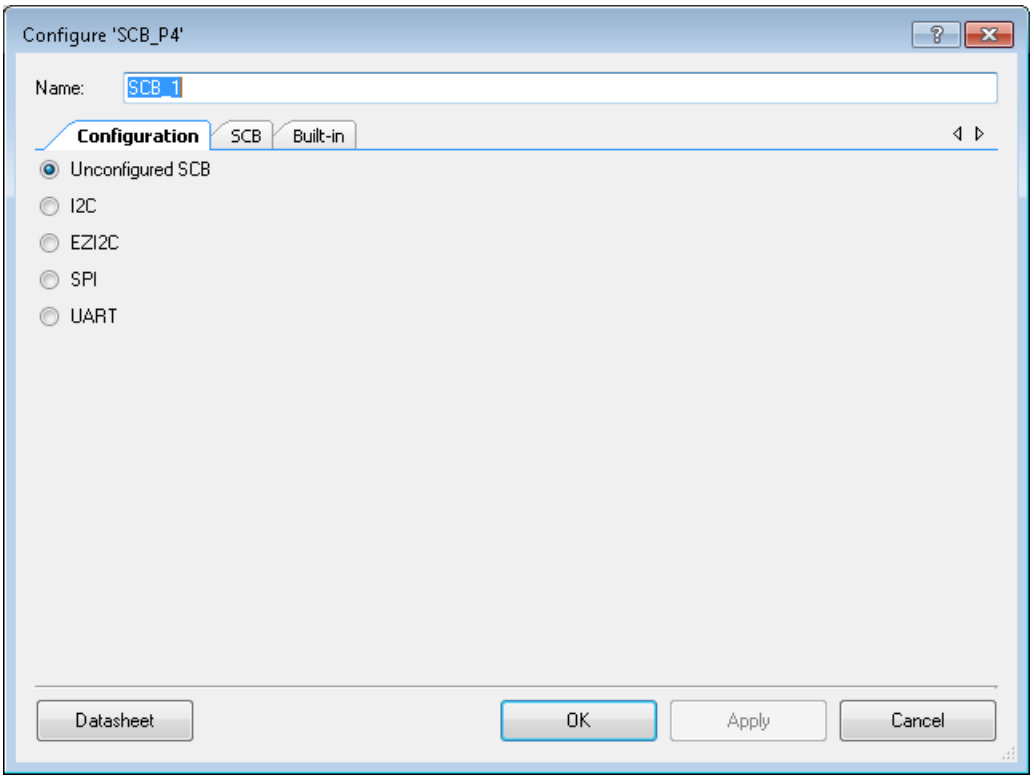
There is also an Unconfigured SCB component entry in the Component Catalog.

When to Use an SCB Component

The SCB can be used in a pre-configured mode: I2C, EZI2C, SPI, and UART. Alternatively, the SCB can be left unconfigured at build time and then configured during run-time into any of the

modes by calling the appropriate API functions. All configuration settings made at build time can also be made during run time.

The following figure shows the component Configure dialog used to select the desired mode.

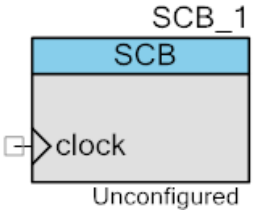


Note PSoC 4000 devices support only I^2C modes. The UART or SPI mode choice is not available.

The pre-configured modes are the typical use case. They are the simplest method to configure the SCB into the mode of operation that is desired. The unconfigured method can be used to create designs for multiple applications and where the specific usage of the SCB in the design is not known when the PSoC Creator hardware design is created.

Unconfigured SCB

The SCB can be run-time configured for operation in any of the modes(I2C, SPI, UART, EZI2C) from the Unconfigured mode. It can also be re-configured from any of these modes to any of the other modes during run time. For example, you can reconfigured the SCB from SPI to UART during run time.



Input/Output Connections

This section describes the various input and output connections for the SCB component. An asterisk (*) in the list of terminals indicates that the terminal may be hidden on the symbol under the conditions listed in the description of that terminal.

clock – Input

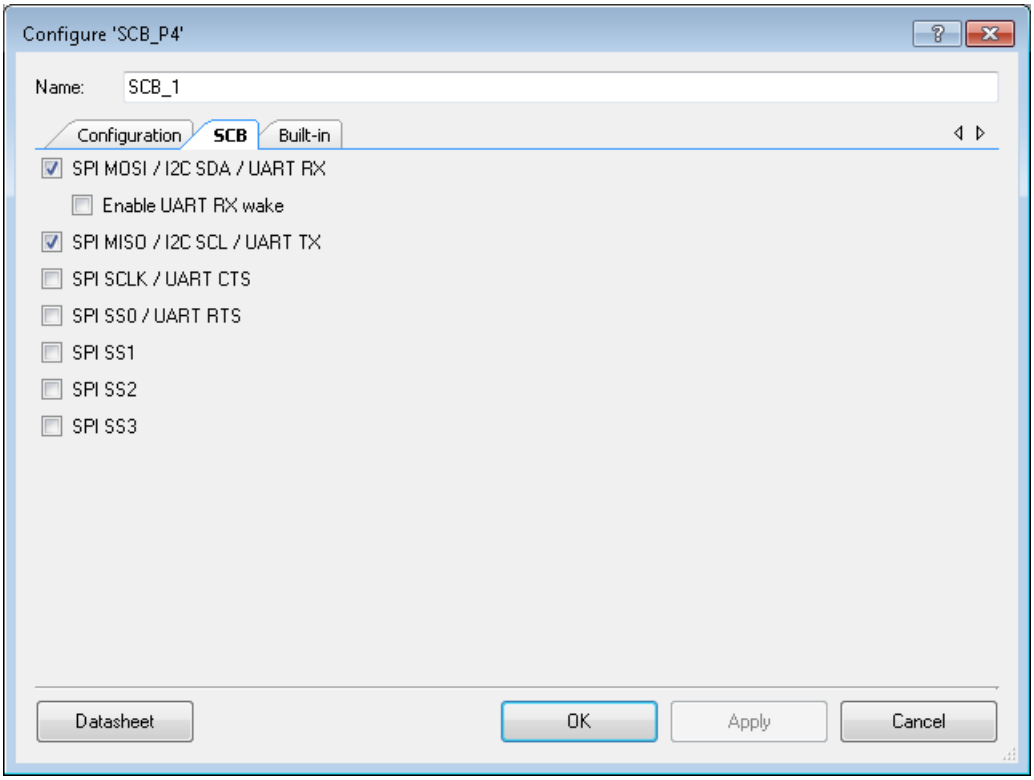
Clock that operates this block. This terminal is required in Unconfigured mode. For other modes the option is provided to use an internal clock or an external clock connected to this terminal.

The interface-specific pins are buried inside the component because these pins use dedicated connections and are not routable as general purpose signals. See the *I/O System* section in the device *Technical Reference Manual (TRM)* for more information.

Note The input buffer of buried output pins is disabled so as not to cause current linkage in low power mode. Reading the status of these pins always returns zero. To get the current status, the input buffer must be enabled before status read.

SCB Tab

Use the **SCB** tab to select the pins that will be used by the SCB component in Unconfigured mode. The communication type along with the pin name is listed with a check box to enable pin. When the pin is enabled it is reserved for the SCB, and cannot be used by other functions. To see what pins were reserved consult the pins tab in the .cydwr file



The **Enable UART RX wake** adds an interrupt to the RX pin to accomplish the UART wake-up capability. This option restricts the processing of any other pin interrupts from the port where this RX pin is placed.

Note PSoC 4000 devices only support I^2C modes and have different locations for the SCL and SDA pins. The following pin names are used for this device:

- SPI MOSI / I2C SDA / UART RX pin name is I2C SCL
- SPI MISO / I2C SCL / UART TX pin name is I2C SDA
- SPI SCLK, SPI SS0 – SS3 pins are not available, nor is the Enable UART RX wake option

Note PSoC 4100/PSoC 4200 devices do not support UART hardware flow control and have different locations for the SCL and SDA pins. The following pin name changes are used for these devices:

- SPI MOSI / I2C SDA / UART RX pin name is SPI MOSI / I2C SCL / UART RX
- SPI MISO / I2C SCL / UART TX pin name is SPI MISO / I2C SDA / UART TX
- SPI SCLK / UART CTS pin name is SPI SCLK
- SPI SS0 / UART RTS pin name is SPI SS0

Unconfigured mode operation

Before starting operation in Unconfigured mode, determine which communication interfaces will be used (more than one communication interface can be used in one SCB). Next, use the **SCB** parameters tab to select pins required to implement the chosen communication interfaces.

The communication interface name is listed first, followed by the pin name related to the specific interface. For example, SPI MOSI / I2C SCL / UART RX means this pin functions as MOSI when the SCB is configured to utilize the SPI interface; it functions as SCL for the I^2C interface; and as RX for the UART interface.

Next, a clock component must be connected to the SCB clock input. This clock frequency along with the SCB oversampling configuration (set in the interface-specific Init API function) determines the operation speed of the communications interface. The clock frequency will be configured later in the firmware by setting a clock divider, using the clock's API. The possible choice of clock configuration is: source HFCLK with divider 1.

To use the UART and I^2C interfaces, select the following pins on the **SCB** tab:

- SPI MOSI / I2C SCL / UART RX pin
- SPI MISO / I2C SDA / UART TX pin

To use the SPI interface, select the following pins on the **SCB** tab:

- SPI MOSI / I2C SCL / UART RX pin
- SPI MISO / I2C SDA / UART TX pin
- SPI SCLK
- Any combination of SPI SS0 – SPI SS3

Interface data rate configuration

The data rate is a function of the clock source frequency and the component configuration of the oversampling parameter. The oversampling parameter is only important for master modes and has no effect for slave (UART mode has the same dependency on oversampling as master modes). For slave modes only, the connected clock source frequency is important.

For I^2C master modes, SPI master, and UART, the data rate is calculated using the formula below (where f_{SCBCLK} is the frequency of the clock component connected to the SCB):

$$\text{Data rate} = (f_{SCBCLK} / \text{Oversampling value})$$

$$f_{SCBCLK} = \text{Data rate} * \text{Oversampling value}$$

Note For the I^2C interface in master modes, the oversampling value is a sum of Low and High oversampling factor values.

For I^2C and EZI^2C in slave modes, the f_{SCBCLK} must match the values provided in Table 1 on page 14.

For SPI slave mode, the clock source frequency impacts T_{DSO} parameter. Refer to SPI AC Specifications for the selected device and to the Slave data rate section.

Refer to the applicable I2C / EZI2C / SPI / UART Parameter section for more information about data rate and oversampling. To change f_{SCBCLK} clock frequency, the clock divider must be changed. The clock component provides API to perform this task.

$$Div_{SCBCLK} = f_{HFCLK} / f_{SCBCLK}$$

Note The Div_{SCBCLK} must be an integer value.

Example of Div_{SCBCLK} calculation for I^2C and UART is as follows:

Design HFCLK configuration of f_{HFCLK} = 24 MHz

Required I2C slave data rate = 100 kbps and UART baud rate = 115200 bps

The f_{SCBCLK} = 1.6MHz is taken from Table 1 on page 14 for data rate 100 kbps:

$$Div_{SCBCLK} = f_{HFCLK} / f_{SCBCLK} = 24 \text{ MHz} / 1.6 \text{ MHz} = 15$$

The oversampling default value 16 is chosen to calculate Div_{SCBCLK} for the UART.

$$f_{SCBCLK} = \text{Data rate} * \text{Oversampling value} = 115200 * 16 = \sim 1{,}843 \text{ MHz}$$

$$Div_{SCBCLK} = f_{HFCLK} / f_{SCBCLK} = 24 \text{ MHz} / 1,\ 843 \text{ MHz} = \sim 13$$

For the UART, the f_{SCBCLK} accuracy is important for correct operation and the actual f_{SCBCLK} must be calculated to use f_{HFCLK} and Div_{SCBCLK}.

$$\text{Actual } f_{SCBCLK} = f_{HFCLK} / Div_{SCBCLK} = 24 \text{ MHz} / 13 = \sim 1{,}846 \text{ MHz}$$

The deviation of actual f_{SCBCLK} from desired must be calculated:

(1,843MHz – 1,846 MHz) / 1,843 MHz = ~0.2%

Taking into account HFCLK accuracy ±2%, the total error is: 0.2 + 2= 2.2%. The total error value is less than 5% and it is enough for correct UART operation.

The following numbers are calculated:

- I^2C master data rate = 100 kbps: Div_{SCBCLK} = 15
- UART baud rate = 115200 bps: Oversampling = 16, Div_{SCBCLK} = 13

Run-time Configuration

Configuration structures are provided for each interface. These structures provide configuration fields that match the selections available in the Configure dialog for the specific interface. The description of each structure is provided in the APIs section of corresponding interface:

- void SCB_**I2CInit**(SCB_I2C_INIT_STRUCT *config)
- void SCB_**EzI2CInit**(SCB_EZI2C_INIT_STRUCT *config)
- void SCB_**SpiInit**(SCB_SPI_INIT_STRUCT *config)
- void SCB_**UartInit**(SCB_UART_INIT_STRUCT *config)

Allocate structures for the selected interfaces and fill the structure with the required configuration information. A pointer to this structure is passed to the appropriate initialization function of the selected interface. The list of initialization functions for each interface is provided above.

The following example provides configuration structures for:

- I2C slave, data rate 100 kbps, slave address is 0x08
- UART RX+TX, sub-mode Standard, buffer size 10 for TX and RX (implies software buffer utilization)

```
#define SCB_BUFFER_SIZE     (10u)

/* Common buffers  or I2C and UART */
uint8 bufferRx[SCB_BUFFER_SIZE + 1u];/* RX software buffer requires one extra
entry for correct operation in UART mode */
uint8 bufferTx[SCB_BUFFER_SIZE];     /* TX software buffer */

const SCB_I2C_INIT_STRUCT configI2C =
{
    SCB_I2C_MODE_SLAVE, /* mode: slave */
    0u,    /* oversampleLow: N/A for slave, SCBCLK determines maximum data rate*/
    0u,    /* oversampleHigh: N/A for slave, SCBCLK determines maximum data rate*/
    0u,    /* enableMedianFilter: N/A since SCB v2.0 */
    0x08u, /* slaveAddr: slave address */
```

```
    0xFEu, /* slaveAddrMask: signle slave address */
    0u,    /* acceptAddr: disable */
    0u,    /* enableWake: disable */
    0u,    /* enableByteMode: disable */
    100u   /* dataRate: 100 kbps */
};

const SCB_UART_INIT_STRUCT configUart =
{
    SCB_UART_MODE_STD, /* mode: Standard */
    SCB_UART_TX_RX,    /* direction: RX + TX */
    8u,     /* dataBits: 8 bits */
    SCB_UART_PARITY_NONE, /* parity: None */
    SCB_UART_STOP_BITS_1, /* stopBits: 1 bit */
    16u,    /* oversample: 16u */
    0u,     /* enableIrdaLowPower: disable */
    1u,     /* enableMedianFilter: enable */
    0u,     /* enableRetryNack: disable */
    0u,     /* enableInvertedRx: disable */
    0u,     /* dropOnParityErr: disable */
    0u,     /* dropOnFrameErr: disable */
    0u,     /* enableWake: disable */
    SCB_BUFFER_SIZE, /* rxBufferSize: software buffer 10 bytes */
    bufferRx, /* rxBuffer: RX software buffer enable */
    SCB_BUFFER_SIZE, /* txBufferSize: software buffer 10 bytes */
    bufferTx, /* txBuffer: TX software buffer enable */
    0u,     /* enableMultiproc: disable */
    0u,     /* multiprocAcceptAddr: disable */
    0u,     /* multiprocAddr: N/A for this configuration */
    0u,     /* multiprocAddrMask: N/A for this configuration */
    1u,     /* enableInterrupt: enable to process software buffer */
    SCB_INTR_RX_NOT_EMPTY,   /* rxInterruptMask: enable NOT_EMPTY for RX software
buffer operations */
    0u, /* rxTriggerLevel: N/A for this configuration */
    0u, /* txInterruptMask: NOT_FULL is enabled when there is data to transmit */
    0u, /* txTriggerLevel: N/A for this configuration */
    0u, /* enableByteMode: N/A for this configuration */
    0u, /* enableCts: N/A for this configuration */
    0u, /* ctsPolarity: N/A for this configuration */
    0u, /* rtsRxTriggerLevel: N/A for this configuration */
    0u  /* rtsPolarity: N/A for this configuration */
};
```

The following example implements a function that changes the SCB configuration according to the passed opMode, and returns the status of the configuration change. This function refers to configuration structures provided for the I^2C and UART previously.

The instance name of the SCB component is “SCB” and the instance name of the clock component is “SCBCLK”.

Note This is a custom function that is not part of the component API.

```
/* The clock divider value written into the register has to be one less from
calculated */
#define SCBCLK_I2C_DIVIDER  (14u)   /* I2C Slave: 100 kbps Required SCBCLK = 1.6
MHz, Div = 15 */
#define SCBCLK_UART_DIVIDER (12u)   /* UART: 115200 kbps with OVS = 16. Required
SCBCLK = 1.846 MHz, Div = 13 */
/* Operation mode: I2C slave or UART */
#define OP_MODE_UART    (1u)
#define OP_MODE_I2C     (2u)

cystatus SetScbConfiguration(uint32 opMode)
{
    cystatus status = CYRET_SUCCESS;

    if (OP_MODE_I2C == opMode)
    {
        SCB_Stop(); /* Disable component before configuration change */

        /* Change clock divider */
        SCBCLK_Stop();
        SCBCLK_SetFractionalDividerRegister(SCBCLK_I2C_DIVIDER, 0u);
        SCBCLK_Start();

        /* Configure to I2C slave operation */
        SCB_I2CSlaveInitReadBuf (bufferTx, SCB_BUFFER_SIZE);
        SCB_I2CSlaveInitWriteBuf(bufferRx, SCB_BUFFER_SIZE);
        SCB_I2CInit(&configI2C);
        SCB_Start(); /* Enable component after configuration change */
    }
    else if (OP_MODE_UART == opMode)
    {
        SCB_Stop(); /* Disable component before configuration change */
        /* Change clock divider */
        SCBCLK_Stop();
        SCBCLK_SetFractionalDividerRegister(SCBCLK_UART_DIVIDER, 0u);
        SCBCLK_Start();
        /* Configure to UART operation */
        SCB_UartInit(&configUart);
        SCB_Start(); /* Enable component after configuration change */
    }
    else
    {
        status = CYRET_BAD_PARAM; /* Uknowns operation mode - no actions */
    }
    return (status);
}
```

Note Before changing the configuration, the SCB component must be disabled.

Note The SCB_Init() function does not initialize the component when the mode is Unconfigured. The SCB_**"Mode"**Init() specific APIs have to called.

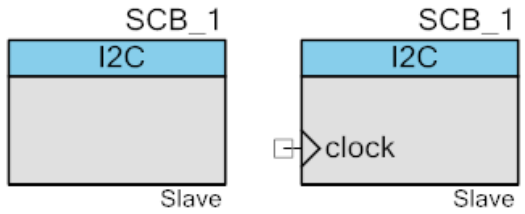
API Names

Some APIs contain specific interface prefixes as part of their name. These APIs operate correctly only when the component is configured to utilize this interface. For example, the SCB_I2CSlaveStatus() function works with the I^2C interface.

Other APIs are shared between two interfaces. In these cases, the API name contains each interface. For example, the SCB_SpiUartWriteTxData() works with both the SPI or UART interfaces.

APIs that do not belong to specific interfaces do not contain interface prefixes. For example, SCB_Enable() or SCB_EnableInt().

I2C



The I^2C bus is an industry-standard, two-wire hardware interface developed by Philips. The master initiates all communication on the I^2C bus and supplies the clock for all slave devices. I^2C is an ideal solution when networking multiple devices on a single board or small system.

The component supports I^2C Slave, Master, Multi-Master and Multi-Master-Slave configurations.

The component supports standard clock speeds up to 1000 kbps. It is compatible [1] with I^2C Standard-mode, Fast-mode, and Fast-mode Plus devices as defined in the NXP I^2C-bus specification [2] on the NXP web site at www.nxp.com. The component is compatible with other third-party slave and master devices.

Input/Output Connections

This section describes the various input and output connections for the SCB component. An asterisk (*) in the list of terminals indicates that the terminal may be hidden on the symbol under the conditions listed in the description of that terminal.

clock – Input*

Clock that operates this block. The presence of this terminal varies depending on the **Clock from terminal** parameter.

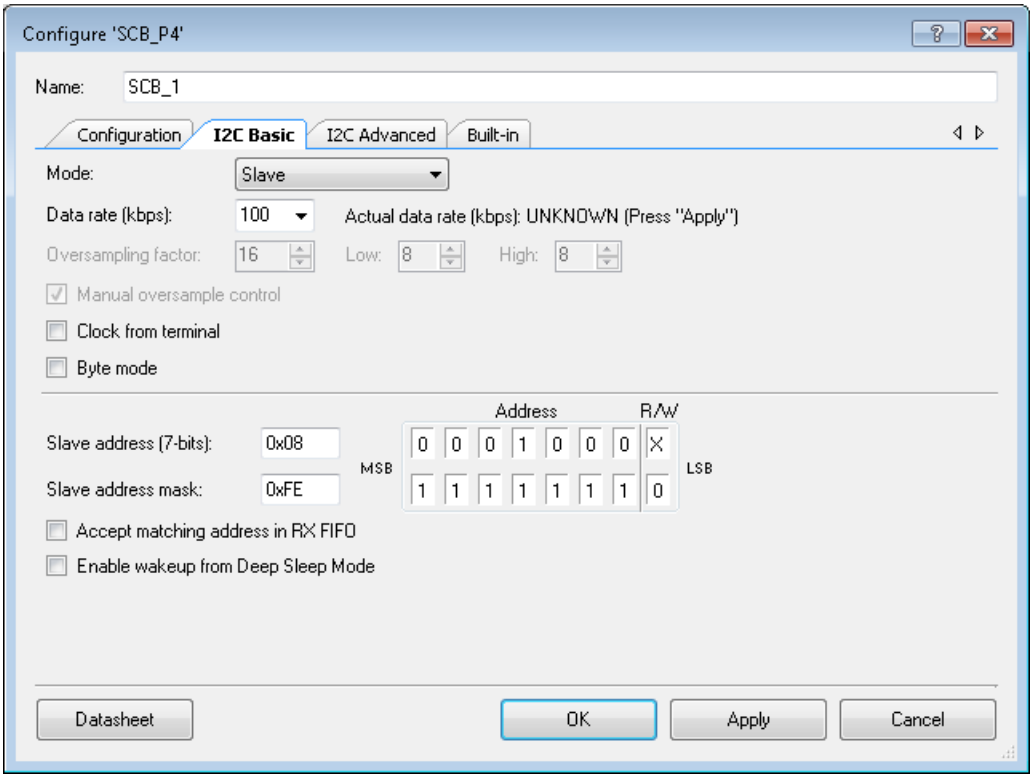
The interface-specific pins are buried inside the component because these pins use dedicated connections and are not routable as general purpose signals. See the *I/O System* section in the device *Technical Reference Manual (TRM)* for more information.

---

[1] The PSoC 4 I^2C peripherals are not completely compliant with the I^2C specification except PSoC 4100 BLE/PSoC 4200 BLE. For detailed information refer to the *Device datasheet*.

[2] Refer to the *I^2C-Bus Specification* (Rev. 5 from October 2012) on the NXP web site at www.nxp.com.

Basic I2C Parameters



The **I2C Basic** tab has the following parameters:

Mode

This option determines which mode will be supported: Slave, Master, Multi-Master or Multi-Master-Slave.

- **Slave** – Slave only operation (default)
- **Master** – Master only operation
- **Multi-Master** – Supports more than one master on the bus
- **Multi-Master-Slave** – Simultaneous slave and multi-master operation

Data rate

This parameter is used to set the I^2C data rate value up to 1000 kbps (400 kbps for PSoC 4000 family); the actual data rate may differ from the selected data rate due to available clock frequency and component settings. The standard data rates are 100 (default), 400, and 1000 kbps. This parameter has no effect if the **Clock from terminal** option is enabled.

Refer to the Data rate configuration section for more information about provided options of the data rate selection.

Actual data rate

The actual data rate displays the data rate at which the component will operate with current settings. The factors that affect the actual data rate calculation are: the accuracy of the component clock (internal or external) and oversampling factor (only for the Master modes). If any of these parameters change the actual data rate is unknown. To calculate the new actual data rate press the Apply button.

Note For Slave mode the actual data rate always provides maximum value for the selected data rate mode (Standard-mode, Fast-mode, Fast-mode Plus).

Oversampling factor

This parameter defines the oversampling factor of the I^2C SCL clock; the number of component clocks within one I^2C SCL clock period.

For Slave mode, the oversampling factor is not applicable. The component configures the SCB clock to be within the valid range for the selected data rate. The valid clock frequency range is taken from the Table 1 on page 14.

For Master modes, the Oversampling factor is the sum of Low and High oversampling values. These values are used to generate the Low and High phases of the I^2C clock.

The valid range of the Oversampling factor values is 12–32. The default is 16.

Low

This parameter is only applicable for **Master** modes. It specifies the oversampling factor of the I^2C SCL clock low phase; the number of component clocks within one low period of the I^2C SCL clock. The minimum oversampling factor value is 7. The default is 8.

High

This parameter is only applicable for **Master** modes. It specifies the oversampling factor of the I^2C SCL clock high phase; the number of component clocks within one high period of the I^2C SCL clock. The minimum oversampling factor value is 5. The default is 8.

Manual oversample control

This option is only available for **Master** modes. It allows a choice between manual and automatic selection of I^2C Oversampling factor parameters.

In the case of manual oversample control all oversampling parameters are editable. The oversampling factor is used to calculate internal clock frequency to achieve this amount of oversampling for the defined Data rate: f_{SCBCLK} = Data rate * Oversampling factor. The oversampling Low and High values are editable but sum of them must be equal to Oversampling factor.

In the case of automatic oversample control all oversampling parameters are disabled and calculated by the GUI. The GUI requests an internal clock that is in range provided in the Table 2

on page 14 for the selected data rate. Creator creates a clock in this range and returns the clock frequency to the GUI. The GUI uses this clock frequency and the selected data rate to calculate the oversampling value. The oversampling values Low and High are adjusted to meet the request data rate.

Note Refer to the Clock from terminal section to get more information about possible deviation of requested clock frequency and actual value.

Clock from terminal

This check box allows choosing an internally configured clock (by the component) or an externally configured clock (by the user) for component operation. Refer to the Oversampling factor section to understand relationship between component clock frequency and the component parameters.

When this option is enabled, the component does not control the data rate, but displays the actual data rate based on the user-connected clock source frequency and the component oversampling factor (only for the Master modes). When this option is not enabled, the clock configuration is provided by the component. The clock source frequency is calculated or selected by the component based on the Data rate parameter and Oversampling factor (only for the Master mode).

The following tables show the valid ranges for the component clock for each data rate. When using the Clock from terminal option, ensure that the external clock is within these ranges.

Table 1. I2C Slave clock frequency ranges

Parameter	Standard-mode (0-100 kbps)		Fast-mode (0-400 kbps)		Fast-mode Plus (0-1000 kbps)		Units
	Min	Max	Min	Max	Min	Max	
f_{SCB}	1.55	12.8	7.82	15.38	15.84	48.0	MHz

Note For Slave mode, if the clock frequency is less than lower limit of 1.55 MHz, an error is generated while building the project.

Table 2. I2C Master modes clock frequency ranges

Parameter	Standard-mode (0-100 kbps)		Fast-mode (0-400 kbps)		Fast-mode Plus (0-1000 kbps)		Units
	Min	Max	Min	Max	Min	Max	
f_{SCB}	1.55	3.2	7.82	10.00	14.32	25.8	MHz
Oversampling Low	8	16	13	16	9	16	–
Oversampling High	8	16	8	16	6	16	–

Taking into account the ranges provided in Table 2 on page 14 for clock frequency and oversampling the calculated data rate ranges are provided in Table 3 on page 15.

It is possible to create data rates that are outside of the ranges provided in Table 3 on page 15. However creating these data rates requires using clock frequencies or oversampling outside of the ranges provided in Table 2 on page 14. If a clock frequency or oversampling is used outside of the range provided in Table 2 on page 14 certain I^2C parameters in the I^2C specification may be violated. To determine which specifications are violated refer to the I2C spec parameters calculation section.

Table 3. I2C master modes data rates ranges

Parameter	Standard-mode (0-100 kbps)		Fast-mode (0-400 kbps)		Fast-mode Plus (0-1000 kbps)		Units
	Min	Max	Min	Max	Min	Max	
Data rate	48	100	244	400	447	1000	kbps

Note PSoC Creator is responsible for providing requested clock frequency (internal or external clock) based on current design clock configuration. When the requested clock frequency with requested tolerance cannot be created, a warning about the clock accuracy range is generated while building the project. This warning contains the actual clock frequency value created by PSoC Creator. To remove this warning you must either change the system clock, component settings or external clock to fit the clocking system requirements.

Byte mode

This option is only applicable for PSoC 4100 BLE/PSoC 4200 BLE devices. It allows doubling the TX and RX FIFO depth from 8 to 16 bytes. Increasing FIFO depth improves performance of I^2C operation, as more bytes can be transmitted or received without software interaction.

Slave address (7-bits)

This is the I^2C address that will be recognized by the slave. It is only applicable for slave modes. This address is the 7-bit right-justified slave address and does not include the R/W bit. A slave address between 0x08 and 0x7F may be selected; the default is 0x08.

The value may be entered as decimal or hexadecimal; for hexadecimal numbers type ‘0x’ before the address. The binary input format is supported as well.

Slave address mask

This parameter is used to mask bits of the slave address during the address match procedure. Bit 0 of the address mask corresponds to the read/write direction bit and is always a do not care in the address match.

- Bit value 0 – excludes bit from address comparison.

- Bit value 1 – the bit needs to match with the corresponding bit of the I²C slave address.

The following example shows the 7-bit slave address is 0x1B and the 8-bit address Mask is 0xDE.

	7-bits Slave address							R/W
Address	0	0	1	1	0	1	1	x
Mask	1	1	0	1	1	1	1	0
Result	0	0	x	1	0	1	1	x

x = Do not care

Thus the matched 7-bit slave addresses are 0x1B and 0x0B.

The value may be entered as decimal or hexadecimal; for hexadecimal numbers type '0x' before the address. The binary input format is supported as well.

Accept matching address in RX FIFO

This parameter determines whether to accept a matched I²C slave address in the RX FIFO or not. Enable this option if more than one I²C address support desired.

Enable wakeup from Deep Sleep Mode

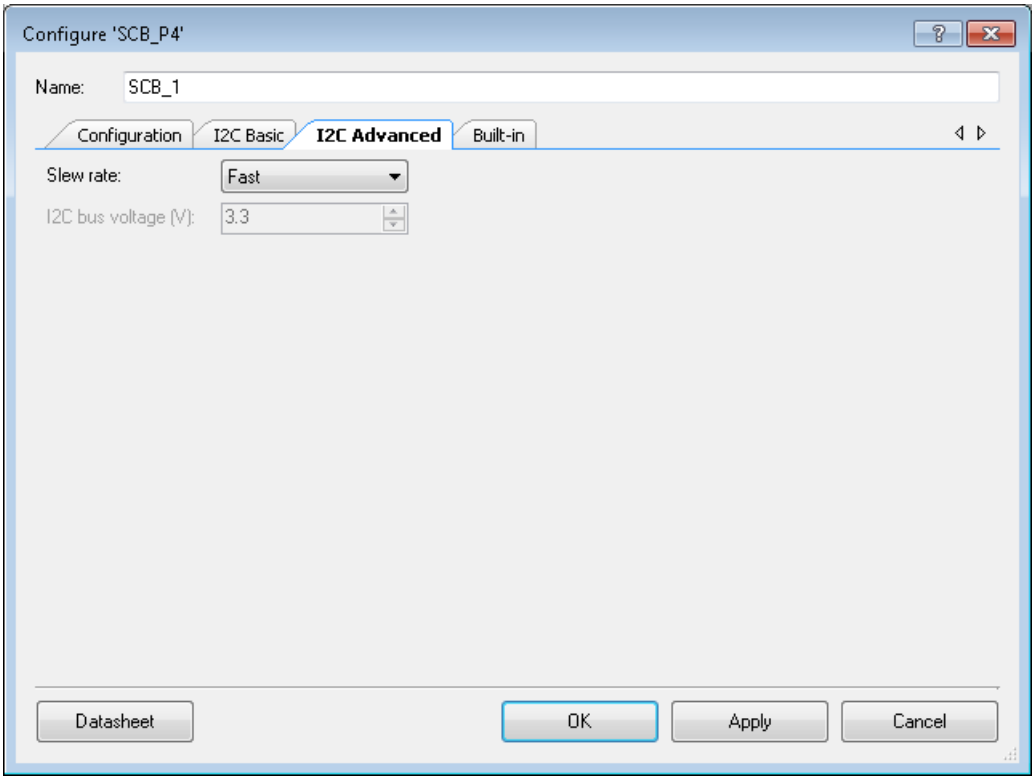
Use this option to enable the component to wake the system from Deep Sleep when a slave address match occurs. It is only available for Slave or Multi-Master-Slave mode.

For PSoC 4100/PSoC 4200 devices, the Slave address (7-bits) must be even (bit 0 equal zero) when this option is enabled.

For all supported devices, the data rate must be less than or equal to 400 kbps for Multi-Master-Slave mode when this option is enabled.

Refer to the Low power modes section under the I²C chapter in this document; refer also to the *Power Management APIs* section of the *System Reference Guide* for more information.

Advanced I2C Parameters



The **I2C Advanced** tab contains the following parameters:

Slew rate

This option allows to control slew rate setting of the SCL and SDA pins. The slow slew rate increases the fall time on the lines, reducing EMI and coupling with neighboring signals. For devices supporting GPIO Over-Voltage Tolerance (GPIO_OVT) pins, I2C FM+ options should be used when I^2C data rate is greater than 400 kbps. This option also requires the I2C bus voltage to be defined. Refer to the *Device Datasheet* to determine which pins are GPIO_OVT capable. Default is fast.

Notes

- GPIO_OVT pins are fully compliant with the I^2C specification but the slew rate must be set appropriately:
  - **Slew rate** "Slow" for Standard mode (100 kbps) and Fast mode (400 kbps)
  - **Slew rate** "I2C FM+" for Fast mode plus (1 Mbps)

  Common GPIO pins are not completely compliant with the I^2C specification. Refer to the *Device Datasheet* for the details.
- **Slew rate** settings are applied to all pins of the associated port.

I2C bus voltage (V)

This option is only applicable for PSoC 4100 BLE/PSoC 4200 BLE devices. It specifies the voltage applied to the I^2C pull up resistors when Slew rate is I2C FM+. The voltage no less than applied to I^2C pulls up resistors must be provided by the V_{DDD} supply input, otherwise the I^2C pins cannot be placed. Valid values of V_{DDD} are determined by the settings in the Design-Wide Resources System Editor (in the *<project>.cydwr* file). This range check is performed outside this dialog; the results appear in the Notice List window if the check fails. Default is 3.3 V.

I2C APIs

Application Programming Interface (API) functions allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent section discusses each function in more detail.

By default, PSoC Creator assigns the instance name “SCB_1” to the first instance of a component in a given design. You can rename the instance to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “SCB”.

Function	Description
SCB_Start()	Starts the SCB component.
SCB_Init()	Initialize the SCB component according to defined parameters in the customizer.
SCB_Enable()	Enables the SCB component operation.
SCB_Stop()	Disable the SCB component.
SCB_Sleep()	Prepares the SCB component to enter Deep Sleep.
SCB_Wakeup()	Prepares component for Active mode operation after Deep Sleep.
SCB_I2CInit()	Configures the SCB component for operation in I2C mode. Only applicable when component is in unconfigured mode.
SCB_I2CSlaveStatus()	Returns slave status flags.
SCB_I2CSlaveClearReadStatus()	Returns the slave read status flags and clears slave read status flags.
SCB_I2CSlaveClearWriteStatus()	Returns the slave write status and clears the slave write status flags.
SCB_I2CSlaveSetAddress()	Sets slave address, a value between 0 and 127 (0x00 to 0x7F).
SCB_I2CSlaveSetAddressMask()	Sets slave address mask, a value between 0 and 254 (0x00 to 0xFE).
SCB_I2CSlaveInitReadBuf()	Sets up the slave receive data buffer (master <- slave).
SCB_I2CSlaveInitWriteBuf()	Sets up the slave write buffer (master -> slave).
SCB_I2CSlaveGetReadBufSize()	Returns the number of bytes read by the master since SCB_I2CSlaveClearReadBuf() was called.

Function	Description
SCB_I2CSlaveGetWriteBufSize()	Returns the number of bytes written by the master since SCB_I2CSlaveClearWriteBuf() was called.
SCB_I2CSlaveClearReadBuf()	Resets the read buffer counter to zero.
SCB_I2CSlaveClearWriteBuf()	Resets the write buffer counter to zero.
SCB_I2CMasterStatus()	Returns the master status.
SCB_I2CMasterClearStatus()	Returns the master status and clears the status flags.
SCB_I2CMasterWriteBuf()	Writes the referenced data buffer to a specified slave address.
SCB_I2CMasterReadBuf()	Reads data from the specified slave address and places the data in the referenced buffer.
SCB_I2CMasterSendStart()	Generates a start condition and sends specified slave address.
SCB_I2CMasterSendRestart()	Generates a restart condition and sends specified slave address.
SCB_I2CMasterSendStop()	Generates a stop condition.
SCB_I2CMasterWriteByte()	Writes a single byte. This is a manual command that should only be used with the SCB_I2CMasterSendStart() or SCB_I2CMasterSendRestart() functions.
SCB_I2CMasterReadByte()	Reads a single byte. This is a manual command that should only be used with the SCB_I2CMasterSendStart() or SCB_I2CMasterSendRestart() functions.
SCB_I2CMasterGetReadBufSize()	Returns the number of bytes that have been transferred with the SCB_I2CMasterReadBuf() function.
SCB_I2CMasterGetWriteBufSize()	Returns the number of bytes that have been transferred with the SCB_I2CMasterWriteBuf() function.
SCB_I2CMasterClearReadBuf()	Resets the read buffer pointer back to the beginning of the buffer.
SCB_I2CMasterClearWriteBuf()	Resets the write buffer pointer back to the beginning of the buffer.

void SCB_Start(void)

Description: Invokes SCB_Init() and SCB_Enable(). After this function call the component is enabled and ready for operation. This is the preferred method to begin component operation.

When configuration is set to “Unconfigured SCB”, the component must first be initialized to operate in one of the following configurations: I^2C, SPI, UART or EZ I^2C. Otherwise this function does not enable the component.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Init(void)

Description: Initializes SCB component to operate in one of selected configurations: I^2C, SPI, UART or EZ I^2C.

When the configuration is set to “Unconfigured SCB”, this function does not do any initialization. Use mode-specific initialization APIs instead: SCB_I2CInit, SCB_SpiInit, SCB_UartInit or SCB_EzI2CInit.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Enable(void)

Description: Enables SCB component operation: activates the hardware and internal interrupt.

For I^2C and EZ I^2C modes the interrupt is internal and mandatory for operation. For SPI and UART modes the interrupt can be configured as none, internal or external.

The SCB configuration should be not changed when the component is enabled. Any configuration changes should be made after disabling the component.

When configuration is set to “Unconfigured SCB”, the component must first be initialized to operate in one of the following configurations: I^2C, SPI, UART or EZ I^2C, using the mode-specific initialization API. Otherwise this function does not enable the component.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Stop(void)

Description: Disables the SCB component: disable the hardware and internal interrupt. Refer to the function SCB_Enable() for the interrupt configuration details.

This function disables the SCB component without checking to see if communication is in progress. Before calling this function it may be necessary to check the status of communication to make sure communication is complete. If this is not done then communication could be stopped mid byte and corrupted data could result.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Sleep(void)

Description: Prepares the SCB component to enter Deep Sleep.

The “Enable wakeup from Deep Sleep Mode” selection has an influence on this function implementation:

- Checked: configures the component to be wakeup source from Deep Sleep.
- Unchecked: stores the current component state (enabled or disabled) and disables the component. See SCB_Stop() function for details about component disabling.

Call the SCB_Sleep() function before calling the CyPmSysDeepSleep() function. Refer to the PSoC Creator *System Reference Guide* for more information about power management functions and *Low power* section of this document for the selected mode.

This function should not be called before entering Sleep.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Wakeup(void)

Description: Prepares the SCB component for Active mode operation after Deep Sleep.

The “Enable wakeup from Deep Sleep Mode” selection has influence on this function implementation:

- Checked: restores the component Active mode configuration.
- Unchecked: enables the component if it was enabled before enter Deep Sleep.

This function should not be called after exiting Sleep.

Parameters: None

Return Value: None

Side Effects: Calling the SCB_Wakeup() function without first calling the SCB_Sleep() function may produce unexpected behavior.

void SCB_I2CInit(SCB_I2C_INIT_STRUCT *config)

Description: Configures the SCB for I²C operation.

This function is **intended specifically** to be used when the SCB configuration is set to "Unconfigured SCB" in the customizer. After initializing the SCB in I2C mode using this function, the component can be enabled using the SCB_Start() or SCB_Enable() function.

This function uses a pointer to a structure that provides the configuration settings. This structure contains the same information that would otherwise be provided by the customizer settings.

Parameters: config: pointer to a structure that contains the following list of fields. These fields match the selections available in the customizer. Refer to the customizer for further description of the settings.

Field	Description
uint32 mode	Mode of operation for I2C. The following defines are available choices: • SCB_I2C_MODE_SLAVE • SCB_I2C_MODE_MASTER • SCB_I2C_MODE_MULTI_MASTER • SCB_I2C_MODE_MULTI_MASTER_SLAVE
uint32 oversampleLow	Oversampling factor for the low phase of the I2C clock. Ignored for Slave mode operation. The oversampling factors need to be chosen in conjunction with the clock rate in order to generate the desired rate of I2C operation.
uint32 oversampleHigh	Oversampling factor for the high phase of the I2C clock. Ignored for Slave mode operation.
uint32 enableMedianFilter	This field is left for compatibility and its value is ignored. Median filter is enabled or disabled depends on the data rate and operation mode.
uint32 slaveAddr	7-bit slave address. Ignored for non-slave modes.
uint32 slaveAddrMask	8-bit slave address mask. Bit 0 must have a value of 0. Ignored for non-slave modes.
uint32 acceptAddr	0 – disable 1 – enable When enabled the matching address is received into the Rx FIFO.
uint32 enableWake	0 – disable 1 – enable Ignored for non-slave modes.
uint8 enableByteMode	Ignored for all devices other than PSoC 4100 BLE/PSoC 4200 BLE 0 – disable 1 – enable When enabled the TX and RX FIFO depth is 16 bytes.
uin16 dataRate	Data rate in kbps used while the of I²C is in operation. Valid values are between 1 and 1000. **Note** This filed must be initialized for correct operation if Unconfigured SCB was utilized with previous version of the component.

Return Value: None

Side Effects: None

uint32 SCB_I2CSlaveStatus(void)

Description: Returns the slave's communication status.

Parameters: None

Return Value: uint32: Current status of I^2C slave.

This status incorporates read and write status constants. Each constant is a bit field value. The value returned may have multiple bits set to indicate the status of the read or write transfer.

Slave Status constants	Description
SCB_I2C_SSTAT_RD_CMPLT	Slave read transfer complete. Set when master indicates it is done reading by sending a NAK [3]. The read error condition status bit must be checked to ensure that the read transfer was completed successfully.
SCB_I2C_SSTAT_RD_BUSY	Slave read transfer is in progress. Set when master addresses slave with a read, cleared when RD_CMPLT is set.
SCB_I2C_SSTAT_RD_OVFL	Master attempted to read more bytes than are in buffer. Slave continually returns 0xFF byte in this case.
SCB_I2C_SSTAT_RD_ERR	Slave captured error on the bus during a read transfer. The sources of error are: misplaced Start or Stop condition or lost arbitration while slave drives SDA.
SCB_I2C_SSTAT_WR_CMPLT	Slave write transfer complete. Set at reception of a Stop or ReStart condition. The write error condition status bit must be checked to ensure that write transfer was completed successfully.
SCB_I2C_SSTAT_WR_BUSY	Slave write transfer is in progress. Set when the master addresses the slave with a write, cleared when WR_CMPLT is set.
SCB_I2C_SSTAT_WR_OVFL	Master attempted to write past end of buffer. Further bytes are ignored.
SCB_I2C_SSTAT_WR_ERR	Slave captured error on the bus during write transfer. The sources of error are: misplaced Start or Stop condition or lost arbitration while slave drives SDA. The write buffer may contain invalid bytes or part of the data transfer when SCB_I2C_SSTAT_WR_ERR is set. It is recommended to discard write buffer content in this case.

Side Effects: None

[3] NAK is an abbreviation for negative acknowledgment or not acknowledged. I^2C documents commonly use NACK while the rest of the networking world uses NAK. They mean the same thing.

uint32 SCB_I2CSlaveClearReadStatus(void)

Description: Clears the read status flags and returns their values. No other status flags are affected.

Parameters: None

Return Value: uint32: Current read status of slave. See the SCB_I2CSlaveStatus() function for constants.

Side Effects: This function does not clear SCB_I2C_SSTAT_RD_BUSY.

uint32 SCB_I2CSlaveClearWriteStatus(void)

Description: Clears the write status flags and returns their values. No other status flags are affected.

Parameters: None

Return Value: uint32: Current write status of slave. See the SCB_I2CSlaveStatus() function for constants.

Side Effects: This function does not clear SCB_I2C_SSTAT_WR_BUSY.

void SCB_I2CSlaveSetAddress(uint32 address)

Description: Sets the I^2C slave address

Parameters: uint32 address: I^2C slave address. This address is the 7-bit right-justified slave address and does not include the R/W bit.

The address value is not checked to see if it violates the I^2C spec. The preferred addresses are between 8 and 120 (0x08 to 0x78).

Return Value: None

Side Effects: None

void SCB_I2CSlaveSetAddressMask(uint32 addressMask)

Description: Sets the I^2C slave address

Parameters: uint32 addressMask: I^2C slave address mask.

Bit value 0 – excludes bit from address comparison.

Bit value 1 – the bit needs to match with the corresponding bit of the I^2C slave address.

The range of valid values is between 0 and 254 (0x00 to 0xFE). The LSB of the address mask must be 0 because it corresponds to R/W bit within I^2C slave address byte.

Return Value: None

Side Effects: None

void SCB_I2CSlaveInitReadBuf(uint8 * rdBuf, uint32 bufSize)

Description: Sets the buffer pointer and size of the read buffer. This function also resets the transfer count returned with the SCB_I2CSlaveGetReadBufSize() function.

Parameters: uint8* rdBuf: Pointer to the data buffer to be read by the master.

uint32 bufSize: Size of the buffer exposed to the I^2C master.

Return Value: None

Side Effects: If this function is called during a bus transaction, data from the previous buffer location and the beginning of the current buffer may be transmitted.

void SCB_I2CSlaveInitWriteBuf(uint8 * wrBuf, uint32 bufSize)

Description: Sets the buffer pointer and size of the write buffer. This function also resets the transfer count returned with the SCB_I2CSlaveGetWriteBufSize() function.

Parameters: uint8* wrBuf: Pointer to the data buffer to be written by the master.

uint32 bufSize: Size of the write buffer exposed to the I^2C master.

Return Value: None

Side Effects: If this function is called during a bus transaction, data may be received in the previous buffer and the current buffer location.

uint32 SCB_I2CSlaveGetReadBufSize(void)

Description: Returns the number of bytes read by the I^2C master since the SCB_I2CSlaveInitReadBuf() or SCB_I2CSlaveClearReadBuf() function was called.

The maximum return value is the size of the read buffer.

Parameters: None

Return Value: uint32: Bytes read by master. If the transfer is not yet complete, it returns zero until transfer completion.

Side Effects: The returned value is not valid if SCB_I2C_SSTAT_RD_ERR was captured by the slave.

uint32 SCB_I2CSlaveGetWriteBufSize(void)

Description: Returns the number of bytes written by the I^2C master since the SCB_I2CSlaveInitWriteBuf() or SCB_I2CSlaveClearWriteBuf() function was called.
The maximum return value is the size of the write buffer.

Parameters: None

Return Value: uint32: Bytes written by master. If the transfer is not yet complete, it returns the byte count transferred so far.

Side Effects: The returned value is not valid if SCB_I2C_SSTAT_WR_ERR was captured by the slave.

void SCB_I2CSlaveClearReadBuf(void)

Description: Resets the read pointer to the first byte in the read buffer. The next byte read by the master will be the first byte in the read buffer.

Parameters: None

Return Value: None

Side Effects: None

void SCB_I2CSlaveClearWriteBuf(void)

Description: Resets the write pointer to the first byte in the write buffer. The next byte written by the master will be the first byte in the write buffer.

Parameters: None

Return Value: None

Side Effects: None

uint32 SCB_I2CMasterStatus(void)

Description: Returns the master's communication status.

Parameters: None

Return Value: uint32: Current status of I^2C master. This status incorporates status constants. Each constant is a bit field value. The value returned may have multiple bits set to indicate the status of the read or write transfer.

Master Status constants	Description
SCB_I2C_MSTAT_RD_CMPLT	Read transfer complete. The error condition status bits must be checked to ensure that read transfer was completed successfully.
SCB_I2C_MSTAT_WR_CMPLT	Write transfer complete. The error condition status bits must be checked to ensure that write transfer was completed successfully.
SCB_I2C_MSTAT_XFER_INP	Transfer in progress.
SCB_I2C_MSTAT_XFER_HALT	Transfer has been halted. The I^2C bus is waiting for ReStart or Stop condition generation.
SCB_I2C_MSTAT_ERR_SHORT_XFER	**Error condition**: Write transfer completed before all bytes were transferred. The slave NAKed the byte which was expected to be ACKed.
SCB_I2C_MSTAT_ERR_ADDR_NAK	**Error condition:** Slave did not acknowledge address.
SCB_I2C_MSTAT_ERR_ARB_LOST	**Error condition**: Master lost arbitration during communications with slave.
SCB_I2C_MSTAT_ERR_BUS_ERROR	**Error condition**: bus error occurred during master transfer due to misplaced Start or Stop condition on the bus.
SCB_I2C_MSTAT_ERR_ABORT_XFER	**Error condition**: Slave was addressed by another master while master performed the start condition generation. As a result, master has automatically switched to slave mode and is responding. The master transaction has not taken place This error condition only applicable for Multi-Master-Slave mode.
SCB_I2C_MSTAT_ERR_XFER	**Error condition**: This is the ORed value of all error conditions provided above.

Side Effects: None

uint32 SCB_I2CMasterClearStatus(void)

Description: Clears all status flags and returns the master status.

Parameters: None

Return Value: uint32: Current status of master. See the SCB_I2CMasterStatus() function for constants.

Side Effects: None

uint32 SCB_I2CMasterWriteBuf(uint32 slaveAddress, uint8 * wrData, uint32 cnt, uint32 mode)

Description: Automatically writes an entire buffer of data to a slave device. Once the data transfer is initiated by this function, further data transfer is handled by the included ISR.

Enables the I^2C interrupt and clears SCB_ I2C_MSTAT_WR_CMPLT status.

Parameters: uint32 slaveAddress: Right-justified 7-bit Slave address (valid range 8 to 120).

uint8 wrData: Pointer to buffer of data to be sent.

uint32 cnt: Number of bytes of buffer to send.

uint32 mode: Transfer mode defines:

(1) Whether a start or restart condition is generated at the beginning of the transfer, and

(2) Whether the transfer is completed or halted before the stop condition is generated on the bus.

Transfer mode, mode constants may be ORed together.

Transfer Mode constants	Description
SCB_I2C_MODE_COMPLETE_XFER	Perform complete transfer from Start to Stop.
SCB_I2C_MODE_REPEAT_START	Send Repeat Start instead of Start. A Stop is generated after transfer is completed unless NO_STOP is specified.
SCB_I2C_MODE_NO_STOP	Execute transfer without a Stop. The following transfer expected to perform ReStart.

Return Value: **uint32**: Error status.

Error Status constants	Description
SCB_I2C_MSTR_NO_ERROR	Function complete without error. The master started the transfer.
SCB_I2C_MSTR_BUS_BUSY	Bus is busy. Nothing was sent on the bus. The attempt has to be retried.
SCB_I2C_MSTR_NOT_READY	Master is not ready for to start transfer. A master still has not completed previous transaction or a slave operation is in progress (in multi-master-slave configuration). Nothing was sent on the bus. The attempt has to be retried.

Side Effects: None

uint32 SCB_I2CMasterReadBuf(uint32 slaveAddress, uint8 * rdData, uint32 cnt, uint32 mode)

Description: Automatically reads an entire buffer of data from a slave device. Once the data transfer is initiated by this function, further data transfer is handled by the included ISR.

Enables the I^2C interrupt and clears SCB_ I2C_MSTAT_RD_CMPLT status.

Parameters: uint32 slaveAddress: Right-justified 7-bit Slave address (valid range 8 to 120).

uint8 rdData: Pointer to buffer where to put data from slave.

uint32 cnt: Number of bytes of buffer to read.

uint32 mode: Transfer mode defines:

(1) Whether a start or restart condition is generated at the beginning of the transfer, and

(2) Whether the transfer is completed or halted before the stop condition is generated on the bus.

Transfer mode, mode constants may be ORed together. See SCB_I2CMasterWriteBuf() function for constants.

Return Value: uint32: Error status.

Error Status constants	Description
SCB_I2C_MSTR_NO_ERROR	Function complete without error. The master started the transfer.
SCB_I2C_MSTR_BUS_BUSY	Bus is busy. Nothing was sent on the bus. The attempt has to be retried.
SCB_I2C_MSTR_NOT_READY	Master is not ready for to start transfer. A master still has not completed previous transaction or a slave operation is in progress (in multi-master-slave configuration). Nothing was sent on the bus. The attempt has to be retried.

Side Effects: None

uint32 SCB_I2CMasterGetReadBufSize(void)

Description: Returns the number of bytes that has been transferred with an SCB_I2CMasterReadBuf() function.

Parameters: None

Return Value: uint32: Byte count of transfer. If the transfer is not yet complete, it returns the byte count transferred so far.

Side Effects: This function returns an invalid value if SCB_I2C_MSTAT_ERR_ARB_LOST or SCB_I2C_MSTAT_ERR_BUS_ERROR occurred during the read transfer.

uint32 SCB_I2CMasterGetWriteBufSize(void)

Description: Returns the number of bytes that have been transferred with an SCB_I2CMasterWriteBuf() function.

Parameters: None

Return Value: uint32: Byte count of transfer. If the transfer is not yet complete, it returns zero unit transfer completion.

Side Effects: This function returns an invalid value if SCB_I2C_MSTAT_ERR_ARB_LOST or SCB_I2C_MSTAT_ERR_BUS_ERROR occurred during the write transfer.

void SCB_I2CMasterClearReadBuf(void)

Description: Resets the read buffer pointer back to the first byte in the buffer.

Parameters: None

Return Value: None

Side Effects: None

void SCB_I2CMasterClearWriteBuf(void)

Description: Resets the write buffer pointer back to the first byte in the buffer.

Parameters: None

Return Value: None

Side Effects: None

uint32 SCB_I2CMasterSendStart(uint32 slaveAddress, uint32 bitRnW)

Description: Generates Start condition and sends slave address with read/write bit. Disables the I^2C interrupt.

This function is blocking. It does not return until the Start condition and address byte are sent, a ACK/NAK is received, or errors have occurred.

Parameters: uint32 slaveAddress: Right justified 7-bit Slave address (valid range 8 to 120).

uint32 bitRnW: Direction of the following transfer. It is defined by read/write bit within address byte.

Direction constants	Description
SCB_I2C_WRITE_XFER_MODE	Set write direction for the following transfer.
SCB_I2C_READ_XFER_MODE	Set read direction for the following transfer.

Return Value: uint32: Error status.

Error Status constants	Description
SCB_I2C_MSTR_NO_ERROR	Function complete without error.
SCB_I2C_MSTR_BUS_BUSY	Bus is busy. Nothing was sent on the bus. The attempt has to be retried.
SCB_I2C_MSTR_NOT_READY	Master is not ready for to start transfer. A master still has not completed previous transaction or a slave operation is in progress (in multi-master-slave configuration). Nothing was sent on the bus. The attempt has to be retried.
SCB_I2C_MSTR_ERR_LB_NAK	**Error condition**: Last byte was NAKed.
SCB_I2C_MSTR_ERR_ARB_LOST	**Error condition**: Master lost arbitration.
SCB_I2C_MSTR_ERR_BUS_ERR	**Error condition**: Master encountered a bus error. Bus error is misplaced start or stop detection.
SCB_I2C_MSTR_ERR_ABORT_START	**Error condition**: The start condition generation was aborted due to beginning of Slave operation. This error condition is only applicable for Multi-Master-Slave mode.

Side Effects: None

uint32 SCB_I2CMasterSendRestart(uint32 slaveAddress, uint32 bitRnW)

Description: Generates ReStart condition and sends slave address with read/write bit.

If the last transaction was a read the NAK is sent before the ReStart to complete the transaction.

This function is blocking and does not return until the ReStart condition and address are sent, a ACK/NAK is received, or errors have occurred.

Parameters: uint32 slaveAddress: Right-justified 7-bit Slave address (valid range 8 to 120).

uint32 bitRnW: Direction of the following transfer. It is defined by read/write bit within address byte. See SCB_I2CMasterSendStart() function for constants.

Return Value: uint32: Error status. See SCB_I2CMasterSendStart() function for constants.

Side Effects: A valid Start or ReStart condition must be generated before calling this function. This function does nothing if Start or ReStart conditions failed before this function was called.

For read transaction, at least one byte has to be read before ReStart generation.

uint32 SCB_I2CMasterSendStop(void)

Description: Generates Stop condition on the bus. The NAK is generated before Stop in case of a read transaction.

At least one byte has to be read if a Start or ReStart condition with read direction was generated before.

This function is blocking and does not return until a Stop condition is generated or error occurred.

Parameters: None

Return Value: uint32: Error status. See the SCB_MasterSendStart() command for constants.

Side Effects: A valid Start or ReStart condition must be generated before calling this function. This function does nothing if Start or ReStart condition failed before this function was called.

For read transaction, at least one byte has to be read before Stop generation.

uint32 SCB_I2CMasterWriteByte(uint32 theByte)

Description: Sends one byte to a slave.

This function is blocking and does not return until the byte is transmitted or an error occurs.

Parameters: uint32 theByte: Data byte to send to the slave.

Return Value: uint32: Error status.

Error Status constants	Description
SCB_I2C_MSTR_NO_ERROR	Function complete without error.
SCB_I2C_MSTR_NOT_READY	Master is not active master on the bus. A Slave operation may be in progress. Nothing was sent on the bus. The attempt has to be retried.
SCB_I2C_MSTR_ERR_LB_NAK	**Error condition**: Last byte was NAKed.
SCB_I2C_MSTR_ERR_ARB_LOST	**Error condition**: Master lost arbitration.
SCB_I2C_MSTR_ERR_BUS_ERR	**Error condition**: Master encountered a bus error. Bus error is misplaced start or stop detection.

Side Effects: A valid Start or ReStart condition must be generated before calling this function. This function does nothing if Start or ReStart conditions failed before this function was called.

uint32 SCB_I2CMasterReadByte(uint32 ackNack)

Description: Reads one byte from a slave and generates ACK or prepares to generate NAK. The NAK will be generated before Stop or ReStart condition by SCB_I2CMasterSendStop() or SCB_I2CMasterSendRestart() function appropriately.

This function is blocking. It does not return until a byte is received or an error occurs.

Parameters: uint32 ackNack: Response to received byte.

Response constants	Description
SCB_I2C_ACK_DATA	Generates ACK. The master notifies slave that transfer continues.
SCB_I2C_NAK_DATA	Prepares to generate NAK. The master will notify slave that transfer is completed.

Return Value: uint32: Byte read from the slave. In case of error the MSB of returned data is set to 1.

Side Effects: A valid Start or ReStart condition must be generated before calling this function. This function does nothing and returns invalid byte value if Start or ReStart conditions failed before this function was called.

Global Variables

Knowledge of these variables is not required for normal operations.

Variable	Description
SCB_initVar	SCB_initVar indicates whether the SCB component has been initialized. The variable is initialized to 0 and set to 1 the first time SCB_Start() is called. This allows the component to restart without reinitialization after the first call to the SCB_Start() routine. If reinitialization of the component is required, then the SCB_Init() function can be called before the SCB_Start() or SCB_Enable() function.

I2C Function Appliance

Function	Slave	Master	Multi-Master	Multi-Master-Slave
SCB_I2CInit()	+	+	+	+
SCB_I2CSlaveStatus()	+	–	–	+
SCB_I2CSlaveClearReadStatus()	+	–	–	+
SCB_I2CSlaveClearWriteStatus()	+	–	–	+
SCB_I2CSlaveSetAddress()	+	–	–	+
SCB_I2CSlaveSetAddressMask()	+	–	–	+
SCB_I2CSlaveInitReadBuf()	+	–	–	+
SCB_I2CSlaveInitWriteBuf()	+	–	–	+
SCB_I2CSlaveGetReadBufSize()	+	–	–	+
SCB_I2CSlaveGetWriteBufSize()	+	–	–	+
SCB_I2CSlaveClearReadBuf()	+	–	–	+
SCB_I2CSlaveClearWriteBuf()	+	–	–	+
SCB_I2CMasterStatus()	–	+	+	+
SCB_I2CMasterClearStatus()	–	+	+	+
SCB_I2CMasterWriteBuf()	–	+	+	+
SCB_I2CMasterReadBuf()	–	+	+	+
SCB_I2CMasterSendStart()	–	+	+	+
SCB_I2CMasterSendRestart()	–	+	+	+
SCB_I2CMasterSendStop()	–	+	+	+
SCB_I2CMasterWriteByte()	–	+	+	+
SCB_I2CMasterReadByte()	–	+	+	+

Function	Slave	Master	Multi-Master	Multi-Master-Slave
SCB_I2CMasterGetReadBufSize()	–	+	+	+
SCB_I2CMasterGetWriteBufSize()	–	+	+	+
SCB_I2CMasterClearReadBuf()	–	+	+	+
SCB_I2CMasterClearWriteBuf()	–	+	+	+

Bootloader Support

The SCB component can be used as a communication component for the Bootloader. Only an SCB in I^2C mode can be used as a Bootloader. You should use the following configurations to support communication protocol from an external system to the Bootloader:

- Configuration: I2C
- I2C Mode: Slave or Multi-Master-Slave
- Data Rate: Must match Host (boot device) data rate.
- Slave Address: Must match Host (boot device) selected slave address.

Refer to the Bootloader component datasheet for more information.

The SCB component provides a set of API functions for Bootloader use.

Function	Description
SCB_CyBtldrCommStart()	Starts the I^2C component and enables its interrupt.
SCB_CyBtldrCommStop()	Disable the I^2C component and disables its interrupt.
SCB_CyBtldrCommReset()	Sets read and write I^2C buffers to the initial state and resets the slave status.
SCB_CyBtldrCommWrite()	Allows the bootloadable to write data to the bootloader host. This function handles polling to allow a block of data to be completely sent to the host device.
SCB_CyBtldrCommRead()	Allows the bootloadable to read data from the bootloader host. This function handles polling to allow a block of data to be completely received from the host device.

void SCB_CyBtldrCommStart(void)

Description: Starts the I²C component and enables its interrupt.
Every incoming I²C write transaction is treated as a command for the bootloader.
Every incoming I²C read transaction returns 0xFF until the bootloader provides a response to the executed command.

Parameters: None

Return Value: None

Side Effects: None

void SCB_CyBtldrCommStop(void)

Description: Disables the I²C component and disables its interrupt.

Parameters: None

Return Value: None

Side Effects: None

void SCB_CyBtldrCommReset(void)

Description: Sets read and write I²C buffers to the initial state and resets the slave status.

Parameters: None

Return Value: None

Side Effects: None

cystatus SCB_CyBtldrCommWrite(const uint8 pData[], uint16 size, uint16 * count, uint8 timeOut)

Description: Allows the bootloadable to write data to the bootloader host. The function handles polling to allow a block of data to be completely sent to the host device.

Parameters: const pData[]: Pointer to the block of data to send to the bootloader host.

uint16 size: Number of bytes to send to bootlaoder host.

uint16 *count: Pointer to variable to write the number of bytes actually written to bootlaoder host.

uint8 timeOut: Number of units in 10 ms to wait before returning because of a timeout.

Return Value: cystatus: Returns CYRET_SUCCESS if no problem was encountered or returns the value that best describes the problem. For more information refer to the “Return Codes” section of the *System Reference Guide*.

Side Effects: None

cystatus SCB_CyBtldrCommRead(uint8 pData[], uint16 size, uint16 * count, uint8 timeOut)

Description: Allows the bootloadable to read data from the bootloader host. The function handles polling to allow a block of data to be completely received from the host device.

Parameters: uint8 pData[]: Pointer to the block of data to be read from bootloader host.

uint16 size: Number of bytes to be read from bootloader host.

uint16 *count: Pointer to variable to write the number of bytes actually read by bootloader host.

uint8 timeOut: Number of units in 10 ms to wait before returning because of a timeout.

Return Value: cystatus: Returns CYRET_SUCCESS if no problem was encountered or returns the value that best describes the problem. For more information, refer to the “Return Codes” section of the *System Reference Guide*.

Side Effects: None

I2C Functional Description

This component supports I^2C Slave, Master, Multi-Master, and Multi-Master-Slave configurations. The following sections provide an overview of how to use the component in these configurations.

This component requires that you enable global interrupts since the I^2C hardware is interrupt-driven. Even though this component requires interrupts, you do not need to add any code to the ISR (interrupt service routine). The component services all interrupts (data transfers) independent of your code. The memory buffers allocated for this interface look like simple dual-port memory between your application and the I^2C master/slave.

Slave Operation

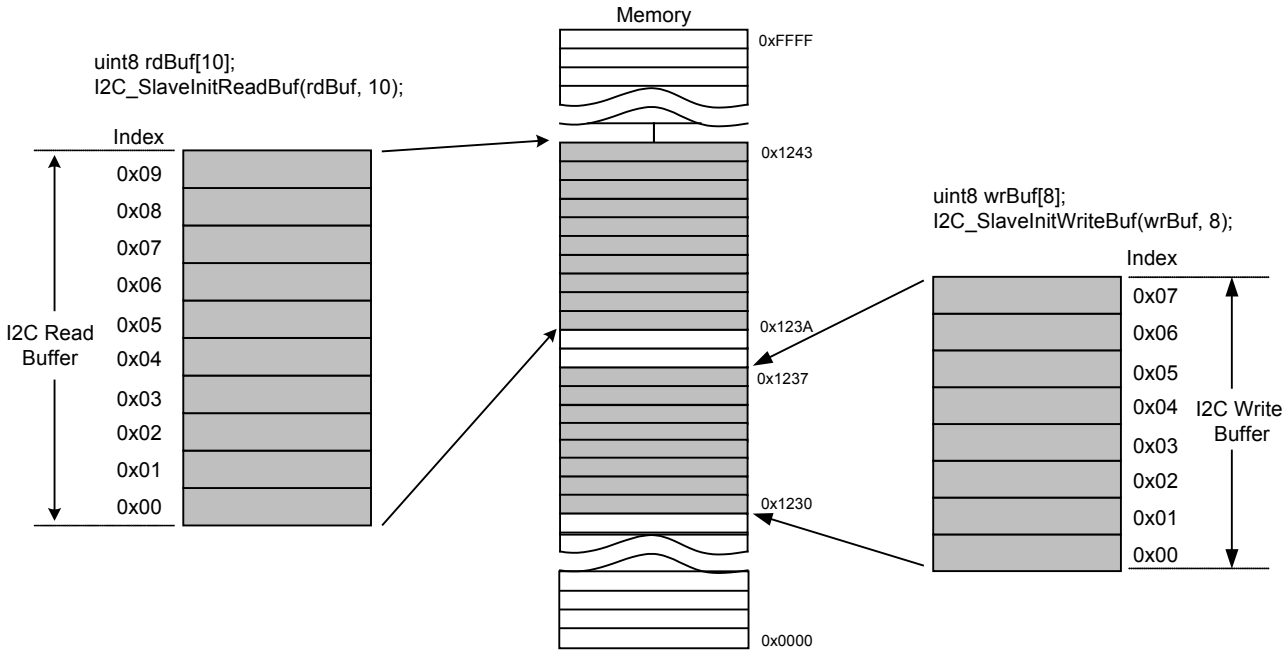
The slave interface consists of two buffers in memory, one for data written to the slave by a master and a second buffer for data read by a master from the slave. Remember that reads and

writes are from the perspective of the I²C master. The I²C slave read and write buffers are set by the initialization commands below. These commands do not allocate memory, but instead copy the array pointer and size to the internal component variables. You must instantiate the arrays used for the buffers because they are not automatically generated by the component. The same buffer may be used for both read and write buffers, but you must be careful to manage the data properly.

```
void SCB_I2CSlaveInitReadBuf(uint8 * rdBuf, uint32 bufSize)
void SCB_I2CSlaveInitWriteBuf(uint8 * wrBuf, uint32 bufSize)
```

Using the functions above sets a pointer and byte count for the read and write buffers. The bufSize for these functions may be less than or equal to the actual array size, but it should never be larger than the available memory pointed to by the rdBuf or wrBuf pointers.

Figure 1. Slave Buffer Structure



When the SCB_I2CSlaveInitReadBuf() or SCB_I2CSlaveInitWriteBuf() function is called, the internal index is set to the first value in the array pointed to by rdBuf and wrBuf, respectively. As bytes are read or written by the I²C master, the index is incremented until the offset is one less than the bufSize. At any time the number of bytes transferred may be queried by calling either SCB_I2CSlaveGetReadBufSize() or SCB_I2CSlaveGetWriteBufSize() for the read and write buffers, respectively. Reading or writing more bytes than are in the buffers causes an overflow. The overflow status is set in the slave status byte and may be read with the SCB_I2CSlaveStatus() API.

To reset the index back to the beginning of the array, use the following commands:

```
void SCB_I2CSlaveClearReadBuf(void)
```

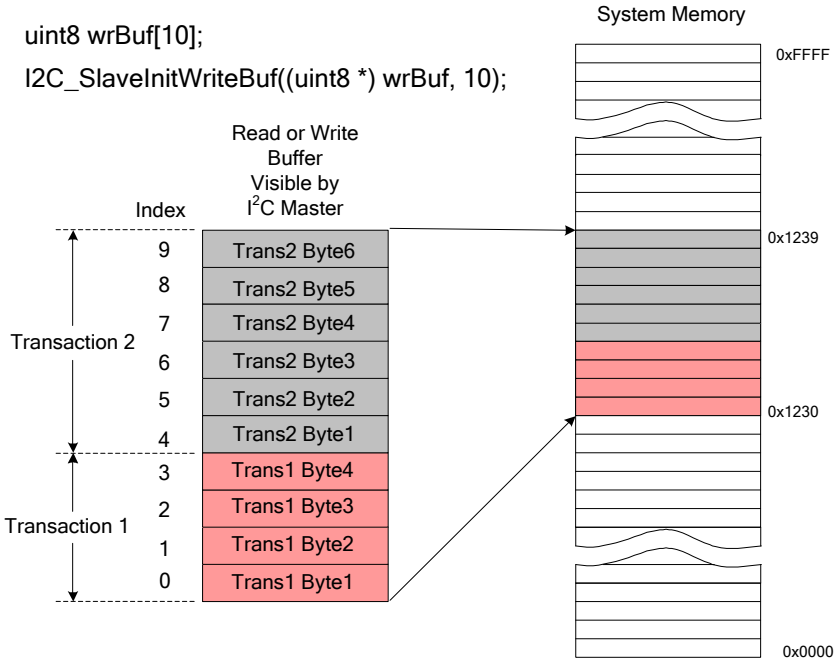
```
void SCB_I2CSlaveClearWriteBuf(void)
```

This resets the index back to zero. The next byte read or written by the I^2C master is the first byte in the array. Before these clear buffer commands are used, the data in the arrays should be read or updated.

Multiple reads or writes by the I^2C master continue to increment the array index until the clear buffer commands are used or the array index attempts to grow beyond the array size. Figure 2 shows an example where an I^2C master has executed two write transactions. The first write was four bytes and the second write was six bytes. The sixth byte in the second transaction was NAKed by the slave to signal that the end of the buffer had occurred. If the master tried to write a seventh byte for the second transaction or started to write more bytes with a third transaction, each byte would be NAKed and discarded until the buffer is reset.

Using the SCB_I2CSlaveClearWriteBuf() function after the first transaction resets the index back to zero and causes the second transaction to overwrite the data from the first transaction. Make sure data is not lost by overflowing the buffer. The data in the buffer should be processed by the slave before resetting the buffer index.

Figure 2. System Memory



Both the read and write buffers have four status bits to signal transfer complete, transfer in progress, and buffer overflow. When a transfer starts, the busy flag is set. When the transfer is complete, the transfer complete flag is set and the busy flag is cleared. If a second transfer is started, both the busy and transfer complete flags may be set at the same time. The following table shows read and write status flags.

Slave Status Constants	Description
SCB_I2C_SSTAT_RD_CMPLT	Slave read transfer complete.
SCB_I2C_SSTAT_RD_BUSY	Slave read transfer in progress (busy).
SCB_I2C_SSTAT_RD_OVFL	Master attempted to read more bytes than are in buffer.
SCB_I2C_SSTAT_RD_ERR	Slave captured error on the bus while read transfer.
SCB_I2C_SSTAT_WR_CMPLT	Slave write transfer complete.
SCB_I2C_SSTAT_WR_BUSY	Slave Write transfer in progress (busy).
SCB_I2C_SSTAT_WR_OVFL	Master attempted to write past end of buffer.
SCB_I2C_SSTAT_WR_ERR	Slave captured error on the bus while write transfer.

The following code example initializes the write buffer then waits for a transfer to complete. Once the transfer is complete, the data is then copied into a working array to handle the data. In many applications, the data does not have to be copied to a second location, but instead can be processed in the original buffer. You could create an almost identical read buffer example by replacing the write functions and constants with read functions and constants. Processing the data may mean new data is transferred into the slave buffer instead of out.

```
uint8 wrBuf[10u];
uint8 userArray[10u];
uint32 byteCnt;
uint32 status;

/* Initialize write buffer before call SCB_Start */
SCB_I2CSlaveInitWriteBuf((uint8 *) wrBuf, 10u);

/* Start I2C Slave operation */
SCB_Start();

/* The code below is not protected from the interruption. This might be required
to not cause buffer update by the following I2C write transaction while it is
handled.*/

/* Wait for I2C master to complete a write */

status = SCB_I2CSlaveStatus();
if(0u != (status & SCB_I2C_SSTAT_WR_CMPLT))
{
    if(0u == (status & SCB_I2C_SSTAT_WR_ERR))
    {
        byteCnt = SCB_I2CSlaveGetWriteBufSize();
        for(i=0; i < byteCnt; i++)
        {
            userArray[i] = wrBuf[i]; /* Copy data to local array */
        }
    }
    /* Clean-up status and buffer pointer */
    SCB_I2CSlaveClearWriteStatus();
    SCB_I2CSlaveClearWriteBuf();
```

```
}
```

Master/Multi-Master Operation

Master and Multi-Master operation are basically the same, with two exceptions. When operating in Multi-Master mode, the bus should always be checked to see if it is busy. Another master may already be communicating with another slave. In this case, the program must wait until the current operation is complete before issuing a start transaction. The program looks at the return value, which sets a busy status if another master has control of the bus.

The second difference is that, in Multi-Master mode, two masters can start at the exact same time. If this happens, one of the two masters loses arbitration. You must check for this condition after each byte is transferred. The component automatically checks for this condition and responds with an error if arbitration is lost.

There are two options when operating the I^2C master: manual and automatic. In automatic mode, a buffer is created to hold the entire transfer. In the case of a write operation, the buffer is prefilled with the data to be sent. If data is to be read from the slave, a buffer at least the size of the packet needs to be allocated. To write an array of bytes to a slave in automatic mode, use the following function.

```
uint32 SCB_I2CMasterWriteBuf(uint32 slaveAddress, uint8 * wrData, uint32 cnt,
uint32 mode)
```

The slaveAddress variable is a right-justified 7-bit slave address of 0 to 127. The component API automatically appends the write flag to the LSb of the address byte. The array of data to transfer is pointed to with the second parameter, wrData. The cnt parameter is the number of bytes to transfer. The last parameter, mode, determines how the transfer starts and stops. A transaction may begin with a restart instead of a start, or halt before the stop sequence. These options allow back-to-back transfers where the last transfer does not send a stop and the next transfer issues a restart instead of a start.

A read operation is almost identical to the write operation. The same parameters with the same constants are used.

```
uint32 SCB_I2CMasterReadBuf(uint32 slaveAddress, uint8 * rdData, uint32 cnt,
uint32 mode);
```

Both of these functions return status. See the status table for the SCB_I2CMasterStatus() function return value. Since the read and write transfers complete in the background during the I^2C interrupt code, the SCB_I2CMasterStatus() function can be used to determine when the transfer is complete. A code snippet that shows a typical write to a slave follows.

```
SCB_I2CMasterClearStatus(); /* Clear any previous status */
SCB_I2CMasterWriteBuf(8u, (uint8 *) wrData, 10u, SCB_I2C_MODE_COMPLETE_XFER);
for(;;)
{
   if(0u != (SCB_I2CMasterStatus() & SCB_I2C_MSTAT_WR_CMPLT))
   {

      /* Transfer complete. Check Master status to make sure that transfer
```

```
            completed without errors. */

         break;
      }
   }
```

The I²C master can also be operated manually. In this mode, each part of the write transaction is performed with individual commands.

```
status = SCB_I2CMasterSendStart(8u, SCB_I2C_WRITE_XFER_MODE);
if(SCB_I2C_MSTR_NO_ERROR == status)   /* Check if transfer completed without
errors */
{
   /* Send array of 5 bytes */
   for(i=0; i<5u; i++)
   {
      status = SCB_I2CMasterWriteByte(userArray[i]);
      if(SCB_I2C_MSTR_NO_ERROR != status)
      {
         break;
      }
   }
}
SCB_I2CMasterSendStop();      /* Send Stop */
```

A manual read transaction is similar to the write transaction except the last byte should be NAKed. The example below shows a typical manual read transaction.

```
status = SCB_I2CMasterSendStart(8u, SCB_I2C_READ_XFER_MODE);
if(SCB_I2C_MSTR_NO_ERROR == status)   /* Check if transfer completed without
errors */
{
   /* Read array of 5 bytes */
   for(i=0; i<5u; i++)
   {
      if(i < 4u)
      {
         userArray[i] = SCB_I2CMasterReadByte(SCB_I2C_ACK_DATA);
      }
      else
      {
         userArray[i] = SCB_I2CMasterReadByte(SCB_I2C_NAK_DATA);
      }
   }
}
SCB_I2CMasterSendStop();    /* Send Stop */
```

Multi-Master-Slave Mode Operation

Both Multi-Master and Slave are operational in this mode. The component may be addressed as a slave, but firmware may also initiate master mode transfers. In this mode, when a master loses

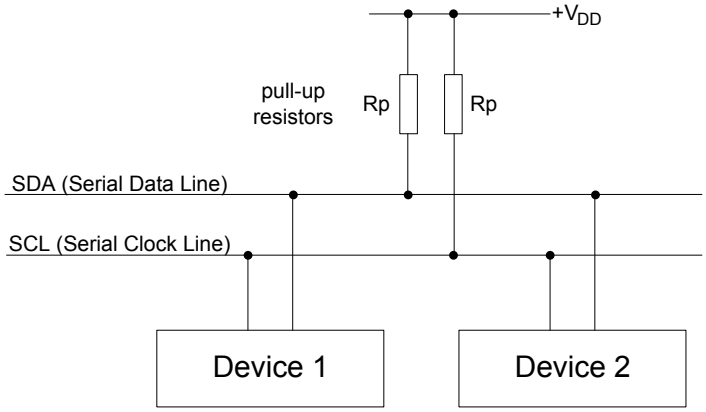
arbitration during an address byte, the slave hardware checks which winning master addressed it. In case of an address match, the slave becomes active.

For Master and Slave operation examples look at the Slave Operation and Master sections.

External Electrical Connections

As Figure 3 shows, the I^2C bus requires external pull-up resistors. The pull-up resistors (R_P) are determined by the supply voltage, clock speed, and bus capacitance. Make the minimum sink current for any device (master or slave) no less than 3 mA at V_{OLmax} = 0.4 V for the output stage. This limits the minimum pull-up resistor value for a 5-V system to about 1.5 kΩ. The maximum value for R_P depends upon the bus capacitance and clock speed. For a 5-V system with a bus capacitance of 150 pF, the pull-up resistors are no larger than 6 kΩ. For more information about sizing pull-up resistors and other physical bus specifications, refer to the *I^2C-Bus Specification*.

Figure 3. Connection of Devices to the I^2C Bus



Note Purchase of I^2C components from Cypress or one of its sublicensed Associated Companies, conveys a license under the Philips I^2C Patent Rights to use these components in an I^2C system, provided that the system conforms to the I^2C Standard Specification as defined by Philips. As of October 1, 2006, Philips Semiconductors has a new trade name - NXP Semiconductors.

Low power modes

The component in I^2C mode is able to be a wakeup source from Sleep and Deep Sleep low power modes.

Sleep mode is identical to Active from a peripheral point of view. No configuration changes are required in the component or code before entering/exiting sleep. Any communication intended to the slave causes an interrupt to occur and leads to wakeup. Any master activity that involves an interrupt to occur leads to wakeup.

Master modes (Master, Multi-Master) are not able to be a wakeup source from Deep Sleep. This capability is only available for slave modes (Slave, Multi-Master-Slave). The slave has to be configured properly to enable this functionality. The “Enable wakeup from Deep Sleep Mode” must be checked in the I2C configuration dialog. The SCB_Sleep() and SCB_Wakeup() functions must be called before/after entering/exiting Deep Sleep.

The wakeup event is a slave address match. The externally clocked logic performs address matching, when the address matches an interrupt request is generated, thus waking up the device. The slave stretches the SCL line until control is passed to its interrupt routine to ACK the address. The wakeup interrupt source is disabled in the interrupt handler or by SCB_Wakeup() after address match has occurred.

Before entering Deep Sleep, the ongoing transaction intended for the slave has to be completed. The following code is suggested:

```
CyGlobalIntDisable; /* Disables interrupts to lock the I2C slave state */

/* Checks if slave is busy */
status = (SCB_I2CSlaveStatus() & (SCB_I2C_SSTAT_RD_BUSY |
                                  SCB_I2C_SSTAT_WR_BUSY));

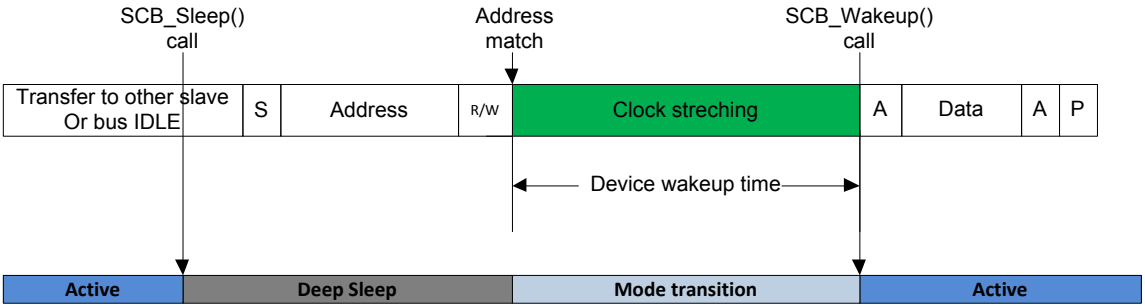
if(0u == status)
{
    /* Slave is not busy: enter Deep Sleep */
    SCB_Sleep();              /* Configure the slave to be wakeup source */
    CySysPmDeepSleep();
    CyGlobalIntEnable;        /* Enable interrupts to continue slave operation */
    SCB_Wakeup();             /* Configure the slave to active mode operation */
}
else
{
    CyGlobalIntEnable; /* Slave is busy: do not enter Deep Sleep */
}
```

For PSoC 4000, PSoC 4100 BLE, and PSoC 4200 BLE devices, the **component clock** must be disabled before calling SCB_Sleep(), and then enabled after calling SCB_Wakeup(); otherwise, the SCL will lock up after wakeup from Deep Sleep. Disabling and re-enabling the component clock is managed by the SCB_Sleep() and SCB_Wakeup() APIs when the Clock from terminal option is disabled. Otherwise, when the Clock from terminal option is enabled, the code provided above requires modification to enable and disable the clock source connected to the SCB component. Review the following modified code and highlighted in blue (ScbClock – the instance name of clock component connected to the SCB):

```
if(0u == status)
{
    SCB_Sleep();               /* Configure the slave to be wakeup source */
    ScbClock_Stop();           /* Disable the SCB clock */
    CySysPmDeepSleep();
    CyGlobalIntEnable;         /* Enable all interrupts to unlock I2C bus state */
    SCB_Wakeup();              /* Configure the slave to active mode operation */
    ScbClock _Start();         /* Enable the SCB clock */
}
```

For PSoC 4000, PSoC 4100 BLE, and PSoC 4200 BLE devices configured in Multi-Master-Slave mode only, in order for the component to be a wake-up source from deep sleep, the configuration of the component must be changed before deep sleep and after the part wakes from deep sleep. This configuration change occurs automatically in the SCB_Sleep() function, and any function that starts an I^2C master transaction. During this configuration change the component is disabled. After the configuration has been changed, it is re-enabled. During the time it is disabled, any incoming addresses to the slave will be NAKed; the external master will have to retry to access the slave.

Figure 4. Master transaction wakes up device on slave address match



Data rate configuration

For correct operation of I^2C mode the component must meet the data rate requirement of the connected I^2C bus. For master mode this means the master data rate cannot be faster than the slowest slave in the system. For slave mode this means the slave cannot be slower than the fastest master in the system.

For slave mode the frequency of the connected clock source is the only parameter used in determining the maximum data rate the slave can operate at. The connected clock is the clock that runs the SCB, not SCL. The frequency of the connected clock source must be fast enough to provide enough oversampling of the SCL and SDA signals to ensure that all I^2C specifications are meet. Table 1 provides the ranges of allowed clock frequencies for the standard I^2C data rates (Standard Mode, Fast Mode, and Fast-mode Plus). There are two ways to control the frequency of the connected clock for slave mode:

1. Use a clock component that is internal to the SCB component (this clock still uses clock divider resources). Based on the data rate set in the GUI the component asks PSoC Creator to create a clock with a frequency in the range provided in Table 1 on page 14.
2. Connect a user configurable clock to the SCB component. This option is enabled by checking the “Clock from terminal” control. In this mode it is the user’s responsibility to ensure the connected clock frequency is in the range provided in Table 1 on page 14. If the frequency is not in that range then proper I2C operation is no longer guaranteed.

Independent of the chosen method the component will display the actual data rate. This is the maximum data rate at which the slave can operate. If the system data rate is faster than the displayed actual data rate correct I^2C operation is no longer guaranteed.

For I^2C master mode the data rate is determined by the connected component clock, and the oversampling factor. These two factors are used to set the frequency of SCL, one SCL period is equal to the period of the connected clock multiplied by the oversampling factor. The oversampling factor is divided into low and high to enable independent control of the high and low phases of SCL. The low and high oversampling factor can be configured independently but their sum has to be equal to overall oversampling. In order to ensure that the master meets all I^2C specifications the connected clock frequency and oversampling factor must be within a specified range. Table 2 on page 14 provides a range of clock frequencies and oversampling factors for the standard I^2C data rates.

The component provides three methods to configure data rate:

1. Set the desired data rate and disable Manual oversampling control. This option uses a clock component that is internal to the SCB component (this clock still uses clock divider resources). Based on the data rate set in the GUI the component asks PSoC Creator to create a clock with a frequency in the range provided in Table 2 on page 14. When available clock frequency is returned the oversampling factors low and high are calculated to meet the set data rate, and the oversampling ranges provided in Table 2 on page 14.
2. Select desired data rate and enable Manual oversampling control. This option uses a clock component that is internal to the SCB component (this clock still uses clock divider resources). The component asks PSoC Creator to create a clock frequency equal to desired (Data rate * Oversampling). The oversampling is controlled by the user. This method is left to provide backward compatibility with previous versions of component.
3. Connect a user configurable clock to the SCB component. This option is enabled by checking the “Clock from terminal” control. The user still uses the GUI to configure the oversampling factor low and high. This method provides full control of the data rate configuration.

Independent of chosen method the component displays actual data rate for Master. This data rate might differ from the observed data rate on the bus due to the t_R and t_F time.

I^2C spec parameters calculation

The ranges provided in Table 2 on page 14 assume worst case conditions on the bus; often a bus will never experience worst case conditions. The following section describes how to calculate various I^2C parameters based on the connected clock frequency and oversampling factors. This information can be used to determine if the chosen connected clock frequency and oversampling factor meet I^2C specifications on your I^2C bus.

To find the frequency of the connected clock open the PSoC Creator Design-Wide Resources (DWR) file, and open the Clock Tab. If you used an internal clock look for a clock name that contains the component instance name with the suffix “_SCBCLK”. If you used the clock from terminal option, look for the name of your clock. The Nominal frequency has to be taken but it might differ from the Desired frequency due to design clock configuration.

Also the clock accuracy should be taken to account. The master device generates f_{SCL} with the accuracy of the provided clock source. Therefore when maximum data rate (for selected data

rate mode) is used the slave has to tolerate this inaccuracy otherwise the f_{SCL} has to be reduced or more accurate clock source provided.

The slave device is less sensitive to clock accuracy than the master as only when clock frequency is close to low or high limit (for the selected data rate mode) is provided the clock source accuracy has to be taken into account. The clock should not run too slow due negative deviation as well as too fast due to positive deviation. The slave has wide range of allowed clocks therefore selecting clock frequency in range (low limit + negative deviation) to (high limit – positive deviation) eliminates effect of the clock source accuracy.

For master modes Table 2 on page 14 contains the ranges of clock frequencies for the selected data rate. Keeping the clock frequency within these ranges ensures that $t_{VD;DAT}$ and $t_{VD;ACK}$ parameters from I^2C spec are met.

For Data rates of 0-400 kbps:

$$t_{VD;DAT} = (3/ f_{SCBCLK}) + t_{R_0\%_70\%} + 90 \text{ nsec}$$

For Data rates of 401-1000 kbps:

$$t_{VD;DAT} = (4/ f_{SCBCLK}) + t_{R_0\%_70\%}$$

Note $t_{R_0\%_70\%}$ is the rising time from 0% to 70% to be measured for specific I^2C bus.

Note The same equation applies for $t_{VD;ACK}$

Meeting $t_{VD;DAT}$ parameter ensures that $t_{SU;DAT}$ parameters is also met.

Meeting t_{LOW} and t_{HIGH} parameter ensures that t_{BUF}, $t_{SU;STA}$, $t_{HD;STA}$ and $t_{SU;STO}$ parameters are also met. To calculate t_{LOW} and t_{HIGH} the use following equation:

$$t_{LOW} = ((1/ f_{SCBCLK}) * \text{Oversampling factor Low}) – t_F$$

$$t_{HIGH} = ((1/ f_{SCBCLK}) * \text{Oversampling factor High}) – t_R$$

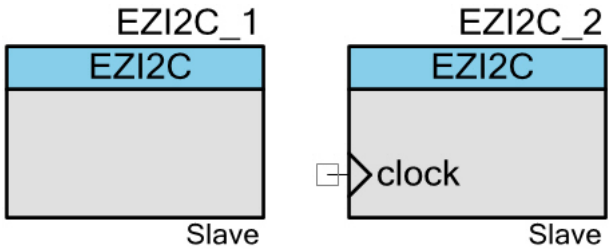
f_{SCBCLK} is a frequency of the connected component clock; Oversampling factor Low and High are parameters of the component.

Note The t_F and t_R values have to be measured for specific I^2C bus.

Note Calculated t_{HIGH} value might be less than observed on the bus due to clock synchronization in the device. The device resets its internal counter of t_{HIGH} when it detects a low level on SCL line while expecting high level. Therefore when t_R and internal device delay is greater than one component clock period – t_{HIGH} is extended. This causes the data rate to be less than expected.

For the slave mode the Table 1 on page 14 contains the ranges of clock frequencies for the selected data rate. Keep the clock frequency within these ranges to ensure that the slave meets all parameters of I^2C specification.

EZI2C [4]



The I²C bus is an industry standard, two-wire hardware interface developed by Philips®. The master initiates all communication on the I²C bus and supplies the clock for all slave devices. The EZI2C Slave implements an I²C register-based slave device. It is compatible [1] with I²C Standard-mode, Fast-mode, and Fast-mode Plus devices as defined in the NXP I2C-bus specification.

The EZI2C Slave is a unique implementation of an I²C slave in that all communication between the master and slave is handled in the ISR (Interrupt Service Routine) and requires no interaction with the main program flow. The interface appears as shared memory between the master and slave. Once the EZI2C_Start() function is called, there is little need to interact with the API.

Input/Output Connections

This section describes the various input and output connections for the SCB component. An asterisk (*) in the list of terminals indicates that the terminal may be hidden on the symbol under the conditions listed in the description of that terminal.

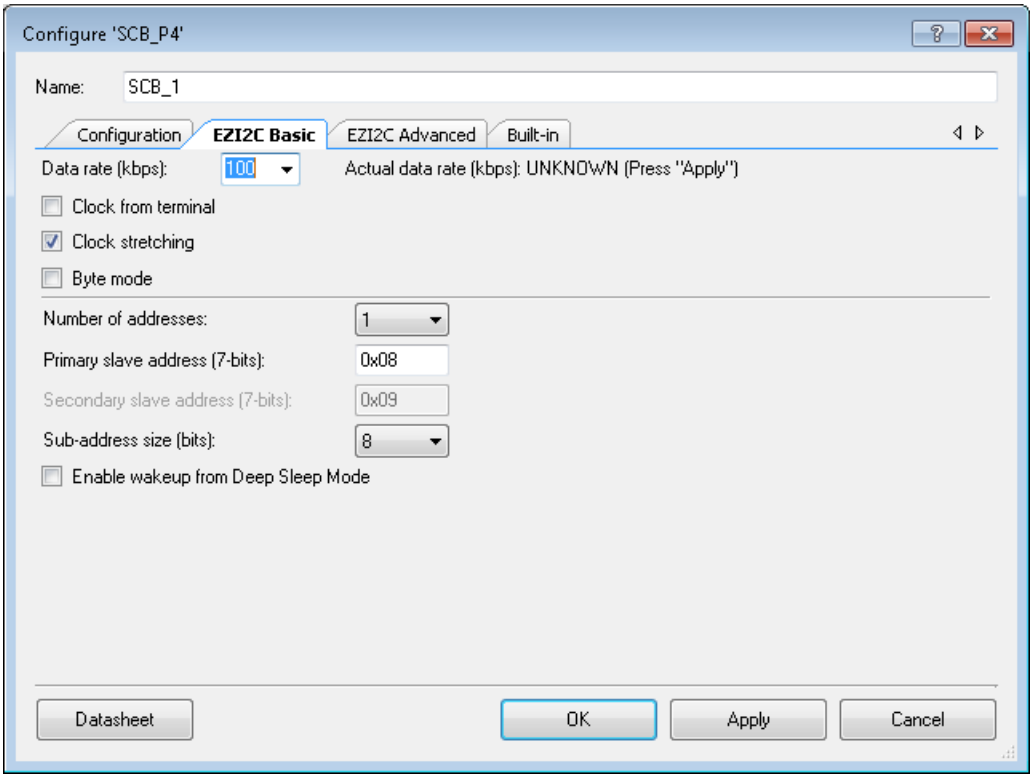
clock – Input*

Clock that operates this block. The presence of this terminal varies depending on the **Clock from terminal** parameter.

The interface-specific pins are buried inside the component because these pins use dedicated connections and are not routable as general purpose signals. See the *I/O System* section in the device *Technical Reference Manual (TRM)* for more information.

[4] This is a firmware implementation of the EZI2C protocol on top of I2C (non-EZ mode). All communication between the master and slave is handled in the ISR (Interrupt Service Routine). The data buffer has to be allocated in the RAM. The SCB component does not support EZI2C (EZ-mode), which uses a 32-bytes hardware buffer.

Basic EZI2C Parameters



The **EZI2C Basic** tab has the following parameters:

Data rate

This parameter is used to set the I^2C data rate value up to 1000 kbps (400 kbps for PSoC 4000 family); the actual data rate may differ from the selected data rate due to available clock frequency. The standard data rates are 100 (default), 400, and 1000 kbps. The **Data rate** is limited to a maximum of 400 kbps if the **Clock stretching** option is disabled. This parameter has no effect if the **Clock from terminal** parameter is enabled.

Actual data rate

Actual data rate displays the data rate at which the component will operate with current settings. The selected data rate could be different from actual data rate. The factors that affect the actual data rate calculation are: system clock and accuracy of the component clock (internal or external). When a change is made to any of the component parameters that affect the actual data rate, it becomes unknown. To calculate the new actual data rate press the Apply button.

Note The actual data rate always provides maximum value for the selected data rate mode (Standard-mode (100 kbps), Fast-mode (400 kbps), Fast-mode Plus (1000 kbps)).

Clock from terminal

This parameter allows choosing between an internally configured clock (by the component) or an externally configured clock (by the user) for the component operation.

When this control is enabled, the component does not control the data rate, but displays the actual data rate based on the user-connected clock source frequency. When this control is not enabled, the clock configuration is provided by the component. The clock source frequency is selected by the component based on the Data rate parameter. The table below shows the valid ranges for the component clock for each data rate. When using clock from terminal ensure that the external clock is within these ranges.

Table 4. EZI2C Slave clock frequency ranges

Parameter	Standard-mode (0-100 kbps)		Fast-mode (0-400 kbps)		Fast-mode Plus (0-1000 kbps)		Units
	Min	Max	Min	Max	Min	Max	
f_{SCB}	1.55	12.8	7.82	15.38	15.84	48.0	MHz

Note When the clock frequency is less than the lower limit of 1.55 MHz, an error is generated while building the project.

Note PSoC Creator is responsible for providing requested clock frequency (internal or external clock) based on current design clock configuration. When the requested clock frequency with requested tolerance cannot be created, a warning about the clock accuracy range is generated while building the project. This warning contains the actual clock frequency value created by PSoC Creator. To remove this warning you must either change the system clock, component settings or external clock to fit the clocking system requirements.

Clock stretching

This parameter applies clock stretching on the SCL line if the EZ I^2C slave is not ready to respond. Enabling this option ensures consistent slave operation for any EZ I^2C slave interrupt latency because the I^2C transaction is paused by clock stretching. Without the clock stretching option enabled, the design needs to service the EZ I^2C slave interrupt fast enough to provide correct slave operation.

Byte mode

This option is only applicable for PSoC 4100 BLE/PSoC 4200 BLE devices. It allows doubling the TX and RX FIFO depth from 8 to 16 bytes. Increasing the FIFO depth improves performance of EZ I^2C operation when clock stretching is enabled, as more bytes can be transmitted or received without software interaction. This option does not improve EZ I^2C operation when clock stretching is disabled; therefore, it is not available for this mode.

Number of addresses

This option determines whether 1 (default) or 2 independent I^2C slave addresses are recognized. If two addresses are recognized, then address detection will be performed in software. When the **Clock Stretching** option is **disabled**, the number of address choices is restricted to 1.

Primary slave address (7-bits)

This is an I^2C address that will be recognized by the slave as the primary address. This address is the 7-bit right-justified slave address and does not include the R/W bit. A slave address between 0x08 and 0x7F may be selected; the default is 0x08.

The value may be entered as decimal or hexadecimal; for hexadecimal numbers type ‘0x’ before the address.

Secondary slave address (7-bits)

This is an I^2C address that will be recognized by the slave as the secondary address. This address is the 7-bit right-justified slave address and does not include the R/W bit. A slave address between 0x08 and 0x7F may be selected; the default is 0x09. Refer to Preferable Secondary Address Choice.

The value may be entered as decimal or hexadecimal; for hexadecimal numbers type ‘0x’ before the address.

Sub-address size

This option determines what range of data can be accessed. You can select a sub-address of 8 bits (default) or 16 bits. If you use a sub-address size of 8 bits, the master can only access data offsets between 0 and 255. You may also select a sub-address size of 16 bits. That will allow the I^2C master to access data arrays of up to 65,535 bytes.

Enable wakeup from Deep Sleep Mode

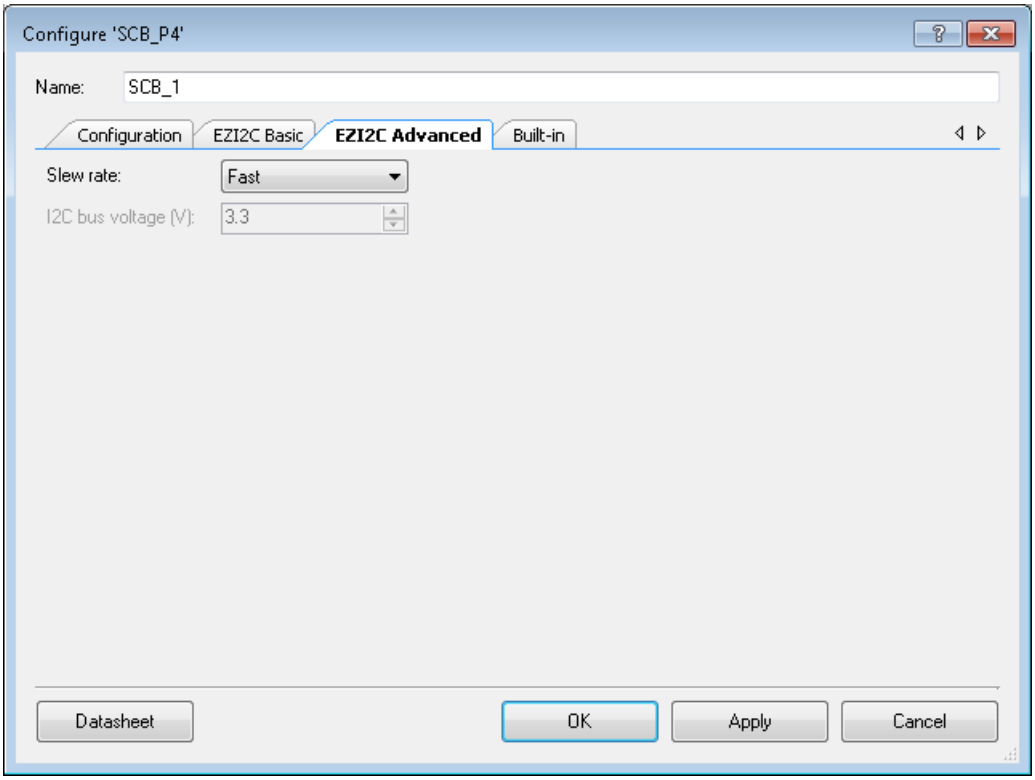
Use this option to enable the component to wake the system from Deep Sleep when a slave address match occurs.

Enabling this option adds the following restrictions (only for PSoC 4100/PSoC 4200 devices):

- Clock stretching must be enabled
- Slave address (7-bits) must be even (bit 0 equal zero)

Refer to the Low Power modes section in the EZI2C chapter of this document and *Power Management APIs* section of the *System Reference Guide* for more information.

Advanced EZI2C Parameters



The **EZI2C Advanced** tab contains the following parameters:

Slew rate

This option allows to control slew rate setting of the SCL and SDA pins. The slow slew rate increases the fall time on the lines, reducing EMI and coupling with neighboring signals. For devices supporting GPIO Over-Voltage Tolerance (GPIO_OVT) pins, I2C FM+ options should be used when I^2C data rate is greater than 400 kbps. This option also requires the I2C bus voltage to be defined. Refer to the *Device Datasheet* to determine which pins are GPIO_OVT capable. Default is fast.

Notes

- GPIO_OVT pins are fully compliant with the I^2C specification but the slew rate must be set appropriately:
  - **Slew rate** "Slow" for Standard mode (100 kbps) and Fast mode (400 kbps)
  - **Slew rate** "I2C FM+" for Fast mode plus (1 Mbps)

  Common GPIO pins are not completely compliant with the I^2C specification. Refer to the *Device Datasheet* for the details.
- **Slew rate** settings are applied to all pins of the associated port.

I2C bus voltage (V)

This option is only applicable for PSoC 4100 BLE/PSoC 4200 BLE devices. It specifies the voltage applied to the I^2C pull up resistors when Slew rate is I2C FM+. The voltage no less than applied to I^2C pulls up resistors must be provided by the V_{DDD} supply input, otherwise the I^2C pins cannot be placed. Valid values of V_{DDD} are determined by the settings in the Design-Wide Resources System Editor (in the *<project>.cydwr* file). This range check is performed outside this dialog; the results appear in the Notice List window if the check fails. Default is 3.3 V.

EZI2C APIs

Application Programming Interface (API) functions allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent section discusses each function in more detail.

By default, PSoC Creator assigns the instance name “SCB_1” to the first instance of a component in a given design. You can rename the instance to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “SCB”.

Function	Description
SCB_Start()	Starts the SCB component.
SCB_Init()	Initialize the SCB component according to defined parameters in the customizer.
SCB_Enable()	Enables the SCB component operation.
SCB_Stop()	Disable the SCB component.
SCB_Sleep()	Prepares the SCB component to enter Deep Sleep.
SCB_Wakeup()	Prepares component for Active mode operation after Deep Sleep.
SCB_EzI2CInit()	Configures the SCB component for operation in EZ I^2C mode. Only applicable when the component is in unconfigured mode.
SCB_EzI2CGetActivity()	Returns EZ I^2C slave status.
SCB_EzI2CSetAddress1()	Sets the primary EZ I^2C slave address.
SCB_EzI2CGetAddress1()	Returns the primary EZ I^2C slave address.
SCB_EzI2CSetBuffer1()	Sets up the data buffer to be exposed to the I^2C master on a primary slave address request.
SCB_EzI2CSetReadBoundaryBuffer1()	Sets the read only boundary of the data buffer to be exposed by I^2C master by the primary address request.
SCB_EzI2CSetAddress2()	Sets the secondary EZ I^2C slave address.
SCB_EzI2CGetAddress2()	Returns the secondary EZ I^2C slave address.

Function	Description
SCB_EzI2CSetBuffer2()	Sets up the data buffer to be exposed to the I^2C master on a secondary slave address request.
SCB_EzI2CSetReadBoundaryBuffer2()	Sets the read boundary of the data buffer to be exposed by I^2C master by the secondary address request.

void SCB_Start(void)

Description: Invokes SCB_Init() and SCB_Enable(). After this function call the component is enabled and ready for operation. This is the preferred method to begin component operation.

When configuration is set to “Unconfigured SCB”, the component must first be initialized to operate in one of the following configurations: I^2C, SPI, UART or EZ I^2C. Otherwise this function does not enable component.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Init(void)

Description: Initializes the SCB component to operate in one of the selected configurations: I^2C, SPI, UART or EZ I^2C.

When configuration set to “Unconfigured SCB”, this function does not do any initialization. Use mode-specific initialization APIs instead: SCB_I2CInit, SCB_SpiInit, SCB_UartInit or SCB_EzI2CInit.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Enable(void)

Description: Enables SCB component operation: activates the hardware and internal interrupt.

For I^2C and EZ I^2C modes the interrupt is internal and mandatory for operation. For SPI and UART modes the interrupt can be configured as none, internal or external.

The SCB configuration should be not changed when the component is enabled. Any configuration changes should be made after disabling the component.

When configuration is set to “Unconfigured SCB”, the component must first be initialized to operate in one of the following configurations: I^2C, SPI, UART or EZ I^2C, Using the mode-specific initialization API. Otherwise this function does not enable the component.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Stop(void)

Description: Disables the SCB component: disable the hardware and internal interrupt. Refer to the function SCB_Enable() for the interrupt configuration details.

This function disables the SCB component without checking to see if communication is in progress. Before calling this function it may be necessary to check the status of communication to make sure communication is complete. If this is not done then communication could be stopped mid byte and corrupted data could result.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Sleep(void)

Description: Prepares component to enter Deep Sleep.

The “Enable wakeup from Deep Sleep Mode” selection has an influence on this function implementation:

- Checked: configures the component to be wakeup source from Deep Sleep.
- Unchecked: stores the current component state (enabled or disabled) and disables the component. See SCB_Stop() function for details about component disabling.

Call the SCB_Sleep() function before calling the CyPmSysDeepSleep() function. Refer to the PSoC Creator *System Reference Guide* for more information about power management functions.

This function should not be called before entering Sleep.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Wakeup(void)

Description: Prepares component to Active mode operation after Deep Sleep.

The “Enable wakeup from Deep Sleep Mode” selection influences this function implementation:

- Checked: restores the component Active mode configuration.
- Unchecked: enables the component if it was enabled before enter Deep Sleep.

This function should not be called after exiting Sleep.

Parameters: None

Return Value: None

Side Effects: Calling the SCB_Wakeup() function without first calling the SCB_Sleep() function may produce unexpected behavior.

void SCB_EzI2CInit(SCB_EZI2C_INIT_STRUCT *config)

Description: Configures the SCB for EZ I²C operation.

This function is **intended specifically** to be used when the SCB configuration is set to "Unconfigured SCB" in the customizer. After initializing the SCB in EZ I²C mode, the component can be enabled using the SCB_Start() or SCB_Enable() function.

This function uses a pointer to a structure that provides the configuration settings. This structure contains the same information that would otherwise be provided by the customizer settings.

Parameters: config: pointer to a structure that contains the list of fields. These fields match the selections available in the customizer. Refer to the customizer for further description of the settings.

Field	Description
uint32 enableClockStretch	0 – disable 1 – enable When enabled the SCL is stretched as required for proper operation.
uint32 enableMedianFilter	This field is left for compatibility and its value is ignored. Median filter is disabled for EZI2C mode.
uint32 numberOfAddresses	Number of supported addresses: SCB_EZI2C_ONE_ADDRESS SCB_EZI2C_TWO_ADDRESSES
uint32 primarySlaveAddr	Primary 7-bit slave address.
uint32 secondarySlaveAddr	Secondary 7-bit slave address.
uint32 subAddrSize	Size of sub-address: SCB_EZI2C_SUB_ADDR8_BITS SCB_EZI2C_SUB_ADDR16_BITS
uint32 enableWake	0 – disable 1 – enable When enabled the matching address generates a wakeup request.
uint8 enableByteMode	Ignored for all devices other than PSoC 4100 BLE/PSoC 4200 BLE. 0 – disable 1 – enable When enabled the TX and RX FIFO depth is 16 bytes.

Return Value: None

Side Effects: None

uint32 SCB_EzI2CGetActivity(void)

Description: Returns EZ I²C slave status.

The read, write and error status flags reset to zero after this function call.

The busy status flag is cleared when the transaction intended for the EZ I²C slave completes.

This function disables EZ I²C slave interrupt during execution to operate correctly. This may have significant impact to correctness of EZ I²C slave operation when the clock stretching option is disabled. The amount of time that the interrupt is disabled should be less than the maximum EZ I²C slave interrupt latency. Refer to section Clock Stretching Disable for more details.

Parameters: None

Return Value: uint32: Current status of EZ I²C slave.

This status incorporates a number of status constants. Each constant is a bit field value. The value returned may have multiple bits set to indicate the status of the transfer.

Slave Status Constants	Description
SCB_EZI2C_STATUS_READ1	Read transfer complete. The transfer used the primary slave address. The error condition status bit must be checked to ensure that read transfer was completed successfully.
SCB_EZI2C_STATUS_WRITE1	Write transfer complete. The buffer content was modified. The transfer used the primary slave address. The error condition status bit must be checked to ensure that write transfer was completed successfully.
SCB_EZI2C_STATUS_READ2	Read transfer complete. The transfer used the secondary slave address. The error condition status bit must be checked to ensure that read transfer was completed successfully.
SCB_EZI2C_STATUS_WRITE2	Write transfer complete. The buffer content was modified. The transfer used the secondary slave address The error condition status bit must be checked to ensure that write transfer was completed successfully.
SCB_EZI2C_STATUS_BUSY	A transfer intended for the primary or secondary address is in progress. The status bit is set after an address match and cleared on a Stop or ReStart condition.
SCB_EZI2C_STATUS_ERR	An error occurred during a transfer intended for the primary or secondary slave address. The sources of error are: misplaced Start or Stop condition or lost arbitration while slave drives SDA. The write buffer may contain invalid byte or part of the transaction when SCB_EZI2C_STATUS_ERR and SCB_EZI2C_STATUS_WRITE1/2 is set. It is recommended to discard buffer content in this case.

Side Effects: None

void SCB_EzI2CSetAddress1(uint32 address)

Description: Sets the primary EZ I^2C slave address.

Parameters: uint32 address: primary I^2C slave address.

This address is the 7-bit right-justified slave address and does not include the R/W bit.

The address value is not checked to see if it violates the I^2C spec. The preferred addresses are in the range between 8 and 120 (0x08 to 0x78).

Return Value: None

Side Effects: None

uint32 SCB_EzI2CGetAddress1(void)

Description: Returns primary the EZ I^2C slave address.

This address is the 7-bit right-justified slave address and does not include the R/W bit.

Parameters: None

Return Value: uint32: Primary I^2C slave address.

Side Effects: None

void SCB_EzI2CSetBuffer1(uint32 bufSize, uint32 rwBoundary, volatile uint8 * buffer)

Description: Sets up the data buffer to be exposed to the I^2C master on a primary slave address request.

Parameters: uint32 bufSize: Size of the data buffer in bytes.

uint32 rwBoundary: number of data bytes starting from the beginning of the buffer with read and write access. Data bytes located at offset rwBoundary or greater are read only.

This value must be less than or equal to the buffer size.

uint8* buffer: Pointer to the data buffer.

Return Value: None

Side Effects: Calling this function in the middle of a transaction intended for the primary slave address leads to unexpected behavior.

void SCB_EzI2CSetReadBoundaryBuffer1(uint32 rwBoundary)

Description: Sets the read only boundary in the data buffer to be exposed to the I^2C master on a primary slave address request.

Parameters: uint32 rwBoundary: number of data bytes starting from the beginning of the buffer with read and write access. Data bytes located at offset rwBoundary or greater are read only.

This value must be less than or equal to the buffer size.

Return Value: None

Side Effects: Calling this function in the middle of a transaction intended for the primary slave address leads to unexpected behavior.

void SCB_EzI2CSetAddress2(uint32 address)

Description: Sets the secondary EZ I^2C slave address.

Parameters: uint32 address: secondary I^2C slave address.

This address is the 7-bit right-justified slave address and does not include the R/W bit.

The address value is not checked to see if it violates the I2C spec. The preferred addresses are in the range between 8 and 120 (0x08 to 0x78).

Return Value: None

Side Effects: None

uint32 SCB_EzI2CGetAddress2(void)

Description: Returns the secondary EZ I^2C slave address.

This address is the 7-bit right-justified slave address and does not include the R/W bit.

Parameters: None

Return Value: uint32: Secondary I^2C slave address.

Side Effects: None

void SCB_EzI2CSetBuffer2(uint32 bufSize, uint32 rwBoundary, volatile uint8 * buffer)

Description: Sets up the data buffer to be exposed to the I^2C master on a secondary slave address request.

Parameters: uint32 bufSize: Size of the data buffer in bytes.

uint32 rwBoundary: number of data bytes starting from the beginning of the buffer with read and write access. Data bytes located at offset rwBoundary or greater are read only.

This value must be less than or equal to the buffer size.

uint8* buffer: Pointer to the data buffer.

Return Value: None

Side Effects: Calling this function in the middle of a transaction intended for the secondary slave address leads to unexpected behavior.

void SCB_EzI2CSetReadBoundaryBuffer2(uint32 rwBoundary)

Description: Sets the read only boundary in the data buffer to be exposed to the I^2C master on a secondary address request.

Parameters: uint32 rwBoundary: number of data bytes starting from the beginning of the buffer with read and write access. Data bytes located at offset rwBoundary or greater are read only.

This value must be less than or equal to the buffer size.

Return Value: None

Side Effects: Calling this function in the middle of a transaction intended to the secondary slave address leads to unexpected behavior.

Global Variables

Knowledge of these variables is not required for normal operations.

Variable	Description
SCB_initVar	SCB_initVar indicates whether the SCB component has been initialized. The variable is initialized to 0 and set to 1 the first time SCB_Start() is called. This allows the component to restart without reinitialization after the first call to the SCB_Start() routine. If reinitialization of the component is required, then the SCB_Init() function can be called before the SCB_Start() or SCB_Enable() function.

EZI2C Functional Description

This component supports an I²C slave device with one or two I²C addresses. Either address may access a memory buffer defined in RAM or flash data space. Flash memory buffers are read only, while RAM buffers may be read/write. The addresses are right justified.

When using this component, you must enable global interrupts because the I²C hardware is interrupt driven. Even though this component requires interrupts, you do not need to add any code to the ISR (Interrupt Service Routine). The component services all interrupts (data transfers) independently from your code. The memory buffers allocated for this interface look like simple dual-port memory between your application and the I²C master.

If required, you can create a higher-level interface between a master and slave by defining semaphores and command locations in the data structure.

Memory Interface

To an I²C master the interface looks very similar to a common I²C EEPROM. The EZ I²C buffer can be configured as a variable, array, or structure but it is preferable to use an array. The buffer acts as a shared memory interface between your program and an I²C master through the I²C bus. The component permits read and write I²C master access to the specified buffer memory and prevents any access outside the buffer or write access into a read only region.

For example, the buffer for the primary slave address is configured using the code below. The buffer elements from 4 to 9 are read only.

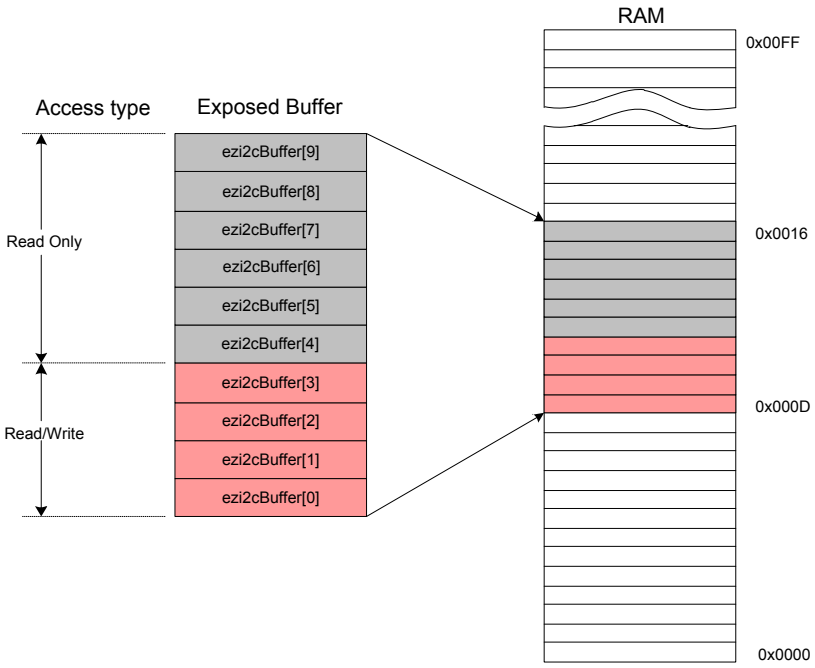
```
#define BUFFER_SIZE             (0x0Au)
#define BUFFER_RW_BOUNDARY      (0x04u)

uint8 ezi2cBuffer[BUFFER_SIZE];

SCB_EzI2CSetBuffer1(BUFFER_SIZE, BUFFER_RW_BOUNDARY, ezi2cBuffer);
```

The buffer ezi2cBuffer is allocated in memory as shown in Figure 5.

Figure 5. EZ I²C buffer exposed to an I²C master



To configure the whole buffer for read and write access, the buffer size and read/write boundary need to use the same value. For example:

```
SCB_EzI2CSetBuffer1(BUFFER_SIZE, BUFFER_SIZE, ezi2cBuffer);
```

Handling endianness

The EZ I²C buffer can be set up as a variable. A variable with a size of more than one byte will require knowledge of endianness (little-endian or big-endian). The endianness will determine the byte order on the I²C bus. It is the I²C master's responsibility to handle byte ordering properly.

```
uint16 ezi2cBuffer = 0xAABB;
#define BUFFER_SIZE (2u)

SCB_EzI2CSetBuffer1(BUFFER_SIZE, BUFFER_SIZE, (uint8 *) &ezi2cBuffer);
```

All PSoC 4 devices are little-endian devices, so the master will read these two bytes in order as: 0xBB 0xAA.

Handling structures

The EZ I²C buffer can be set up as structure. The compiler lays out structures in memory and may add extra bytes. This is called byte padding. The compiler will add these bytes to align the fields of the structure to match the requirements of the Cortex-M0. This processor does not support unaligned access to multi-byte fields. When using a structure, the application must take

this alignment into account. If fields need to be packed, then a byte array should be used instead of a structure.

Handling a status byte

To define a higher level protocol, a status byte placed inside the EZ I²C buffer may be required. This status byte would be modified by the I²C master, but the compiler is not aware that an interrupt routine may modify this buffer. This can result in the compiler optimizing a while loop that tests for a change in the status byte. The keyword volatile must be used to inform the compiler that the status byte might change state even though no statements in the program appear to change it.

Code example:

```
#define BUFFER_SIZE       (0x0Au)
#define STATUS_BYTE_POS   (0u)

volatile uint8 ezi2cBuffer[BUFFER_SIZE];

SCB_EzI2CSetBuffer1(BUFFER_SIZE, BUFFER_SIZE, ezi2cBuffer);

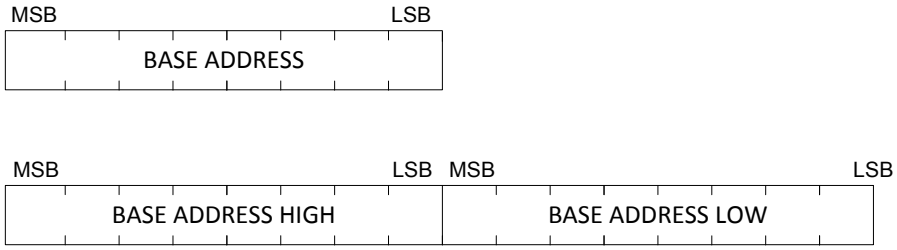
ezi2cBuffer[STATUS_BYTE_POS] = 0x01u;
while(0x01u == ezi2cBuffer[STATUS_BYTE_POS])
{
    /* Wait for status byte to be changed by the master */
}
```

Interface as Seen by an External Master

The EZ I²C slave component supports basic read and write operations for the read/write region and read operations for the read-only region. The two I²C address interfaces contain separate data buffers that are addressed with separate base addresses. The base address is an index within the EZ I²C buffer, its range is 0 to buffer size - 1. The base address comes first, followed by the data bytes. The base address size depends on **Sub-address size** parameter: one byte (Sub-address size = 8bits) or two bytes (Sub-address size = 16bits). The sub-address size of 8 bits is used to access buffers up to 256 bytes and sub-address size of 16 bits is used for buffers up to 65535 bytes. In the case of a two byte address, the first byte is the high byte and the second is the low byte of the 16-bit value Figure 6.

For example, the desired base address to access is 0x0201: high byte is 0x02 and low is 0x01.

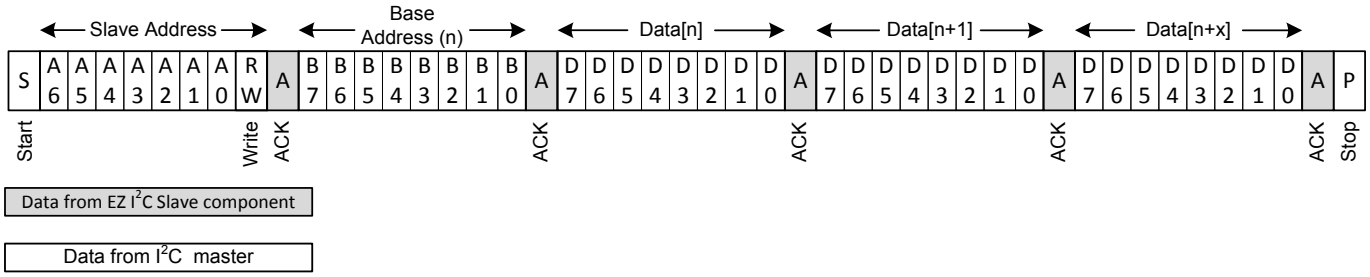
Figure 6. The 8-bit and 16-bit Sub-Address Size



For write operations, a base address is always provided and is one or two bytes depending on the configuration. This base address is retained and will be used for later read operations. Following the base address is a sequence of bytes that are written into the buffer starting from the base address location. The buffer index is incremented for each written byte, but this does not affect the base address, which is retained. The length of a write operation is **limited** by the maximum buffer read/write region size. The EZ I^{2}C slave behaves differently on the I^{2}C bus when a master attempts to write outside the read/write region or past the end of the buffer depending on the setting for Clock Stretching:

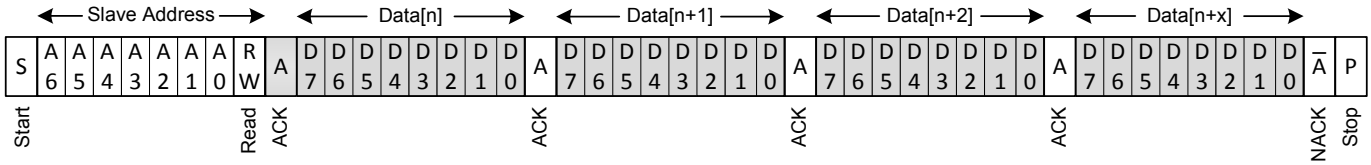
- Enabled: the byte is NAKed by the slave and the master has to stop the current transaction. The NAKed byte is discarded by the slave.
- Disabled: all written bytes are ACKed by the slave, but these bytes are discarded.

Figure 7. I^{2}C Master writes X bytes to the EZ I^{2}C Slave buffer



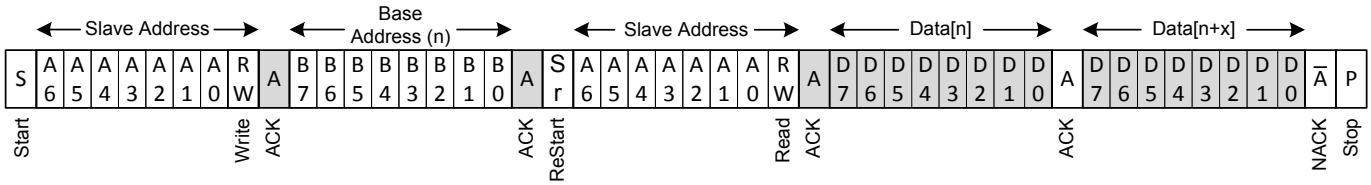
A read operation **always starts** from the base address set by the most recent write operation. The buffer index is incremented for each read byte. Two sequential read operations start from the same base address no matter how many bytes were read. The length of a read operation is **not limited** by the maximum size of the data buffer. The EZ I^{2}C slave returns 0xFF bytes if the read operation passes the end of the buffer.

Figure 8. I²C Master reads X bytes from the EZ I²C Slave buffer



Typically, a read operation requires the base address to be updated before starting the read. In this case, the write and read operations need to be combined together. The I²C master may use ReStart or Stop/Start conditions to combine the operations. The write operation only sets the base address and the following read operation will start reading from the new base address. In cases where the base address remains the same, there is no need for a write operation to be performed.

Figure 9. I²C Master sets the base address and reads X bytes from the EZ I²C Slave buffer



Detailed descriptions of the I²C bus and its implementation are available in the complete I²C specification on the NXP website, and by referring to the device datasheet.

Data Coherency

Although a data buffer may include a data structure larger than a single byte, a Master read or write operation consists of multiple single-byte operations. This can cause a data coherency problem, because there is no mechanism to guarantee that a multi-byte read or write will be synchronized on both sides of the interface (Master and Slave). For example, consider a buffer that contains a single two-byte integer. While the master is reading the two-byte integer one byte at a time, the slave may have updated the entire integer between the time the master read the first byte of the integer (LSB) and was about to read the second byte (MSB). The data read by the master may be invalid, since the LSB was read from the original data and the MSB was read from the updated value.

You must provide a mechanism on the master, slave, or both that guarantees that updates from the master or slave do not occur while the other side is reading or writing the data. The SCB_EzI2CGetActivity() function can be used to develop an application-specific mechanism.

Note The buffer setup APIs are not interrupt protected and must be called when the component is disabled or the slave is not busy.

Clock Stretching

Clock stretching pauses a transaction by holding the SCL line low. The transaction cannot continue until the SCL line is released allowing the signal to go high again. The support of clock stretching is an optional feature of the I^2C spec. For that reason, the EZ I^2C slave provides an option to enable or disable this feature.

Clock Stretching Enable

Enabling the clock stretching option makes is possible for the slave to insert a pause into the transaction at the byte level. This allows for consistent EZ I^2C slave operation for any slave interrupt latency. The drawback is that the master has to support clock stretching as well.

Clock Stretching Disable

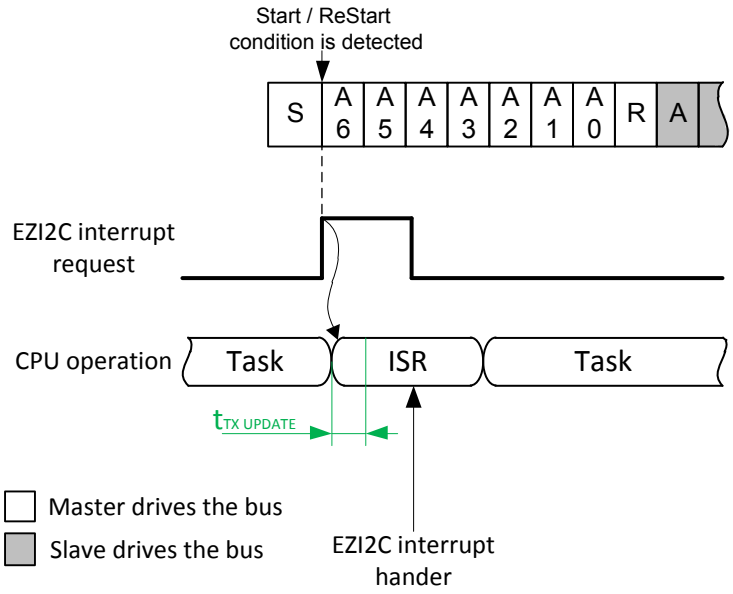
Disabling clock stretching configures the EZ I^2C slave to operate with an optimized interrupt service routine. This allows the EZ I^2C slave to operate without clock stretching. Despite the optimization, the slave interrupt still must be serviced fast enough. The maximum time that the slave interrupt service can be delayed is defined as the maximum EZ I^2C slave interrupt latency. A design that does not satisfy the required maximum EZ I^2C slave interrupt latency will cause erroneous slave behavior. It is recommended to enable clock stretching if the design cannot satisfy the required maximum EZ I^2C slave interrupt latency. When selecting the clock stretching disable option, refer also to the following:

- Maximum slave interrupt latency
- Transactions chained with ReStart
- Slave busy management

Maximum slave interrupt latency

All master transactions begin with a Start condition. The slave hardware detects this condition and generates an interrupt request, which starts slave operation (Figure 10). The ReStart condition has the same effect on the slave as Start condition, except previous transaction completion flags have to be set; service of the ReStart condition has greater priority.

Figure 10. EZ I²C slave starts operation



The time between starting the slave interrupt handler to the moment when the TX FIFO has been updated with the first byte is referred as $t_{TX\ UPDATE}$[5] [6] (Figure 10). The TX FIFO update consists of clearing the TX FIFO and writing a byte from the slave buffer into the TX FIFO. The TX FIFO update must be completed before the master starts reading the first data byte. Otherwise, a number of issues can occur, including: reading old the TX FIFO content, clock stretching when the TX FIFO is cleared [7], or reading a partial byte due to the TX FIFO clear in the middle of the byte transfer.

The constraint applied to the TX FIFO update during the master read transaction causes the maximum slave interrupt latency to be defined as maximum delay, which can be inserted from the Start condition detection by the slave hardware, to the start of the execution of the slave interrupt handler (Figure 11 on page 70).

5 This time depends on design settings such as CPU clock, compiler, optimization, etc.

6 This time does not include interrupt latency of the Cortext-M0 processor.

7 The slave hardware stretches the clock when TX FIFO is empty.

Therefore, the maximum interrupt latency must be less than the master address byte transmit time ($t_{ADDRESS}$) plus slave ACK bit transmit time (t_{ACK}). But taking to account TX FIFO update constraint, the maximum interrupt latency must be less than:

$$t_{MAX\ LATENCY} = (t_{ADDRESS} + t_{ACK}) - t_{TX\ UPDATE} = (8bits / f_{SCL} + 1bit / f_{SCL}) - t_{TX\ UPDATE} = 9 / f_{SCL} - t_{TX\ UPDATE}$$

For example I^2C data rate of 100 kbps, the maximum interrupt latency must be less than:

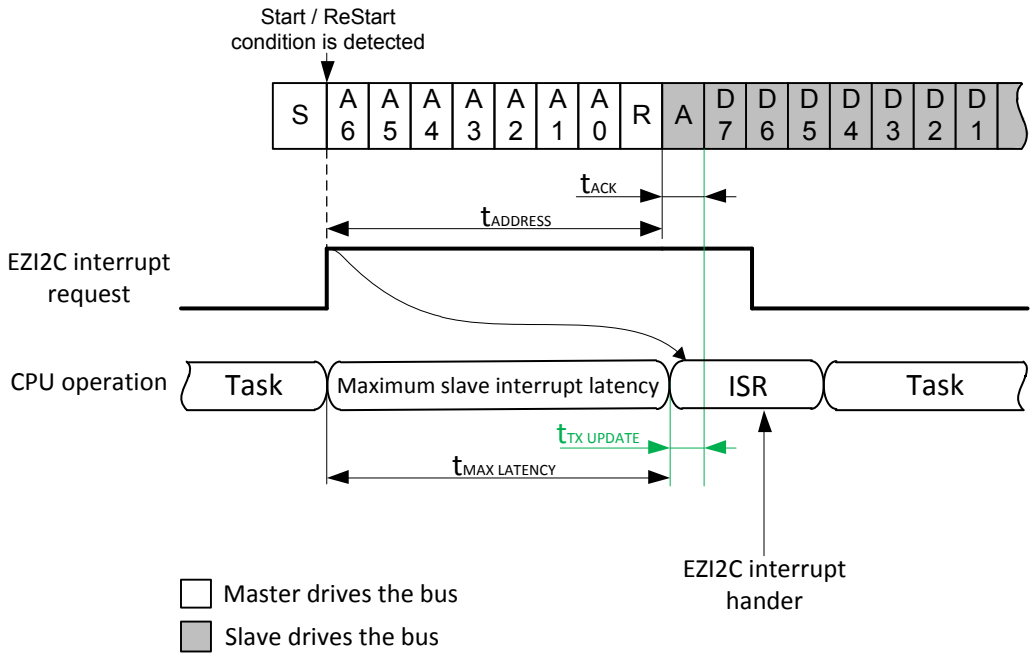
$$t_{MAX\ LATENCY} = 9 / f_{SCL} - t_{TX\ UPDATE} = 90\ uS - t_{TX\ UPDATE}$$

The number of CPU cycles to put the first byte into TX FIFO is calculated in the EZ I^2C interrupt. This number is equal to 71 cycles (mode Release, Compiler GCC, optimization Size). Taking into account that the maximum interrupt latency of the Cortext-M0 processor is 16 cycles; the number of cycles is increased to 87.

With the assumption that clock configuration of the design is internal IMO and HFCLK = SYSCLK = 24 MHz, the $t_{MAX\ LATENCY}$ is calculated for the I^2C data rate of 100 kbps. The accuracy of the internal IMO is +/-2%; therefore, the number of CPU cycles is increased by 2% and equal to ~89 cycles. The $t_{TX\ UPDATE}$ = 89 / SYSCLK = 3.71uS. Referring to the above equation, the maximum interrupt latency is equal to:

$$t_{MAX\ LATENCY} = 9 / f_{SCL} - t_{TX\ UPDATE} = 90\ uS - t_{TX\ UPDATE} = 90\ uS - 3.71\ uS = 86.29\ uS$$

Figure 11. EZ I2C slave maximum interrupt latency



Design recommendations:

1. Use the highest possible SYSCLK frequency, as it runs the CPU to reduce execution time of the EZ I^2C slave interrupt.

2. Use optimization options of the compiler as it reduces the number of instructions to execute in the EZ I^2C slave interrupt.
3. Set the EZ I^2C interrupt priority to be the highest in the design. If there are other interrupts with the same priority, make sure that their execution time is less than EZ I^2C maximum interrupt latency.
4. Calculate the duration of the each critical section in the design and compare with the EZ I^2C maximum interrupt latency to make sure that design meets criteria.

Transactions chained with ReStart

A common use case is for the master to write the base address, and then using a repeated start (no Stop) read data from the slave starting at the base address (Figure 12 on page 72). The base address (or data byte) written by the master is received into the RX FIFO and must be serviced by the slave interrupt handler (Figure 12, case 1). The service of the RX FIFO has greater priority than the ReStart condition service because the base address may have been updated by the master write transaction, and it is used for the TX FIFO update. The time spent to service the RX FIFO might affect the service of the ReStart condition. If this is the case (Figure 12, case 2), the maximum interrupt latency is reduced by the time it takes to service the RX FIFO after the ReStart condition is detected:

$$t_{MAX\ LATENCY} = 9 / f_{SCL} – (t_{RX\ DELAY} + t_{TX\ UPDATE})$$

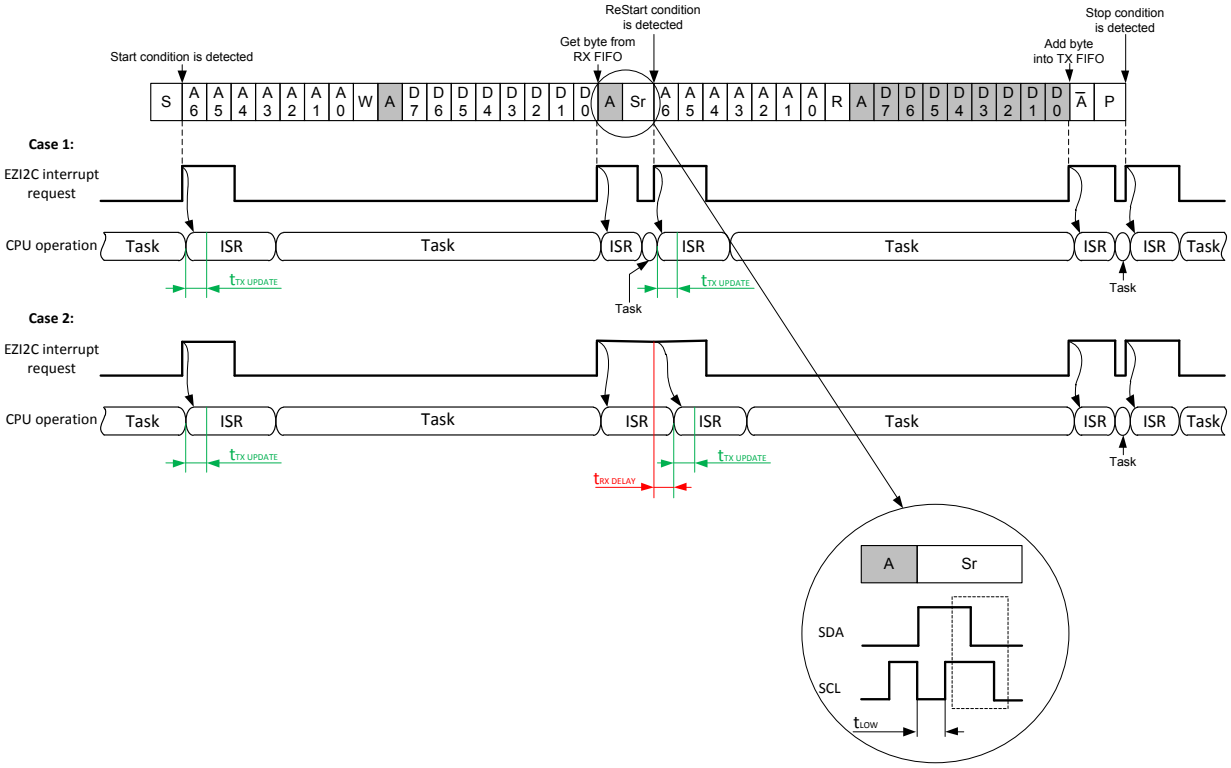
The RX FIFO service will affect $t_{MAX\ LATENCY}$ when it takes longer than (1bit / f_{SCL} + t_{LOW} + $t_{SU;STA}$), (where 1bit / f_{SCL} is duration of ACK condition generation, t_{LOW} and $t_{SU;STA}$ minimum values for the selected bus speed mode). For data rate 100 kbps this time equal to: 10 uS + 4.7 uS + 4.7 uS = 19.4 uS.

The longest RX FIFO service path in the EZ I^2C interrupt is consumed by the handling of base address written by the master ($t_{RX\ SERVICE}$). It consumes 100 CPU cycles (mode Release, Compiler GCC, optimization Size). Taking into account that the maximum interrupt latency of the Cortex-M0 processor is 16 cycles; the number of cycles is increased to 116.

With the assumption that clock configuration of the design is internal IMO and HFCLK = SYSCLK = 24 MHz, the $t_{RX\ SERVICE}$ is calculated. The accuracy of the internal IMO is +/-2%; therefore, the number of CPU cycles is increased by 2% and equal to ~118 cycles. The $t_{RX\ SERVICE}$ = 118 / SYSCLK = 4.92 uS which is less than 19.4 uS, therefore $t_{RX\ DELAY}$ = 0 uS.

The master ReStart timings must be examined; the I^2C spec provides minimum values. Some masters before ReStart generation extend t_{LOW} to prepare for the next transaction, but it is device specific. If there is possibility to control this time, the RX FIFO service effect on the maximum interrupt latency can be eliminated by increasing t_{LOW} before ReStart until $t_{RX\ DELAY}$ is equal to zero.

Figure 12. Master set base address and read data



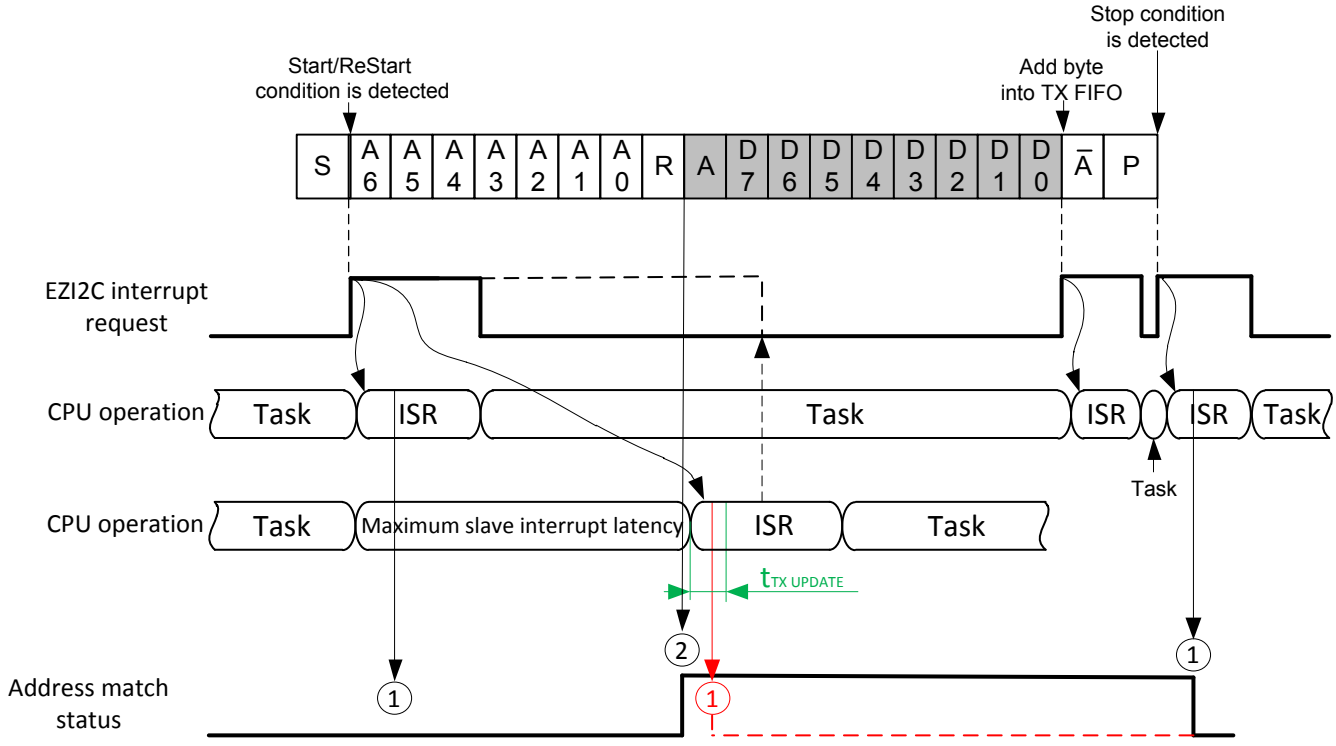
Slave busy management

The SCB_EzI2CGetActivity() API and SCB_Sleep() API (only for PSoC 4000 devices) use address match status to track slave busy status. This event is triggered by hardware on the rising edge of 8th SCL within the address byte for PSoC 4100/PSoC 4200 devices and on falling edge of 8th SCL for PSoC 4000 devices (Figure 13, black circle 2).

To be used as slave busy status, the address match is cleared by firmware on the Start / ReStart or Stop condition (Figure 13, black circle 1). If Start / ReStart interrupt service is delayed for maximum interrupt latency, the address match status is cleared too early (Figure 13, red circle 1).

This causes incorrect slave busy reporting. For correct slave busy status reporting, the maximum interrupt latency must be reduced to 1.5 bit / f_{SCL} for PSoC 4100/PSoC 4200 devices and to 1 bit / f_{SCL} for PSoC 4000 devices. As an alternative, the SCB hardware bus busy status can be used to manage bus activity. See the *SCB_I2C_STATUS* register bit *SCB_BUS_BUSY* description in *Technical Reference Manual (TRM)* for more information.

Figure 13. Slave busy management



(1) Address match status is cleared by firmware

(2) Address match status is set by hardware

External Electrical Connections

Refer to the External Electrical Connections section for I^2C.

Preferable Secondary Address Choice

The hardware address-match-logic uses address bit masking to support both addresses. The address mask defines which bits in the address are treated as non-significant while performing an address match. One non-significant bit results in two matching addresses; two bits will match 4 and so on. Due to this reason, it is preferable to select a secondary address that is different from the primary by one bit. The address mask in this case makes one bit non-significant. If the two addresses differ by more than a single bit, then the extra addresses that will pass the hardware match and will rely on firmware address matching to generate a NAK.

For example:

- Primary address = 0x24 and secondary address = 0x34, only one bit differs. Only the two addresses are treated as matching by the hardware.
- Primary address = 0x24 and secondary address = 0x30, two bits differ. Four addresses are treated as matching by the hardware: **0x24**, 0x34, 0x20 and **0x30**. Firmware is required to ACK only the primary and secondary addresses 0x24 and 0x30 and NAK all others 0x20 and 0x34.

Low power modes

The component in EZ I^2C mode is able to be a wakeup source from Sleep and Deep Sleep low power modes.

Sleep mode is identical to Active from a peripheral point of view. No configuration changes are required in the component or code before entering/exiting this mode. Any communication intended to the slave causes interrupt to occur and leads to wakeup.

Deep Sleep mode requires that the slave be properly configured to be a wakeup source. The “Enable wakeup from Deep Sleep Mode” must be checked in the I2C configuration dialog. The SCB_Sleep() and SCB_Wakeup() functions must be called before/after entering/exiting Deep Sleep.

The wakeup event is slave address match. The externally clocked logic performs address matching and when a matched address is detected the hardware generated an interrupt request. But the slave behavior after address match depends on clock stretching option selection.

Clock stretching enable: the slave stretches SCL line until control is passed to the slave interrupt routine to ACK the address.

Before entering Deep Sleep, the on-going transaction intended for the slave must be completed as suggested in the following code:

```
CyGlobalIntDisable; /* Disable interrupts to lock the I2C slave state */

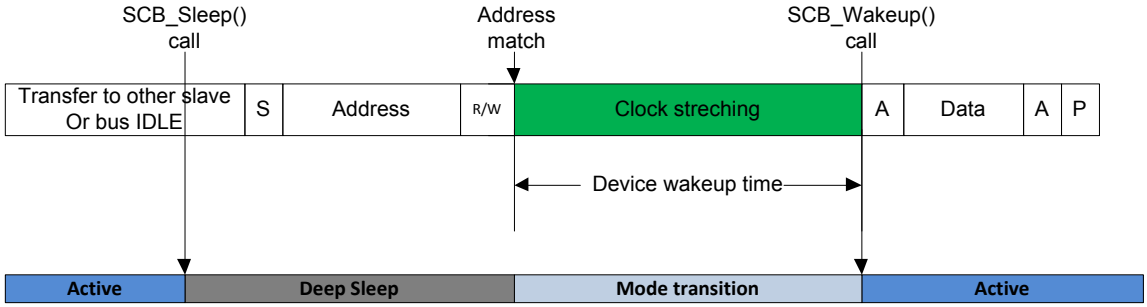
/* Checks if slave is busy */
status = (SCB_EzI2CGetActivity() & SCB_EZI2C_STATUS_BUSY);

if(0u == status)
{
    /* Slave is not busy: enter Deep Sleep */
    SCB_Sleep();             /* Configure the slave to be wakeup source */
    CySysPmDeepSleep();
    CyGlobalIntEnable;       /* Enable interrupts to continue slave operation */
    SCB_Wakeup();            /* Configure the slave to active mode operation */
}
else
{
   CyGlobalIntEnable; /* Slave is busy: do not enter Deep Sleep */
}
```

For PSoC 4000, PSoC 4100 BLE, and PSoC 4200 BLE devices, the **component clock** must be disabled before calling SCB_Sleep(), and then enabled after calling SCB_Wakeup(); otherwise, the SCL will lock up after wakeup from Deep Sleep. Disabling and re-enabling the component clock is managed by the SCB_Sleep() and SCB_Wakeup() APIs when the Clock from terminal option is disabled. Otherwise, when the Clock from terminal option is enabled, the code provided above requires modification to enable and disable the clock source connected to the SCB component. Review the following modified code and highlighted in blue (ScbClock – the instance name of clock component connected to the SCB):

```
if(0u == status)
{
    /* Slave is not busy: enter to Deep Sleep */
    SCB_Sleep();            /* Configure the slave to be wakeup source */
    ScbClock_Stop();        /* Disable the SCB clock */
    CySysPmDeepSleep();
    CyGlobalIntEnable;      /* Enable interrupts to to service wakeup source */
    SCB_Wakeup();           /* Configure the slave to active mode operation */
    ScbClock _Start();      /* Enable SCB clock */
}
```

Figure 14. Master transfer wakes up device on slave address match (Clock stretching enable)



Note The values for the primary and secondary addresses affect the range of matched addresses. The preferable choice for the secondary address is when it differs from the primary only by one bit. If it differs by more than one bit, then some transactions that are not intended for this device will still wake the device from deep sleep. The address is going to be NAKed in this case.

Clock stretching disable: the slave NAKs the matched address and any subsequent transactions until the device wakes up.

Before entering Deep Sleep, the ongoing transaction intended for the slave must be completed. The waiting loop is implemented inside the SCB_Sleep() function. This function is blocking and waits until the slave is free to configure it to be a wakeup source. For proper SCB_Sleep() function operation the slave busy status has to be managed properly by the EZI2C slave(refer to the Slave busy management section for more information). After reconfiguration, the sampling of the address match event is started and the device has time to enter Deep Sleep mode. To operate correctly in active mode, the slave configuration must be restored back by SCB_Wakeup(). The agreement between slave and master must be concluded so as not to access the slave after wakeup until SCB_Wakeup() is executed. The following code is suggested:

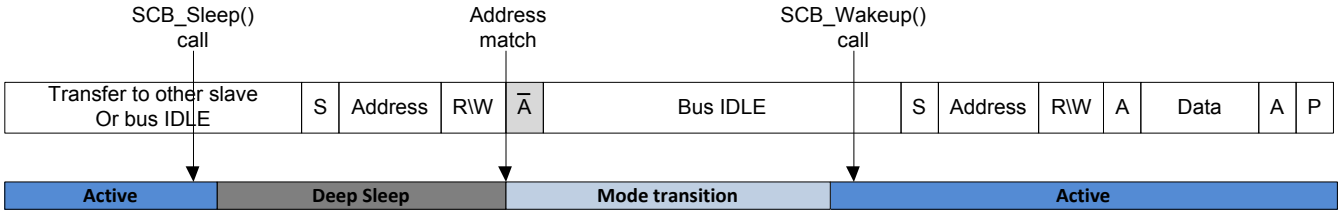
```
SCB_Sleep(); /* Wait for the slave to be free and configures it to be wakeup
source */

CySysPmDeepSleep();

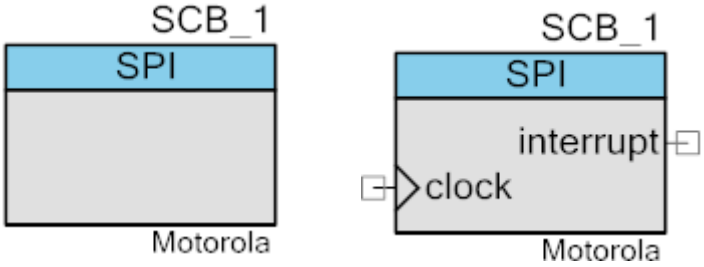
SCB_Wakeup(); /* Configure the slave to active mode operation */
```

Note The interrupts are required for the slave operations and global interrupts must be enabled before calling SCB_Sleep().

Figure 15. Master transfer wakes up device on slave address match (Clock stretching disable)



SPI



This component provides an industry-standard, 4-wire SPI interface. Three different SPI protocols or modes are supported:

- Original SPI protocol as defined by Motorola.
- TI: Uses a short pulse on “spi_select” to indicate start of transaction.
- National Semiconductor (Microwire): Transmission and Receptions occur separately.

In addition to the standard 8-bit word length, the component supports a configurable 4 to 16-bit data width for communicating at nonstandard SPI data widths.

Input/Output Connections

This section describes the various input and output connections for the SCB component. An asterisk (*) in the list of terminals indicates that the terminal may be hidden on the symbol under the conditions listed in the description of that terminal.

clock – Input*

Clock that operates this block. The presence of this terminal varies depending on the **Clock from terminal** parameter.

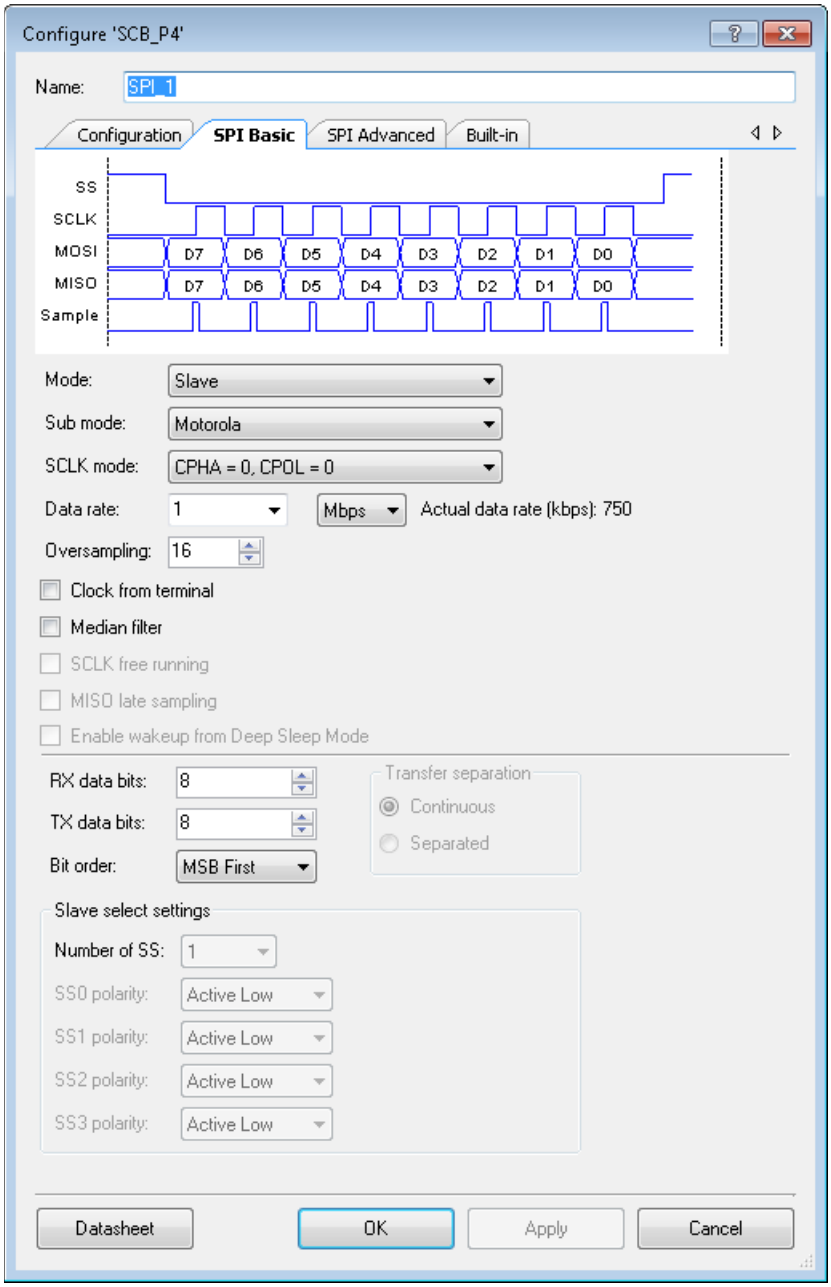
interrupt – Output*

This signal can only be connected to an interrupt component or left unconnected. The presence of this terminal varies depending on the **Interrupt** parameter.

The interface-specific pins are buried inside component because these pins use dedicated connections and are not routable as general purpose signals. See the *I/O System* section in the device *Technical Reference Manual (TRM)* for more information.

Note The input buffer of buried output pins is disabled so as not to cause current linkage in low power mode. Reading the status of these pins always returns zero. To get the current status, the input buffer must be enabled before status read.

Basic SPI Parameters



The **SPI Basic** tab contains the following parameters:

Mode

This option determines in which SPI mode the SCB operates.

- **Slave** – Slave only operation (default)
- **Master** – Master only operation

Sub mode

This option determines what SPI sub-modes are supported:

- **Motorola** – The original SPI protocol as defined by Motorola (default).
- **TI (Start Coincides)** – The Texas Instruments' SPI protocol.
- **TI (Start Precedes)** – The Texas Instruments' SPI protocol.
- **National Semiconductor (Microwire)** – The National Semiconductor's Microwire protocol.

SCLK mode

This parameter defines the serial clock phase and clock polarity mode for communication.

- **CPHA = 0, CPOL= 0 –** Data is driven on a falling edge of SCLK. Data is captured on a rising edge of SCLK. SCLK idles low.This is default mode.
- **CPHA = 0, CPOL= 1** – Data is driven on a rising edge of SCLK. Data is captured on a falling edge of SCLK. SCLK idles high.
- **CPHA = 1, CPOL= 0** – Data is driven on a rising edge of SCLK. Data is captured on a falling edge of SCLK. SCLK idles low
- **CPHA = 1, CPOL= 1** – Data is driven on a falling edge of SCLK. Data is captured on a rising edge of SCLK. SCLK idles high

Refer to the section Motorola sub mode operation in the SPI chapter of this document for more information.

Data rate

This parameter is used to set the SPI data rate value up to 8000 kbps; the actual rate may differ based on available clock frequency and component settings. The standard data rates are 500, 1000 (default), 2000, 4000 to 8000 in multiples of 2000 kbps. This parameter has no effect if the **Clock from terminal** parameter is enabled.

Actual data rate

The actual data rate displays the data rate at which the component will operate with current settings. The factors that affect the actual data rate calculation are: the accuracy of the component clock (internal or external) and oversampling factor (only for the Master mode). When a change is made to any of the component parameters that affect actual data rate, it becomes unknown. To calculate the new actual data rate press the Apply button.

Note For Slave mode the actual data rate always provides maximum value for the selected clock frequency. As external Master parameters are unknown, the assumption was made that MISO is sampled in the leading edge of SCLK.

Refer to the Slave data rate section for actual data rate calculation which takes to account external environment timing conditions.

Oversampling

This parameter defines the oversampling factor of the SPI clock; the number of component clocks within one SPI clock period. Oversampling factor is used to calculate the internal component clock frequency required to achieve this amount of oversampling as follows: SCBCLK = Data rate * Oversampling factor.

- For **Slave** mode, only the component clock source frequency is important. The oversampling value is used to create a clock fast enough to operate at the selected data rate. Refer to the Maximum data rate calculation section for more information. The created clock is equal to the (data rate * Oversampling).
- For **Master** mode, the oversampling value is used for serial clock signal (SCLK) generation. The oversampling is equal to number of component clocks within one SPI clock period. When the oversampling is even the first and second phase of the clock period are the same. Otherwise the first phase of the clock signal period is one component clock cycle longer than the second phase. The level of the first phase of the clock period depends on CPOL settings: 0 – low level and 1 – high level.

An oversampling factor maximum value is 16 and minimum depends on component settings. For **Master** the minimum oversampling factor value is 6. For **Slave** the minimum oversampling value 6 (Median filter is disabled) or 8 (Median filter is enabled).

Clock from terminal

This parameter allows choosing between an internally configured clock (by the component) or an externally configured clock (by the user) for the component operation. Refer to the **Oversampling** section to understand relationship between component clock frequency and the component parameters.

When this option is enabled, the component does not control the data rate, but displays the actual data rate based on the user-connected clock source frequency and the component oversampling factor (only for the Master mode). When this option is not enabled, the clock configuration and Oversampling factor (only for the Master mode) is provided by the component. The clock source frequency is calculated by the component based on the Data rate parameter.

Note PSoC Creator is responsible for providing requested clock frequency (internal or external clock) based on current design clock configuration. When the requested clock frequency with requested tolerance cannot be created, a warning about the clock accuracy range is generated while building the project. This warning contains the actual clock frequency value created by

PSoC Creator. To remove this warning you must either change the system clock, component settings or external clock to fit the clocking system requirements.

Median filter

This parameter applies 3 taps digital median filter on the input line. The master has one input line: MISO, and the slave has three input lines: SCLK, MOSI, and SS. This filter reduces the susceptibility to errors. However, minimum oversampling factor value is increased. The default value is a **Disabled**.

SCLK free running

This option is only applicable for PSoC 4100 BLE/PSoC 4200 BLE devices in Master mode. It allows master to generate SCLK continually. It is useful when master SCLK is connected to the slave device which uses it for functional operation rather than just SPI functionality.

The default value is a **Disabled**.

Enable late MISO sample

This option allows the master to sample the MISO signal by half of SCLK period later (on the alternate serial clock edge). Late sampling addresses the round-trip delay associated with transmitting SCLK from the master to the slave and transmitting MISO from the slave to the master. The default value is a **Disabled**.

Enable wakeup from Deep Sleep Mode

Use this option to enable the component to wake the system from Deep Sleep when slave select occurs.

To enable this option all of the following restrictions must be met:

- Sub mode is Motorola
- SCLK mode is CPHA = 0, CPOL = 0 (only for PSoC 4100/PSoC 4200 devices)
- Interrupt is Internal

Refer to the Low power modes section under the SPI chapter in this document and *Power Management APIs* section of the *System Reference Guide* for more information.

TX data bits

This option defines the bit width in a transmitted data frame. The default number of bits is a single byte (8 bits). Any integer from 4 to 16 is a valid setting.

RX data bits

This option defines the bit width in a received data frame. The default number of bits is a single byte (8 bits). Any integer from 4 to 16 is a valid setting.

Note The number of **TX data bits** and **RX data bits** should be set the same for **Motorola** and **Texas Instruments** sub-modes; they can be set different for **National Semiconductor** sub-mode.

Bit order

The **Bits order** parameter defines the direction in which the serial data is transmitted. When set to **MSB first**, the most-significant bit is transmitted first. When set to **LSB first**, the least-significant bit is transmitted first.

Number of SS

This parameter determines the number of SPI slave select lines. Only one slave select line is available in Slave mode and it is not optional. The values between 0 and 4 are valid choices in Master mode. The default number of lines is 1.

Transfer separation

This parameter determines if individual data transfers are separated by slave select de-selection:

- **Continuous –** The slave select line is held in active state until the end of transfer (default).
- **Separated –** Every data frame 4-16 bits is separated by slave select line de-selection by one SCLK period.

SS0-SS3 polarity

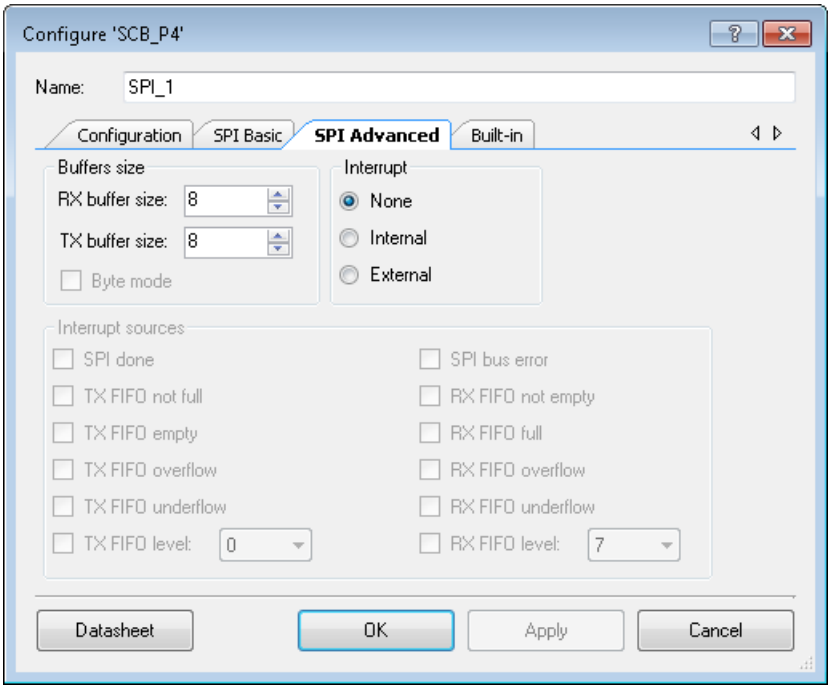
This option is only applicable for PSoC 4100 BLE/PSoC 4200 BLE devices. It determines the active polarity of the slave select signal as Active Low (default) or Active High. For other devices, only Active Low is available.

Each slave select line active polarity can be configured independently.

For Texas Instruments precede/coincide sub-modes the active polarity logic is inverted:

- **Active Low –** Slave select line is inactive low and generated pulse is active high.
- **Active High –** Slave select line is inactive high and generated pulse is active low.

Advanced SPI Parameters



The **SPI Advanced** tab contains the following parameters:

RX buffer size

The **RX buffer size** parameter defines the size (in bytes/words) of memory allocated for a receive data buffer. The minimum value is equal to the RX FIFO depth. The RX FIFO is implemented in hardware. Values greater than the RX FIFO depth up to (2^{32} – 2) imply using the RX FIFO, and a circular software buffer controlled by the supplied APIs, and the internal interrupt handler. The software buffer size is limited only by the available memory. The interrupt mode is automatically set to internal and the RX FIFO not empty interrupt source is reserved if a software buffer is used.

- For 4100/PSoC 4200 devices, the RX and TX FIFO depth is equal to 8 bytes/words.
- For PSoC 4100 BLE/PSoC 4200 BLE devices, the RX and TX FIFO depth is equal to 8 bytes/words or 16 bytes; refer to **Byte mode** for more information.

TX buffer size

The **TX buffer size** parameter defines the size (in bytes/words) of memory allocated for a circular transmit data buffer. The TX buffer size minimum value is equal to the TX FIFO depth. The TX FIFO is implemented in hardware. Values greater than the TX FIFO depth up to (2^{32} – 1) imply using the TX FIFO, circular software buffer controlled by the supplied APIs, and the internal interrupt handler. The software buffer size is limited only by the available memory. The interrupt mode is automatically set to the internal and the TX FIFO not full interrupt source is reserved if a software buffer is used.

- For 4100/PSoC 4200 devices, the RX and TX FIFO depth is equal to 8 bytes/words.
- For PSoC 4100 BLE/PSoC 4200 BLE devices, the RX and TX FIFO depth is equal to 8 bytes/words or 16 bytes; refer to **Byte mode** for more information.

Byte mode

This option is only applicable to PSoC 4100 BLE/PSoC 4200 BLE devices. It allows doubling the TX and RX FIFO depth from 8 to 16 bytes, by reducing the FIFO width from 16bits to 8 bits. This implies that the number of TX and RX data bits must be less than or equal to 8 bits. Increasing FIFO depth improves performance of SPI operation as more bytes can be transmitted or received without software interaction.

Interrupt

This option determines what interrupt modes are supported None, Internal or External.

- **None** – This option removes the internal interrupt component.
- **Internal** – This option leaves the interrupt component inside the SCB component. The predefined internal interrupt handler is hooked up to the interrupt. The **Interrupt sources** option sets one or more interrupt sources, which trigger the interrupt. To add your own code to the interrupt service routine you need to register a function using the SCB_SetCustomInterruptHandler() function.
- **External** – This option removes the internal interrupt and provides an output terminal. Only an interrupt component can be connected to this terminal if an interrupt handler is desired. The **Interrupt sources** option sets one or more interrupt sources, which trigger the interrupt output.

Note For buffer sizes greater than the hardware FIFO depth, the component automatically enables the internal interrupt sources required for proper internal software buffer operations. In addition, the global interrupt enable must be explicitly enabled for proper buffer handling.

Interrupt sources

The interrupt sources are either level or pulse. Level interrupt sources in the following list are indicated with an asterisk (*). Refer to sections TX FIFO interrupt sources and RX FIFO interrupt sources for more information about level interrupt sources operation. The SPI supports interrupts on the following events:

- SPI done – Master transfer done event: all data elements from the TX FIFO are sent. This interrupt source triggers later than TX FIFO empty by the amount of time it takes to transmit a single data element. The TX FIFO empty triggers when the last data element from the TX FIFO goes to the shifter register. However, SPI done triggers after this data element has been transmitted. This means SPI done will be asserted one SCLK clock cycle earlier than the reception of the data element has been completed. It is

recommended to use SCB_SpiIsBusBusy() after checking SPI done to determine when the data element reception has been fully completed. As an alternative, the number of received data elements can be checked to make sure that it is equal to the number of the transmitted data elements.

- **TX FIFO not full * –** TX FIFO is not full. At least one data element can be written into the TX FIFO.
- **TX FIFO empty * –** TX FIFO is empty.
- **TX FIFO overflow –** Firmware attempts to write to a full TX FIFO.
- **TX FIFO underflow –** Hardware attempts to read from an empty TX FIFO.
- **TX FIFO level * –** When the TX FIFO has fewer entries than the number in this field, an interrupt request is generated.
- **SPI bus error –** SPI slave deselected at an unexpected time during the SPI transfer.
- **RX FIFO not empty * –** RX FIFO is not empty. At least one data element is available in the RX FIFO to be read.
- **RX FIFO full * –** RX FIFO is full.
- **RX FIFO overflow –** Hardware attempts to write to a full RX FIFO.
- **RX FIFO underflow –** Firmware Attempts to read from an empty RX FIFO.
- **RX FIFO level * –** When the RX FIFO has more entries than the number in this field, an interrupt request is generated.

Notes

When **RX buffer size** is greater than the RX FIFO depth, the **RX FIFO not empty** interrupt source is reserved by the component and used for the internal interrupt.

When **TX buffer size** is greater than the TX FIFO depth, the **TX FIFO not full** interrupt source is reserved by the component and used for the internal interrupt.

SPI APIs

APIs allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections discuss each function in more detail.

By default, PSoC Creator assigns the instance name “SCB_1” to the first instance of a component in a given design. You can rename the instance to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “SCB”.

Function	Description
SCB_Start()	Starts the SCB.
SCB_Init()	Initialize the SCB component according to defined parameters in the customizer.
SCB_Enable()	Enables SCB component operation.
SCB_Stop()	Disable the SCB component.
SCB_Sleep()	Prepares component to enter Deep Sleep.
SCB_Wakeup()	Prepares component for Active mode operation after Deep Sleep.
SCB_SpiInit()	Configures the SCB for SPI operation.
SCB_SpiIsBusBusy()	Returns the current status on the bus.
SCB_SpiSetActiveSlaveSelect()	Selects the active slave select line. Only applicable in Master mode.
SCB_SpiSetSlaveSelectPolarity()	Sets active polarity for the slave select line.
SCB_SpiUartWriteTxData()	Places a data entry into the transmit buffer to be sent at the next available bus time.
SCB_SpiUartPutArray()	Places an array of data into the transmit buffer to be sent.
SCB_SpiUartGetTxBufferSize()	Returns the number of elements currently in the transmit buffer.
SCB_SpiUartClearTxBuffer()	Clears the transmit buffer and TX FIFO.
SCB_SpiUartReadRxData()	Retrieves the next data element from the receive buffer.
SCB_SpiUartGetRxBufferSize()	Returns the number of received data elements in the receive buffer.
SCB_SpiUartClearRxBuffer()	Clears the receive buffer and RX FIFO.

void SCB_Start(void)

Description: Invokes SCB_Init() and SCB_Enable(). After this function call the component is enabled and ready for operation. This is the preferred method to begin component operation.

When configuration is set to “Unconfigured SCB”, the component must first be initialized to operate in one of the following configurations: I^2C, SPI, UART or EZ I^2C. Otherwise this function does not enable component.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Init(void)

Description: Initializes the SCB component to operate in one of the selected configurations: I^2C, SPI, UART or EZ I^2C.

When configuration is set to “Unconfigured SCB”, this function does not do any initialization. Use mode-specific initialization APIs instead: SCB_I2CInit, SCB_SpiInit, SCB_UartInit or SCB_EzI2CInit.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Enable(void)

Description: Enables SCB component operation: activates the hardware and internal interrupt.

For I^2C and EZ I^2C modes the interrupt is internal and mandatory for operation. For SPI and UART modes the interrupt can be configured as none, internal or external.

The SCB configuration should be not changed when the component is enabled. Any configuration changes should be made after disabling the component.

When configuration is set to “Unconfigured SCB”, the component must first be initialized to operate in one of the following configurations: I^2C, SPI, UART or EZ I^2C. Otherwise this function does not enable component.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Stop(void)

Description: Disables the SCB component: disable the hardware and internal interrupt. Refer to the function SCB_Enable() for the interrupt configuration details.

This function disables the SCB component without checking to see if communication is in progress. Before calling this function it may be necessary to check the status of communication to make sure communication is complete. If this is not done then communication could be stopped mid byte and corrupted data could result.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Sleep(void)

Description: Prepares component to enter Deep Sleep.

The “Enable wakeup from Deep Sleep Mode” selection has an influence on this function implementation:

- Checked: configures the component to be wakeup source from Deep Sleep.
- Unchecked: stores the current component state (enabled or disabled) and disables the component. See SCB_Stop() function for details about component disabling.

Call the SCB_Sleep() function before calling the CyPmSysDeepSleep() function. Refer to the PSoC Creator *System Reference Guide* for more information about power-management functions.

This function should not be called before entering Sleep.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Wakeup(void)

Description: Prepares component for Active mode operation after Deep Sleep.

The “Enable wakeup from Deep Sleep Mode” selection has influence to on this function implementation:

- Checked: restores the component Active mode configuration.
- Unchecked: enables the component if it was enabled before enter Deep Sleep.

This function should not be called after exiting Sleep.

Parameters: None

Return Value: None

Side Effects: Calling the SCB_Wakeup() function without first calling the SCB_Sleep() function may produce unexpected behavior.

void SCB_SpiInit(SCB_SPI_INIT_STRUCT *config)

Description: Configures the SCB for SPI operation.

This function is **intended specifically** to be used when the SCB configuration is set to "Unconfigured SCB" in the customizer. After initializing the SCB in SPI mode, the component can be enabled using the SCB_Start() or SCB_Enable() function.

This function uses a pointer to a structure that provides the configuration settings. This structure contains the same information that would otherwise be provided by the customizer settings.

Parameters: config: pointer to a structure that contains the following list of fields. These fields match the selections available in the customizer. Refer to the customizer for further description of the settings.

Field	Description
uint32 mode	Mode of operation for SPI. The following defines are available choices: SCB_SPI_SLAVE SCB_SPI_MASTER
uint32 submode	Submode of operation for SPI. The following defines are available choices: SCB_SPI_MODE_MOTOROLA SCB_SPI_MODE_TI_COINCIDES SCB_SPI_MODE_TI_PRECEDES SCB_SPI_MODE_NATIONAL
uint32 sclkMode	Determines the sclk relationship for Motorola submode. Ignored for other submodes. The following defines are available choices: SCB_SPI_SCLK_CPHA0_CPOL0 SCB_SPI_SCLK_CPHA0_CPOL1 SCB_SPI_SCLK_CPHA1_CPOL0 SCB_SPI_SCLK_CPHA1_CPOL1
uint32 oversample	Oversampling factor for the SPI clock. Ignored for Slave mode operation.
uint32 enableMedianFilter	0 – disable 1 – enable
uint32 enableLateSampling	0 – disable 1 – enable Ignored for slave mode.
uint32 enableWake	0 – disable 1 – enable Ignored for master mode.
uint32 rxDataBits	Number of data bits for RX direction. Different dataBitsRx and dataBitsTx are only allowed for National submode.
uint32 txDataBits	Number of data bits for TX direction. Different dataBitsRx and dataBitsTx are only allowed for National submode.

uint32 bitOrder	Determines the bit ordering. The following defines are available choices: SCB_BITS_ORDER_LSB_FIRST SCB_BITS_ORDER_MSB_FIRST
uint32 transferSeperation	Determines whether transfers are back to back or have SS disabled between words. Ignored for slave mode. The following defines are available choices: SCB_SPI_TRANSFER_CONTINUOUS SCB_SPI_TRANSFER_SEPARATED
uint32 rxBufferSize	Size of the RX buffer in bytes/words (depends on rxDataBits parameter). A value equal to the RX FIFO depth implies the usage of buffering in hardware. A value greater than the RX FIFO depth results in a software buffer. The SCB_INTR _RX_NOT_EMPTY interrupt has to be enabled to transfer data into the software buffer. For 4100/PSoC 4200 devices, the RX and TX FIFO depth is equal to 8 bytes/words. For PSoC 4100 BLE/PSoC 4200 BLE devices, the RX and TX FIFO depth is equal to 8 bytes/words or 16 bytes (Byte mode is enabled).
uint8* rxBuffer	Buffer space provided for a RX software buffer: • A NULL pointer must be provided to use hardware buffering. • A pointer to an allocated buffer must be provided to use software buffering. The buffer size must equal (rxBufferSize + 1) in bytes if dataBitsRx is less or equal to 8, otherwise (2 * (rxBufferSize + 1)) in bytes. The software RX buffer always keeps one element empty. For correct operation the allocated RX buffer has to be one element greater than maximum packet size expected to be received.
uint32 txBufferSize	Size of the TX buffer in bytes/words(depends on txDataBits parameter). A value equal to the TX FIFO depth implies the usage of buffering in hardware. A value greater than the TX FIFO depth results in a software buffer. For 4100/PSoC 4200 devices, the RX and TX FIFO depth is equal to 8 bytes/words. For PSoC 4100 BLE/PSoC 4200 BLE devices, the RX and TX FIFO depth is equal to 8 bytes/words or 16 bytes (Byte mode is enabled).
uint8* txBuffer	Buffer space provided for a TX software buffer: • A NULL pointer must be provided to use hardware buffering. • A pointer to an allocated buffer must be provided to use software buffering. The buffer size must equal txBufferSize if dataBitsTx is less or eqal to 8, otherwise (2* rxBufferSize).
uint32 enableInterrupt	0 – disable 1 – enable The interrupt has to be enabled if software buffer is used.

uint32 rxInterruptMask	Mask of enabled interrupt sources for the RX direction. This mask is written regardless of the setting of the enable Interrupt field. Multiple sources are enabled by providing a value that is the OR of all of the following sources to enable: • SCB_INTR_RX_FIFO_LEVEL • SCB_INTR_RX_NOT_EMPTY • SCB_INTR_RX_FULL • SCB_INTR_RX_OVERFLOW • SCB_INTR_RX_UNDERFLOW • SCB_INTR_SLAVE_SPI_BUS_ERROR
uint32 rxTriggerLevel	FIFO level for an RX FIFO level interrupt. This value is written regardless of whether the RX FIFO level interrupt source is enabled.
uint32 txInterruptMask	Mask of enabled interrupt sources for the TX direction. This mask is written regardless of the setting of the enable Interrupt field. Multiple sources are enabled by providing a value that is the OR of all of the following sources to enable: • SCB_INTR_TX_FIFO_LEVEL • SCB_INTR_TX_NOT_FULL • SCB_INTR_TX_EMPTY • SCB_INTR_TX_OVERFLOW • SCB_INTR_TX_UNDERFLOW • SCB_INTR_MASTER_SPI_DONE
uint32 txTriggerLevel	FIFO level for a TX FIFO level interrupt. This value is written regardless of whether the TX FIFO level interrupt source is enabled.
uint8 enableByteMode	Ignored for all devices other than For PSoC 4100 BLE/PSoC 4200 BLE. 0 – disable 1 – enable When enabled the TX and RX FIFO depth is 16 bytes. This implies that number of TX and RX data bits must be less than or equal to 8.
uint8 enableFreeRunSclk	Ignored for all devices other than For PSoC 4100 BLE/PSoC 4200 BLE. Enables continuous SCLK generation by the SPI master. 0 – disable 1 – enable
uint8 polaritySs	Ignored for all devices other than For PSoC 4100 BLE/PSoC 4200 BLE. Active polarity of slave select lines 0-3. This is bitmask where bit SCB_SPI_SLAVE_SELECT0 corresponds to slave select 0 polarity, bit SCB_SPI_SLAVE_SELECT1 – slave select 1 polarity and so on. Polarity constants are: SCB_SPI_SS_ACTIVE_LOW SCB_SPI_SS_ACTIVE_HIGH

Return Value: None

Side Effects: None

uint32 SCB_SpiIsBusBusy(void)

Description: Returns the current status on the bus. The bus status is determined using the slave select signal.

- Motorola and National Semiconductor sub-modes: The bus is busy after the slave select line is activated and lasts until the slave select line is deactivated.
- Texas Instrument sub-modes: The bus is busy at the moment of the initial pulse on the slave select line and lasts until the transfer is complete.

If SPI Master is configured to use "separated transfers" (see Continuous versus Separated Transfer Separation), the bus is busy during each element transfer and is free between each element transfer. The Master does not activate SS line immediately after data has been written into the TX FIFO.

Parameters: None

Return Value: uint32: Current status on the bus. If the returned value is nonzero, the bus is busy. If zero is returned, the bus is free. The bus status is determined using the slave select signal.

Side Effects: None

void SCB_SpiSetActiveSlaveSelect(uint32 slaveSelect)

Description: Selects one of the four slave select lines to be active during the transfer. After initialization the active slave select line is 0.

The component should be in one of the following states to change the active slave select signal source correctly:

- The component is disabled
- The component has completed transfer

This function does not check that these conditions are met.

This function is only applicable to SPI Master mode of operation.

Parameters: uint32 slaveSelect: slave select line that will be active after the transfer.

Active Slave Select constants	Description
SCB_SPI_SLAVE_SELECT0	Slave select 0
SCB_SPI_SLAVE_SELECT1	Slave select 1
SCB_SPI_SLAVE_SELECT2	Slave select 2
SCB_SPI_SLAVE_SELECT3	Slave select 3

Return Value: None

Side Effects: None

void SCB_SpiSetSlaveSelectPolarity(uint32 slaveSelect, uint32 polarity)

Description: Sets active polarity for the slave select line.

The component should be in one of the following states to change the active slave select signal correctly:

- The component is disabled
- The component has completed transfer

This function does not check that these conditions are met.

Parameters: uint32 slaveSelect: slave select line to change active polarity.

Slave Select constants	Description
SCB_SPI_SLAVE_SELECT0	Slave select 0. For SPI slave mode the slave select 0 is only valid argument due to slave select placement constraint.
SCB_SPI_SLAVE_SELECT1	Slave select 1
SCB_SPI_SLAVE_SELECT2	Slave select 2
SCB_SPI_SLAVE_SELECT3	Slave select 3

uint32 Polarity: active polarity of slave select line.

Active Slave Select constants	Description
SCB_SPI_SS_ACTIVE_LOW	Slave select is active low
SCB_SPI_SS_ACTIVE_HIGH	Slave select is active high

Return Value: None

Side Effects: None

void SCB_SpiUartWriteTxData(uint32 txData)

Description: Places a data entry into the transmit buffer to be sent at the next available bus time.

This function is blocking and waits until there is space available to put the requested data in the transmit buffer.

Parameters: uint32 txData: the data to be transmitted.

The amount of data bits to be transmitted depends on TX data bits selection (the data bit counting starts from LSB of txDataByte).

Return Value: None

Side Effects: None

void SCB_SpiUartPutArray(const uint16/uint8 wrBuf[], uint32 count)

Description: Places an array of data into the transmit buffer to be sent.

This function is blocking and waits until there is a space available to put all the requested data in the transmit buffer.

The array size can be greater than transmit buffer size.

Parameters: const uint16/uint8 wrBuf[]: pointer to an array of data to be placed in transmit buffer. The width of the data to be transmitted depends on TX data width selection (the data bit counting starts from LSB for each array element).

uint32 count: number of data elements to be placed in the transmit buffer.

Return Value: None

Side Effects: None

uint32 SCB_SpiUartGetTxBufferSize(void)

Description: Returns the number of elements currently in the transmit buffer.

TX software buffer is disabled: Returns the number of used entries in TX FIFO.

TX software buffer is enabled: Returns the number of elements currently used in the transmit buffer. This number does not include used entries in the TX FIFO. The transmit buffer size is zero until the TX FIFO is not full.

Parameters: None

Return Value: uint32: Number of data elements ready to transmit.

Side Effects: None

void SCB_SpiUartClearTxBuffer(void)

Description: Clears the transmit buffer and TX FIFO.

Parameters: None

Return Value: None

Side Effects: None

uint32 SCB_SpiUartReadRxData(void)

Description: Retrieves the next data element from the receive buffer.

RX software buffer is disabled: Returns data element retrieved from RX FIFO. Undefined data will be returned if the RX FIFO is empty.

RX software buffer is enabled: Returns data element from the software receive buffer. Zero value is returned if the software receive buffer is empty.

Parameters: None

Return Value: uint32: Next data element from the receive buffer. The amount of data bits to be received depends on RX data bits selection (the data bit counting starts from LSB of return value).

Side Effects: None

uint32 SCB_SpiUartGetRxBufferSize(void)

Description: Returns the number of received data elements in the receive buffer.

RX software buffer is disabled: Returns the number of used entries in RX FIFO.

RX software buffer is enabled: Returns the number of elements that were placed in the receive buffer. This does not include the hardware RX FIFO.

Parameters: None

Return Value: uint32: Number of received data elements

Side Effects: None

void SCB_SpiUartClearRxBuffer(void)

Description: Clears the receive buffer and RX FIFO.

Parameters: None

Return Value: None

Side Effects: None

Global Variables

Knowledge of these variables is not required for normal operations.

Variable	Description
SCB_initVar	SCB_initVar indicates whether the SCB component has been initialized. The variable is initialized to 0 and set to 1 the first time SCB_Start() is called. This allows the component to restart without reinitialization after the first call to the SCB_Start() routine. If reinitialization of the component is required, then the SCB_Init() function can be called before the SCB_Start() or SCB_Enable() function.

Variable	Description
SCB_rxBufferOverflow	SCB_rxBufferOverflow sets when internal software receive buffer overflow was occurred.

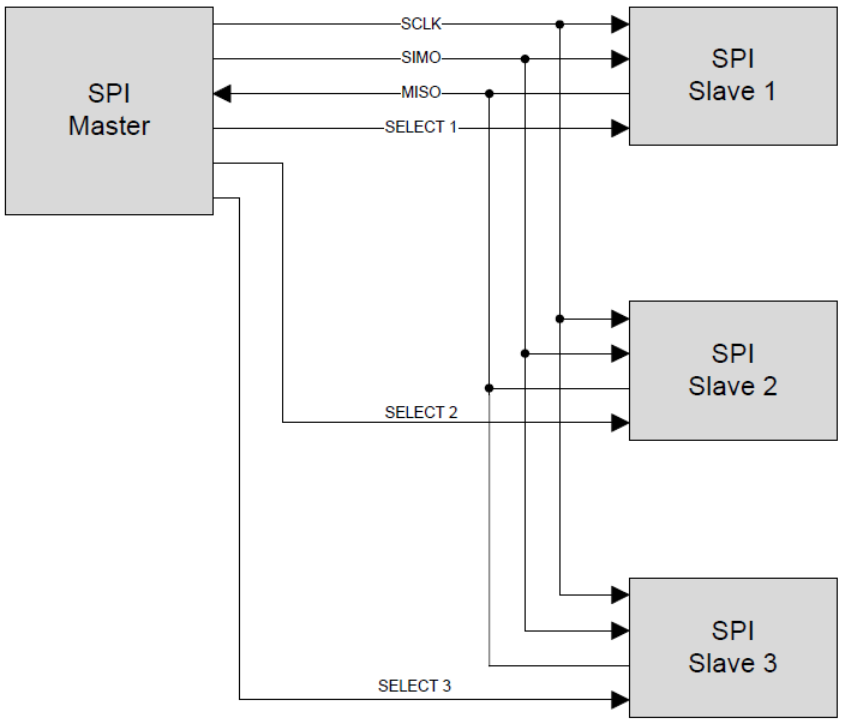
SPI Functional Description

The Serial Peripheral Interface (SPI) protocol is a synchronous serial interface, with “single-master-multi-slave” topology. Devices operate in either master or slave mode. The master initiates transfers of data frames. Multiple slaves are supported with individual slave select lines.

The SPI interface consists of four signals:

- **SCLK** – Serial clock (output from master, input to the slave).
- **MOSI** – Master output, slave input (output from the master, input to the slave).
- **MISO** – Master input, slave output (input to the master, output from the slave).
- **SELECT** – Slave select (typically an active low signal, output from the master, input to the slave).

Figure 16. SPI Bus Connections Example



Motorola sub mode operation

This is the original SPI protocol defined by Motorola. It is a full duplex protocol: transmission and reception occur at the same time.

The Motorola SPI protocol has four different modes that determine how data is driven and captured on the MOSI and MISO lines. These modes are determined by clock polarity (CPOL) and clock phase (CPHA).

- **CPHA = 0, CPOL= 0 –** Data is driven on a falling edge of SCLK. Data is captured on a rising edge of SCLK. The idle state of SCLK line is low.
- **CPHA = 0, CPOL= 1** – Data is driven on a rising edge of SCLK. Data is captured on a falling edge of SCLK. The idle state of SCLK line is high.
- **CPHA = 1, CPOL= 0** – Data is driven on a rising edge of SCLK. Data is captured on a falling edge of SCLK. The idle state of SCLK line is low.
- **CPHA = 1, CPOL= 1** – Data is driven on a falling edge of SCLK. Data is captured on a rising edge of SCLK. The idle state of SCLK line is high.

Figure 17 illustrates driving and capturing of MOSI/MISO data as a function of CPOL and CPHA.

Figure 17. SPI Motorola frame format

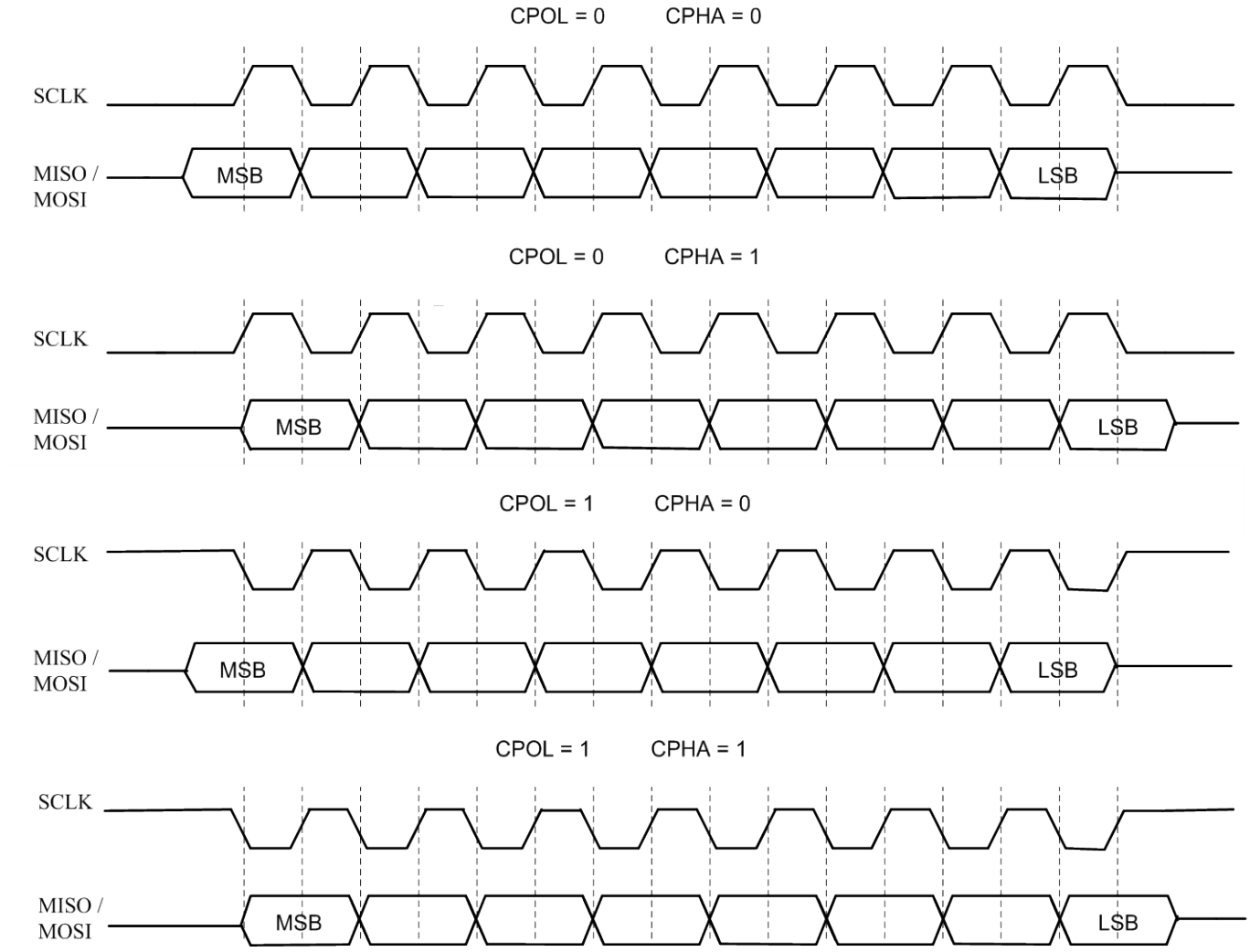
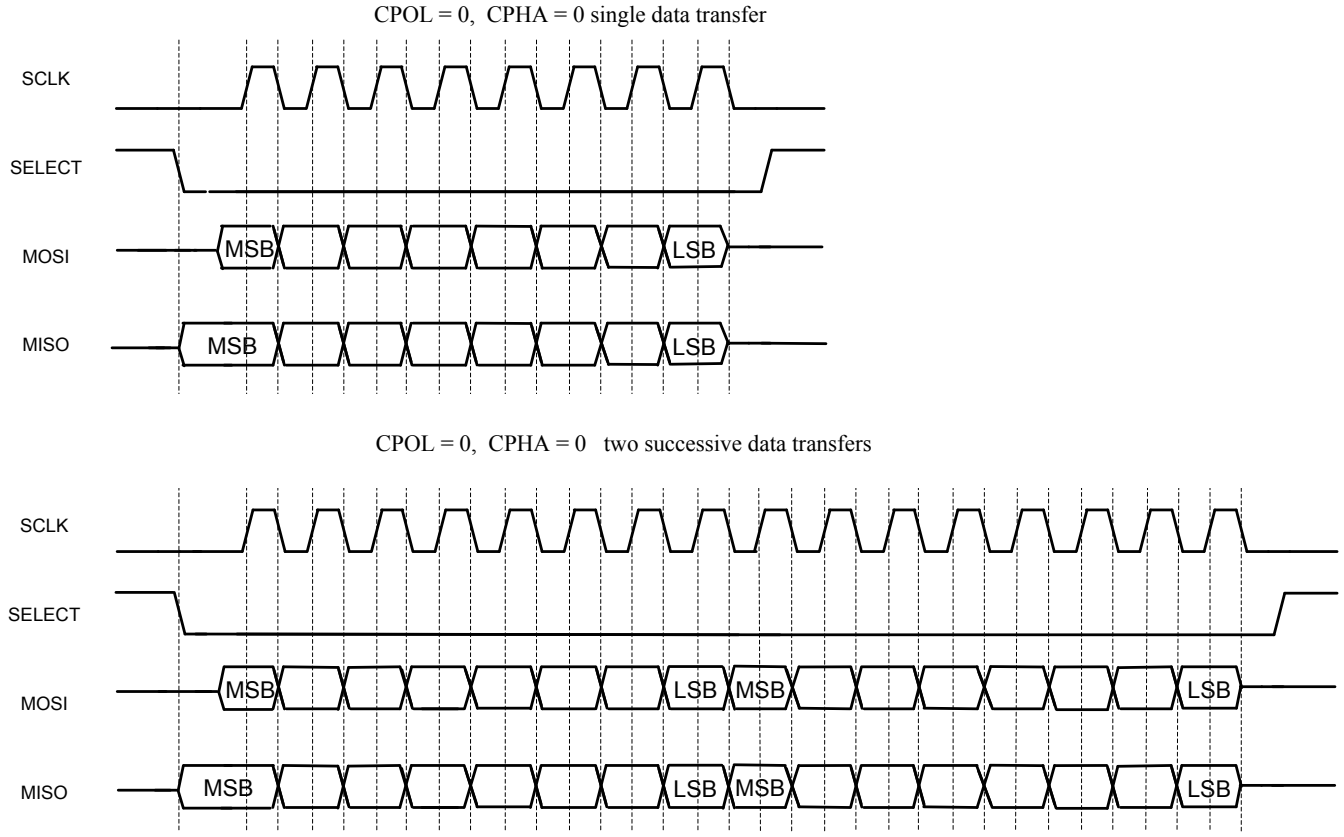


Figure 18 illustrates a single 8-bit data transfer and two successive 8-bit data transfers in mode 0 (CPOL is '0', CPHA is '0').

Figure 18. SPI Motorola Data Transfer Example



Texas Instruments sub modes operation

The Texas Instruments' SPI protocol redefines the use of the SS signal. It uses the signal to indicate the start of a data transfer, rather than a low active slave select signal. This protocol only supports CPHA = 1, CPOL= 0.

The start of a transfer is indicated by a high active pulse of a single bit transfer period. This pulse may precede the transfer of the first data frame bit on one SCLK period, or may coincide with the transmission of the first data bit. The transmitted clock SCLK is a free running clock.

Figure 19 illustrates a single 8-bit data transfer and two successive 8-bit data transfers. The SS pulse precedes the first data bit.

Note The SELECT pulse of the second data transfer coincides with the last data bit of the first data transfer.

Figure 19. TI (Precede) Data Transfer Example

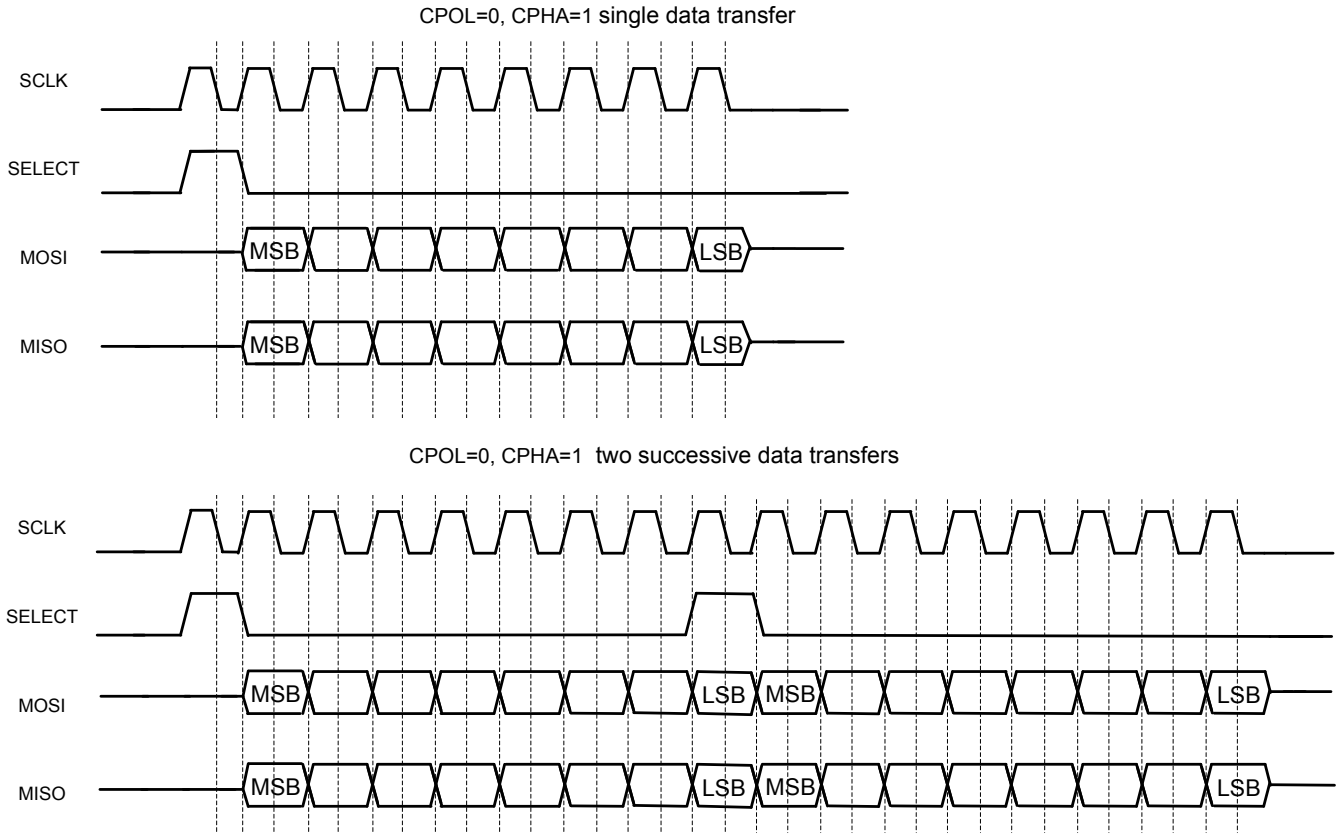
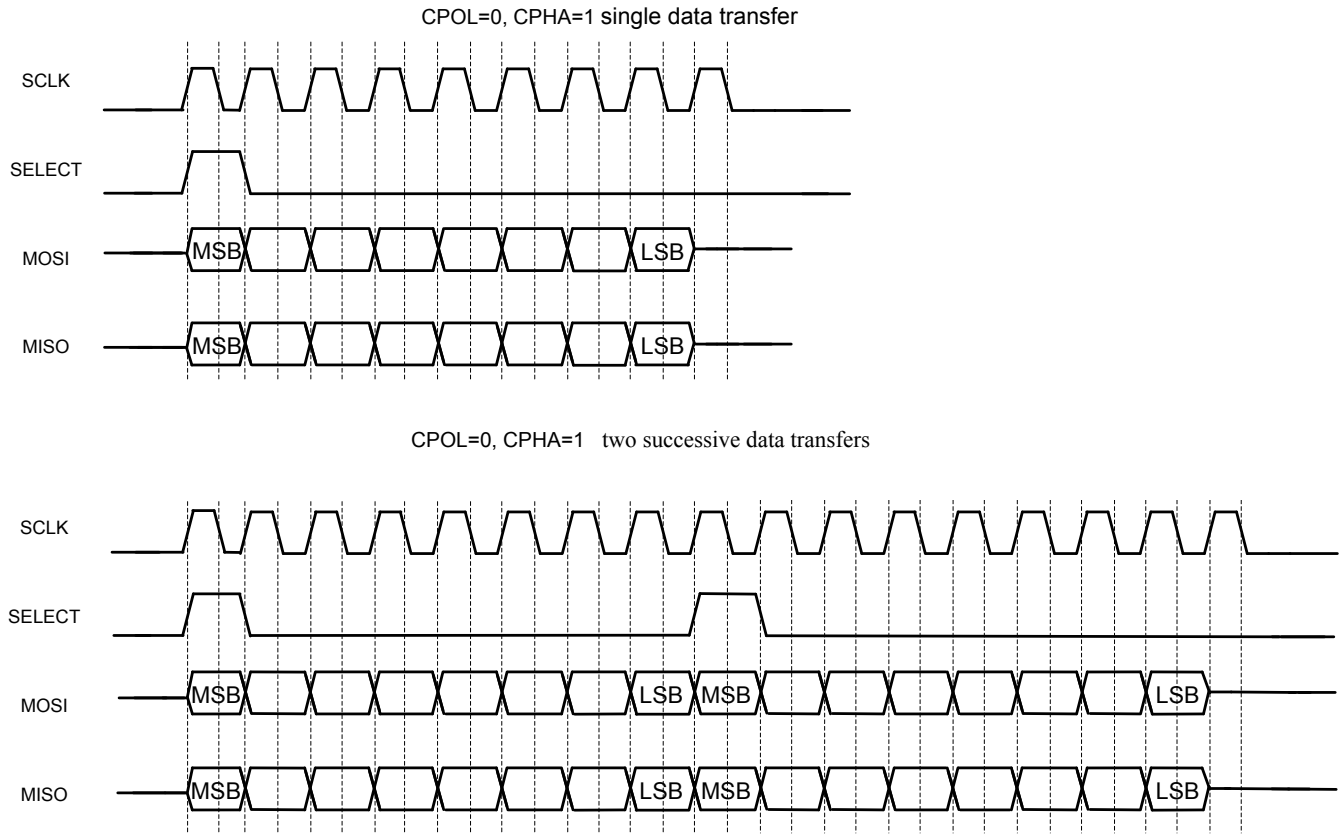


Figure 20 illustrates a single 8-bit data transfer and two successive 8-bit data transfers. The SS pulse coincides with the first data bit.

Figure 20. TI (Coincide) Data Transfer Example



National Semiconductor's Microwire sub modes operation

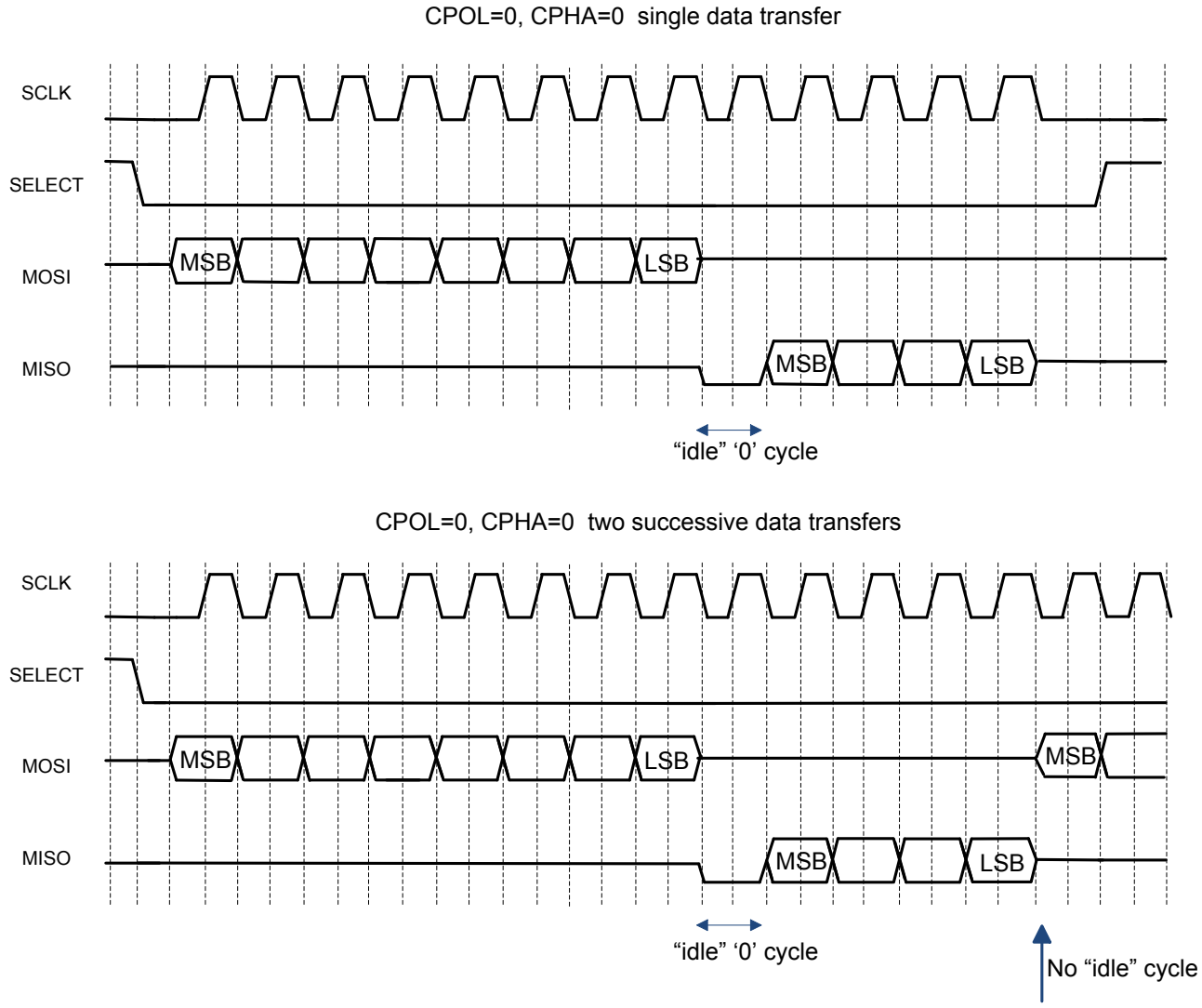
The National Semiconductor's Microwire protocol is a half-duplex protocol. Rather than transmission and reception occurring at the same time, transmission and reception take turns (transmission happens before reception). A single "idle" bit transfer period separates transmission from reception. This protocol only supports CPHA = 1, CPOL= 0.

Note The successive data transfers (transmission and reception) are NOT separated by an "idle" bit transfer period.

The transmission data transfer size and reception data transfer size may differ.

Figure 21 illustrates a single data transfer and two successive data transfers. In both cases the transmission data transfer size is 8 bits and the reception transfer size is 4 bits.

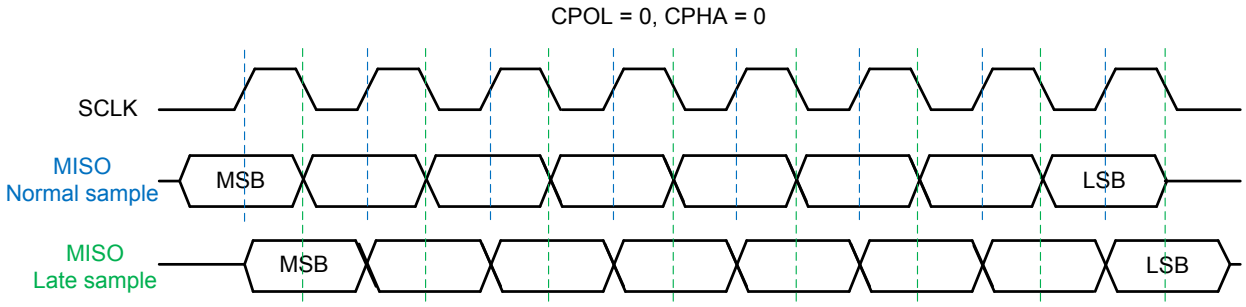
Figure 21. National Semiconductor's Microwire Data Transfer Example



MISO late sampling

The MISO is captured by Master by half of SCLK period later (on the alternate serial clock edge). Late sampling addresses the round-trip delay associated with transmitting SCLK from the master to the slave and transmitting MISO from the slave to the master.

Figure 22. Late MISO sampling example



Slave select lines

The slave select lines are used by the master to notify the slave device that it will communicate with it. The master has control of four slave select lines, and one of them has to be chosen as active before starting communication. To start communication, the data is written into the TX FIFO. The master hardware then asserts the active slave select line and sends data to the MOSI.

There are cases when firmware control of the slave select line is desired. In this case, the slave select lines that are controlled by the master hardware need to be deactivated. There are two options to do this:

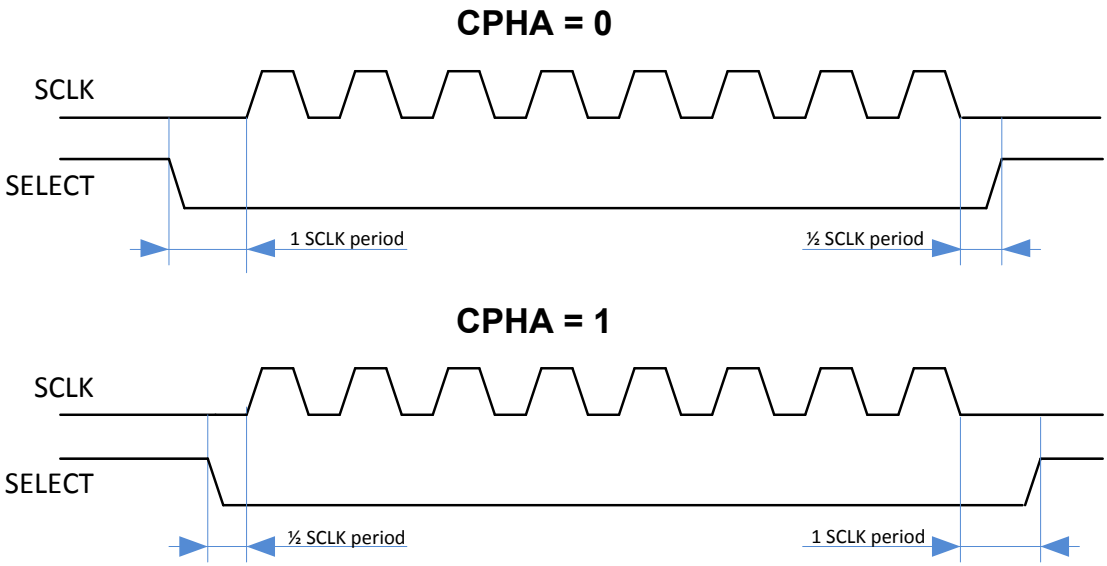
- Set the active slave select line to one that is not routed out to the pin. This option is recommended when less than four slave select lines are used by the master. Example: SPI master consumes 3 slave select lines SS0, SS1 and SS2. Call SCB_SpiSetActiveSlaveSelect(SCB_SPIM_ACTIVE_SS3) to deactivate hardware controlled slave select lines SS0-SS2.
- Change the source of the control of the active slave select line in HSIOM (High Speed I/O Matrix) from the SCB SPI interface to GPIO (CPU firmware control). Refer to the High-Speed I/O Matrix description in the *Technical Reference Manual (TRM)* for more information. This option is recommended when the master uses four slave select lines or when multiplexing between hardware controlled and firmware controlled slave select lines is required.

SELECT and SCLK Timing Correlation

The master activates SELECT before starting the transfer and makes it inactive when the transfer is completed. A minimum time is guaranteed before SELECT activation and the first

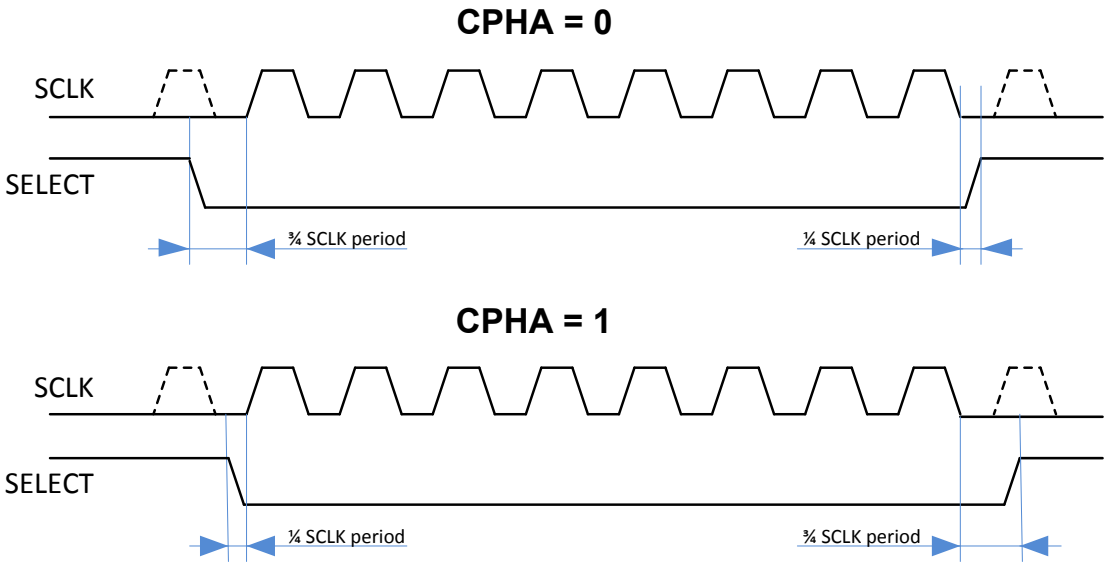
SCLK edge and SELECT deactivation and last SCLK edge. This time depends on the master sampling edge, which is defined by CPHA settings. Thus, two combinations are available.

Figure 23. SELECT and SCLK Timing Correlation (PSoC 4100/PSoC 4200)



Note PSoC 4100/PSoC 4200 devices support only SCLK gated and active low SELECT polarity.

Figure 24. SELECT and SCLK Timing Correlation (PSoC 4100 BLE/PSoC 4200 BLE)

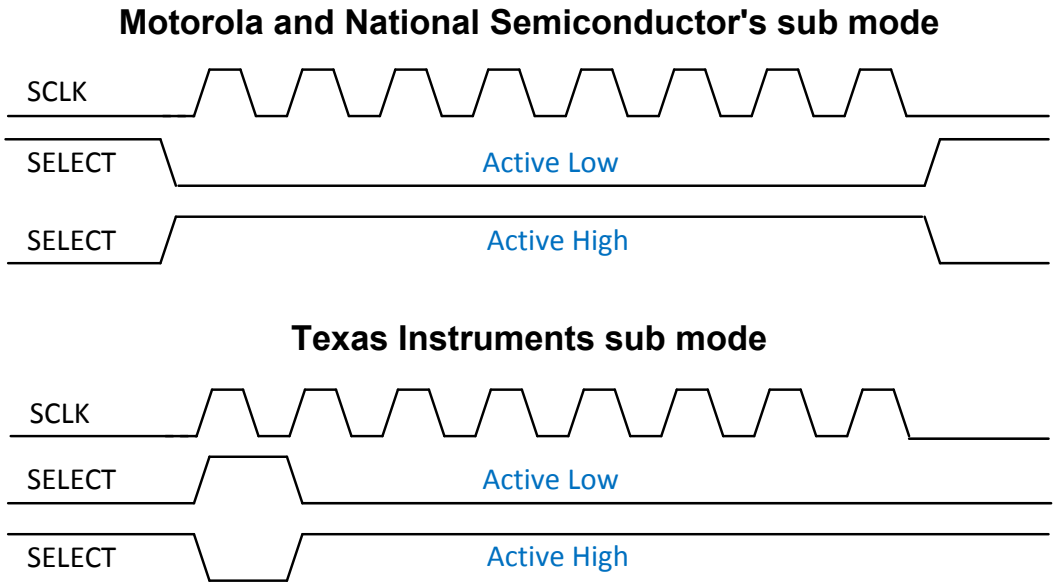


Note PSoC 4100 BLE/PSoC 4200 BLE devices support SCLK gated and free running, as well as active low and high SELECT polarity. For all configurations, the same correlation is preserved.

SELECT polarity

The SELECT line polarity for PSoC 4100/PSoC 4200 devices is active low. PSoC 4100 BLE/PSoC 4200 BLE devices provide the capability to select the active polarity of the line as active low or active high.

Figure 25. SELECT line polarity



Continuous versus Separated Transfer Separation

During separated data transfer, the SELECT line always changes from active to inactive state between the individual data frames until completion of the transfer.

During continuous data transfer, the individual data frame is not necessarily separated by the SELECT line inactivation. At the start of data transfer, the SELECT line is activated and keeps its state active until the end of transfer. The end of transfer is defined as all data from the TX FIFO is sent out and SPI Done event triggered. Figure 12 on page 72 illustrates two continuous 8-bit data transfers in SCLK mode: CPHA=0, CPOL= 0.

FIFO depth

The hardware provides two FIFOs. One is used for the receive direction, RX FIFO, and the other for transmit direction, TX FIFO. The FIFO depth is 8 data elements. The width of each data element is 16 bits. The data frame width is configurable from 4-16 bits. One element from the FIFO is consumed regardless of the data frame width.

PSoC 4100 BLE/PSoC 4200 BLE devices provide the ability to double the FIFO depth to be 16 data elements when the data frame width is 4-8 bits.

Software Buffer

Selecting RX or TX Buffer Size values greater than the FIFO depth enables usage of the RX or TX FIFO and a circular software buffer. An array of the requested size is allocated internally by the component for the TX software buffer. The allocated array for RX software buffer has one extra element that remains empty while in operation. Keeping this element empty simplifies circular buffer operation. The interrupt option is automatically set to Internal, and the RX or TX interrupt source is reserved to provide software buffer operation.

The internal interrupt is connected to the interrupt output. This interrupt runs a predefined interrupt service routine. Its main purpose is to provide interaction between software buffers and hardware RX or TX FIFO. The software buffer overflow can happen only for the RX direction. This event is reported via global variable SCB_rxBufferOverflow. For the TX direction, the provided APIs do not allow the software buffer overflow.

Interrupts

When **RX buffer size** or **TX buffer size** is greater than the FIFO depth, the **RX FIFO not empty** or **TX FIFO not full** interrupt sources are reserved by the component for the internal software buffers operations. **Do not clear or disable them** because it causes incorrect software buffer operation. However, it is the user's responsibility to clear interrupt events from other enabled interrupt sources because they are not cleared automatically. Create a custom function that clears these interrupt sources and register it using SCB_SetCustomInterruptHandler(). Each time an internal interrupt handler executes, the custom function is called before handling software buffer operation.

In case **RX buffer size** or **TX buffer size** is equal to the FIFO depth instead of software buffer only the hardware TX or RX FIFO is used. In the **Internal** interrupt mode the interrupts are not cleared automatically. It is user responsibility to do this. The **External** or **None** interrupt selection is preferred in this case.

Low power modes

The component in SPI mode is able to be a wakeup source from Sleep and Deep Sleep low power modes.

Sleep mode is identical to Active from a peripheral point of view. No configuration changes are required in the component or code before entering/exiting this mode. Any communication intended for the slave causes an interrupt to occur and leads to wakeup. Any master activity that causes an interrupt to occur leads to wakeup.

The master mode is not able to be a wakeup source from Deep Sleep. This capability is only available in slave mode. Deep Sleep mode requires that the slave be properly configured to be a wakeup source. The "Enable wakeup from Deep Sleep Mode" must be checked in the SPI configuration dialog. The SCB_Sleep() and SCB_Wakeup() functions must be called before/after entering/exiting Deep Sleep.

In **Slave** mode operation, the device wakes up from Deep Sleep on detecting slave select activation. Waking up takes time and the ongoing SPI transfer is negatively acknowledged –

"0xff" bytes are sent out on the MISO line. The Master must poll the component again after the device wake-up time is passed.

Slave data rate calculations

The SPI GUI calculates the actual data rate for master or slave devices. This value is based on the parameters of the component and does not take to account such factors as: parameters of external master or slave device as well as PCB delays. The master and slave parameters for PSoC4 can be found in the DC and AC Electrical Characteristics of this document or Device datasheet.

The main factor limiting the maximum data rate between master and slave is the round trip path delay. This delay includes the PCB delay from the falling edge of SCLK at the pin of the master to the SCLK pin of the slave, the internal slave delay from the falling edge of SCLK to MISO transition, the PCB delay from the slave MISO pin to the master MISO pin, and the master setup time. The following equation takes to account delays listed above:

$$t_{ROUND_TRIP_DELAY} = t_{SCLK_PD_PCB} + t_{DSO_SLAVE} + t_{MISO_PD_PCB} + t_{DSI_MASTER}$$

$t_{SCLK_PD_PCB}$ is the PCB path delay of SCLK from the pin of the master device to the pin of the slave device.

t_{DSO_SLAVE} is the time it takes the slave to change MISO after SCLK clock driving edge is captured. This parameter commonly listed in the slave device datasheet.

$t_{MISO_PD_PCB}$ is the PCB path delay of MISO from the pin of the slave device to the pin of the master device.

t_{DSI_MASTER} is the setup time of MISO signal to be sampled correctly by the master (the MISO must be valid before SCLK clock capturing edge). This parameter commonly listed in the master device datasheet.

When $t_{ROUND_TRIP_DELAY}$ was calculated the maximum communication data rate between master and slave can be defined as following:

$$f_{SCLK}(max) = 1 / (2 * t_{ROUND_TRIP_DELAY})$$

The assumption is made that master samples the MISO signal a half SCLK period after the driving edge.

When master is capable of sampling the MISO signal a full of SCLK period after the driving edge (late MISO sampling) the communication data rate is doubled and calculated as following:

$$f_{SCLK}(max) = 1 / t_{ROUND_TRIP_DELAY}$$

Refer to the section MISO late sampling for more information about MISO sampling by the master device.

As an example the f_{SCLK} (max) is calculated for SCB SPI Master and Slave implemented on PSoC 4100/PSoC 4200 devices. The design clock settings are following: IMO = HFCLK = SYSCLK = 48 MHz. The clock source frequency connected to the SCB SPI Slave and Master components is equal to 48MHz as well.

$t_{DSO_SLAVE} = T_{DSO} = 42 + 3*t_{SCB} = 42 + 3 * (1 / 48\ MHz) = 105$ ns

$t_{DSI_MASTER} = T_{DSI} = 20$ ns (Full clock, late MISO Sampling used)

For simplicity of the calculations assume that $t_{SCLK_PD_PCB} = 0$ ns and $t_{MISO_PD_PCB} = 0$ ns.

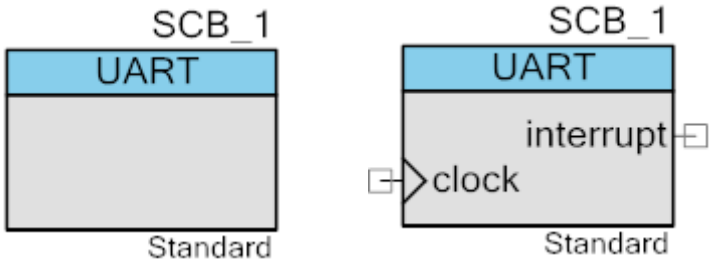
$t_{ROUND_TRIP_DELAY} = t_{SCLK_PD_PCB} + t_{DSO_SLAVE} + t_{MISO_PD_PCB} + t_{DSI_MASTER} = 0 + 105 + 20 + 0 = 125$ ns

f_{SCLK} (max) = 1 / $t_{ROUND_TRIP_DELAY}$ = 1 / 125 ns = 8 MHz

The SPI master is capable to generate maximum $F_{SPI} = 8$ MHz and accordingly to calculation above the MISO line will be sampled properly for this data rate.

For real applications the PCB delays would need to be added, and t_{DSO_SLAVE} and t_{DSI_MASTER} adjusted to match the real master or slave device.

UART



The UART provides asynchronous communications commonly referred to as RS-232. Three different UART-like serial interface protocols are supported:

- UART – this is the standard UART.
  - UART Hardware flow control
- SmartCard – similar to UART, but with the possibility to send a negative acknowledgement.
- IrDA – modification to the modulation scheme used for infrared communication.

Input/Output Connections

This section describes the various input and output connections for the SCB component. An asterisk (*) in the list of terminals indicates that the terminals may be hidden on the symbol under the conditions listed in the description of that terminals.

clock – Input*

Clock that operates this block. The presence of this terminal varies depending on the **Clock from terminal** parameter.

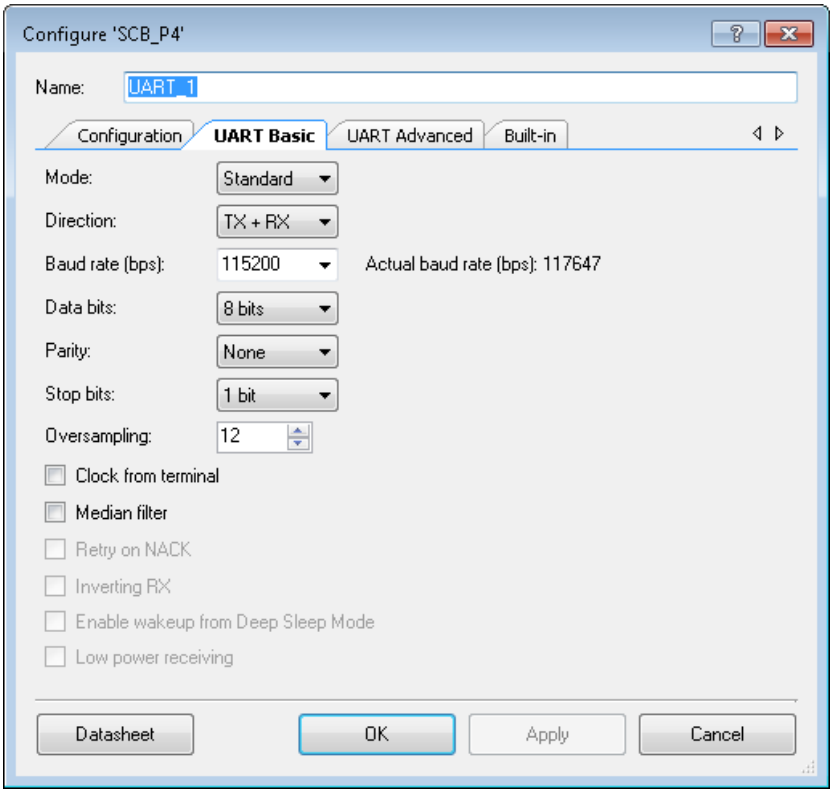
interrupt – Output*

This signal can only be connected to an interrupt component or left unconnected. The presence of this terminal varies depending on the **Interrupt** parameter.

The interface-specific pins are buried inside component because these pins use dedicated connections and are not routable as general purpose signals. See the *I/O System* section in the device *Technical Reference Manual (TRM)* for more information.

Note The input buffer of buried output pins is disabled so as not to cause current linkage in low power mode. Reading the status of these pins always returns zero. To get the current status, the input buffer must be enabled before a status read.

Basic UART Parameters



The **UART Basic** tab contains the following parameters:

Mode

This option determines the operating mode of the UART: Standard, SmartCard or IrDA. The default mode is **Standard**.

Direction

This parameter defines the functional components you want to include in the UART. This can be setup to be a bidirectional **TX + RX** (default), Receiver (**RX only**) or Transmitter (**TX only**).

Baud rate

This parameter defines the baud-rate configuration of the hardware for baud rate generation up to 921600. The actual baud rate may differ based on available clock frequency and component settings. This parameter has no effect if the **Clock from terminal** parameter is enabled. The default is 115200.

Note The integer clock divider is used to provide the desired internal clock frequency to obtain the specified baud rate (Clock from terminal option is disabled). To use a different clock source configuration (for example: fractional clock divider), the clock must be provided externally to the component by enabling the Clock from terminal option.

Actual baud rate

The actual data rate displays the data rate at which the component will operate with current settings. The factors that affect the actual data rate calculation are: the accuracy of the component clock (internal or external) and oversampling factor. When a change is made to any of the component parameters that affect actual data rate, it becomes unknown. To calculate the new actual data rate press the Apply button

Data bits

This parameter defines the number of data bits transmitted between start and stop of a single UART transaction. Options are **5**, **6**, **7**, **8** (default), or **9**.

- Eight data bits is the default configuration, sending a byte per transfer.
- The 9-bit mode does not transmit 9 data bits; the ninth bit takes the place of the parity bit as an indicator of address or data.

Parity

This parameter defines the functionality of the parity bit location in the transfer. This can be set to **None** (default), **Odd** or **Even**.

Stop bits

This parameter defines the number of stop bits implemented in the transmitter. This parameter can be set to **1** (default), **1.5** or **2** data bits.

Oversampling

This parameter defines the oversampling factor of the UART interface; the number of the component clocks within one UART bit time. Oversampling factor is used to calculate the internal component clock frequency required to achieve this amount of oversampling for the selected Data rate. An oversampling factor between 8 and 16 is the range of valid values. The default is 12.

For **IrDA** mode the oversampling values are predefined and Median filter is always enabled.

Clock from terminal

This parameter allows choosing between an internally configured clock (by the component) or an externally configured clock (by the user) for component operation. Refer to the Oversampling section to understand relationship between component clock frequency and the component parameters.

When this option is enabled the component does not control the data rate, but displays the actual data rate based on the user-connected clock source frequency and the component oversampling factor. When this option is not enabled the clock configuration is provided by the component. The clock source frequency is calculated or selected by the component based on the Data rate parameter and Oversampling factor.

Note PSoC Creator is responsible for providing requested clock frequency (internal or external clock) based on current design clock configuration. When the requested clock frequency with requested tolerance cannot be created, a warning about the clock accuracy range is generated while building the project. This warning contains the actual clock frequency value created by PSoC Creator. To remove this warning you must either change the system clock, component settings or external clock to fit the clocking system requirements.

Median filter

This parameter applies 3 taps digital median filter on input path of RX line. This filter reduces the susceptibility to errors. The default value is a **Disabled**.

Retry on NACK

This option is applicable only for **SmartCard** mode. It enables retry on NACK feature. The Data frame is retransmitted when a negative acknowledgement is received.

Inverting RX

This option is applicable only for **IrDA** mode. It enables the inversion of the incoming RX line signal.

Enable wakeup from Deep Sleep Mode

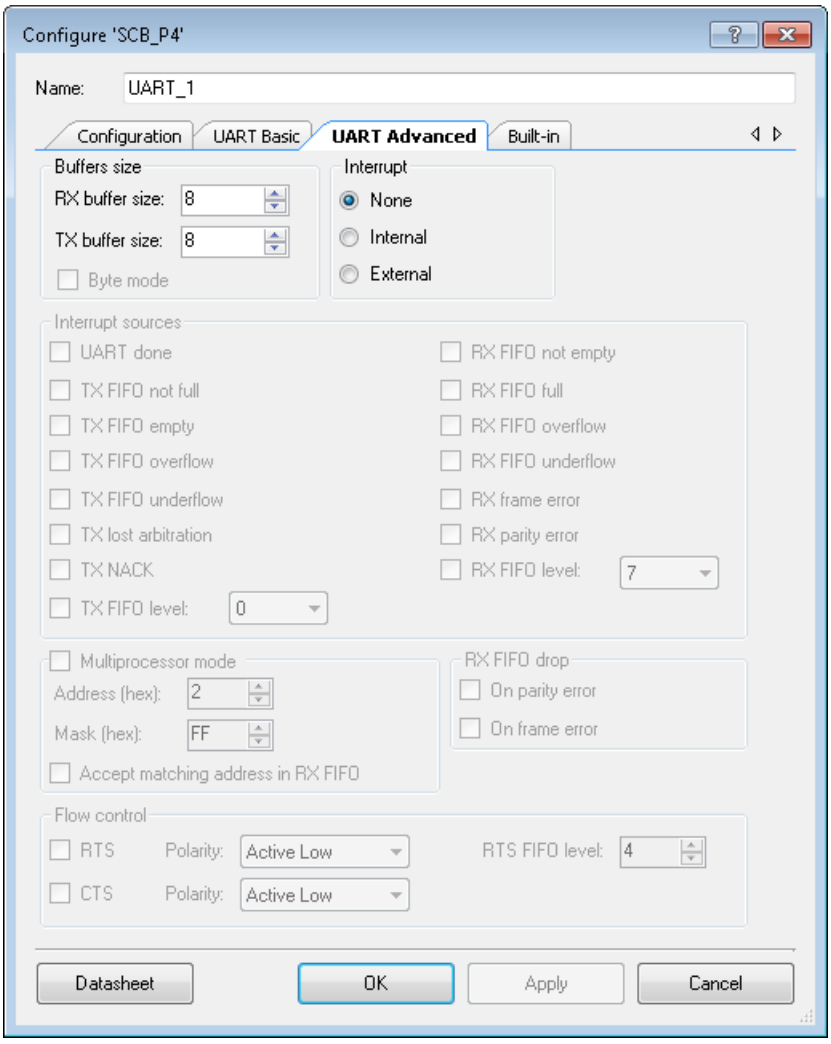
Use this option to enable the component to wake the system from Deep Sleep on the start bit. It is applicable for Standard mode when **RX Direction** is enabled.

Refer to the Low power modes section under UART chapter in this document and *Power Management APIs* section of the *System Reference Guide* for more information.

Low power receiving

This option is applicable only when **RX Direction** is enabled. It enables IrDA low power receiver mode.

Advanced UART Parameters



The **UART Advanced** tab contains the following parameters:

RX buffer size

The **RX buffer size** parameter defines the size (in bytes/words) of memory allocated for a receive data buffer. The RX buffer size minimum value is equal to the RX FIFO depth. The RX FIFO is implemented in hardware. Values greater than the RX FIFO depth up to (2^{32} – 2) imply usage of the RX FIFO, a circular software buffer controlled by the supplied APIs, and internal ISR. The software buffer size is limited only by the available memory. The interrupt mode is automatically set to internal and the RX FIFO not empty interrupt source is reserved if a software buffer is used.

- For 4100/PSoC 4200 devices, the RX and TX FIFO depth is equal to 8 bytes/words.
- For PSoC 4100 BLE/PSoC 4200 BLE devices, the RX and TX FIFO depth is equal to 8 bytes/words or 16 bytes; refer to **Byte mode** for more information.

TX buffer size

The **TX buffer size** parameter defines the size (in bytes/words) of memory allocated for a transmit data buffer. The TX buffer size minimum value is equal to the TX FIFO depth. The TX FIFO is implemented in hardware. Values greater than the TX FIFO depth up to (2^{32} – 1) imply usage of the TX FIFO, a circular software buffer controlled by the supplied APIs, and internal ISR. The software buffer size is limited only by the available memory. The interrupt mode is automatically set to the internal and the TX FIFO not full interrupt source is reserved if a software buffer is used.

- For 4100/PSoC 4200 devices, the RX and TX FIFO depth is equal to 8 bytes/words.
- For PSoC 4100 BLE/PSoC 4200 BLE devices, the RX and TX FIFO depth is equal to 8 bytes/words or 16 bytes; refer to **Byte mode** for more information.

Byte mode

This option is only applicable for PSoC 4100 BLE/PSoC 4200 BLE devices. It allows doubling the TX and RX FIFO depth from 8 to 16 bytes. This implies that the number of data bits must be less than or equal to 8 bits. Increasing the FIFO depth improves performance of UART operation as more bytes can be transmitted or received without software interaction.

Interrupt

This option determines what interrupt modes are supported None, Internal or External.

- **None** – This option removes the internal interrupt component.
- **Internal** – This option leaves the interrupt component inside the SCB component. The predefined internal interrupt handler is hooked up to the interrupt. The **Interrupt sources** option sets one or more interrupt sources, which trigger the interrupt. To add your own code to the interrupt service routine you need to register a function using the SCB_SetCustomInterruptHandler() function.

- **External** – This option removes the internal interrupt and provides an output terminal. Only an interrupt component can be connected to the terminal if an interrupt handler is desired. The **Interrupt sources** option sets one or more interrupt sources, which trigger the interrupt output.

Note For buffer sizes greater than the hardware FIFO depth, the component automatically enables the internal interrupt sources required for proper internal software buffer operations. In addition, the global interrupt enable must be explicitly enabled for proper buffer handling.

Interrupt sources

The interrupt sources are either level or pulse. Level interrupt sources in the following list are indicated with an asterisk (*). Refer to sections TX FIFO interrupt sources and RX FIFO interrupt sources for more information about level interrupt sources operation. The UART supports interrupts on the following events:

- **UART done –** UART transmitter done event: all data elements from the TX FIFO are sent. This interrupt source triggers later than TX FIFO empty by time it takes to transmit a single data element. The TX FIFO empty triggers when the last data element from the TX FIFO goes to the shifter register. However UART done triggers after this data element has been transmitted.
- **TX FIFO not full * –** TX FIFO is not full. At least one data element can be written into the TX FIFO.
- **TX FIFO empty * –** TX FIFO is empty.
- **TX FIFO overflow –** Firmware attempts to write to a full TX FIFO.
- **TX FIFO underflow –** Hardware attempts to read from an empty TX FIFO.
- **TX lost arbitration –** UART lost arbitration: the value driven on the TX line is not the same as the value observed on the RX line. This condition event is useful when transmitter and receiver share a TX/RX line. This is the case in SmartCard mode.
- **TX NACK –** UART transmitter received a negative acknowledgement in SmartCard mode.
- **TX FIFO level * –** When the TX FIFO has less entries than the amount of this field, an interrupt request is generated.
- **RX FIFO not empty * –** RX FIFO is not empty. At least one data element is available in the RX FIFO to be read.
- **RX FIFO full * –** RX FIFO is full.
- **RX FIFO overflow** – Hardware attempts to write to a full RX FIFO.
- **RX FIFO underflow** – Firmware attempts to read from an empty RX FIFO.

- **RX frame error** – Frame error in received data frame. This can be either a start or stop bit(s) error:
  - Start bit error – after the detection of the beginning of a start bit period (RX line changes from '1' to '0'), the middle of the start bit period is sampled erroneously (RX line is '1').

    Note A start bit error is detected BEFORE a data frame is received.
  - Stop bit error: the RX line is sampled as '0', but a '1' was expected.

    Note A stop bit error may result in failure to receive successive data frame(s). A stop bit error is detected AFTER a data frame is received.

    Note For stop bit duration equal to 1bit, the frame error is not tracked.
- **RX parity error –** Parity error in received data frame.
- **RX FIFO level * –** When the RX FIFO has more entries than the number of this field, an interrupt request is generated.

Notes

When **RX buffer size** is greater than the RX FIFO depth, the **RX FIFO not empty** interrupt source is reserved by the component and used for the internal interrupt.

When **TX buffer size** is greater than the TX FIFO depth, the **TX FIFO not full** interrupt source is reserved by the component and used for the internal interrupt.

Multiprocessor mode

This parameter enables the multiprocessor mode where the 9th bit (in the place of the parity bit) indicates an address. The default value is a **Disabled**. The number of Data bits must be set to 9 bits to get possibility to enable this option.

Address (hex)

Slave device address. Used to match when multiprocessor mode is enabled. The default value is 0x02.

Mask (hex)

Slave device address mask. These bits are used when matching to the slave address. The default value is **0xFF**.

- Bit value 0 – excludes bit from address comparison.
- Bit value 1 – the bit needs to match with the corresponding bit of the address.

Accept matching address in RX FIFO

This parameter determines whether to accept a matched address in the RX FIFO.

Note Non-matching addresses are never put in the RX FIFO.

RX FIFO drop

Provides hardware data drop options for RX FIFO.

- **On parity error** – Defines behavior when a parity check fails. When parity check is passed, received data is sent to the RX FIFO. Otherwise, received data is dropped and lost. Only applicable in **Standard** and **SmartCard** modes.
- **On frame error** – Defines behavior when a frame error is detected. When no frame error is captured, received data is sent to the RX FIFO. Otherwise, received data is dropped and lost.

RTS

This parameter is only applicable for PSoC 4100 BLE/PSoC 4200 BLE devices. It enables the Ready to Send (RTS) output signal. The RTS signal is the part of flow control functionality used by the receiver. As long as the receiver is ready to accept more data it will keep the RTS signal active. The RTS FIFO level parameter determines if RTS remains active. The default value is a **Disabled**.

RTS Polarity

This parameter defines active polarity of the RTS output signal as Active Low (default) or Active High.

RTS FIFO level

This parameter is only available for PSoC 4100 BLE/PSoC 4200 BLE devices. It determines whether the RTS signal remains active. While the RX FIFO has fewer entries than the RTS FIFO level, the RTS signal remains active; otherwise, the RTS signal becomes inactive. The RTS remains inactive unit data from RX FIFO will be read to match RTS FIFO level. The default value is 4.

CTS

This parameter is only applicable for PSoC 4100 BLE/PSoC 4200 BLE devices. It enables the Clear to Send (CTS) input signal to be routed-out to the pin. The CTS signal is the part of flow control functionality utilized by the transmitter. The transmitter checks whether CTS signal is active before sending data from the TX FIFO. The transmission of data is suspended if CTS signal is inactive and will be resumed when CTS signal becomes active again. The default value is a **Disabled**.

CTS Polarity

This parameter defines active polarity of CTS input signal as Active Low (default) or Active High.

UART APIs

APIs allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections discusses each function in more detail.

By default, PSoC Creator assigns the instance name “SCB_1” to the first instance of a component in a given design. You can rename the instance to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “SCB”.

Function	Description
SCB_Start()	Starts the SCB.
SCB_Init()	Initialize the SCB component according to defined parameters in the customizer.
SCB_Enable()	Enables SCB component operation.
SCB_Stop()	Disable the SCB component.
SCB_Sleep()	Prepares component to enter Deep Sleep.
SCB_Wakeup()	Prepares component for Active mode operation after Deep Sleep.
SCB_UartInit()	Configures the SCB for UART operation. Only used when using the SCB in unconfigured mode.
SCB_UartPutChar()	Places a byte of data in the transmit buffer to be sent at the next available bus time.
SCB_UartPutString()	Places a NULL terminated string in the transmit buffer to be sent at the next available bus time.
SCB_UartPutCRLF()	Places byte of data followed by a carriage return (0x0D) and line feed (0x0A) to the transmit buffer
SCB_UartGetChar()	Retrieves next data element from receive buffer.
SCB_UartGetByte()	Retrieves next data element from the receive buffer.
SCB_UartSetRxAddress()	Sets the hardware detectable receiver address for the UART in Multiprocessor mode.
SCB_UartSetRxAddressMask()	Sets the hardware address mask for the UART in Multiprocessor mode.
SCB_UartSetRtsPolarity()	Sets active polarity of RTS input signal.
SCB_UartSetRtsFifoLevel()	Sets level in the RX FIFO to activate RTS signal.
SCB_UartEnableCts()	Enables usage of CTS input signal by the UART transmitter.

Function	Description
SCB_UartDisableCts()	Disables usage of CTS input signal by the UART transmitter
SCB_UartSetCtsPolarity()	Sets active polarity of CTS input signal.
SCB_SpiUartWriteTxData()	Places a data entry into the transmit buffer to be sent at the next available bus time.
SCB_SpiUartPutArray()	Places an array of data into the transmit buffer to be sent.
SCB_SpiUartGetTxBufferSize()	Returns the number of elements currently in the transmit buffer.
SCB_SpiUartClearTxBuffer()	Clears the transmit buffer and TX FIFO.
SCB_SpiUartReadRxData()	Retrieves the next data element from the receive buffer.
SCB_SpiUartGetRxBufferSize()	Returns the number of received data elements in the receive buffer.
SCB_SpiUartClearRxBuffer()	Clears the receive buffer and RX FIFO.

void SCB_Start(void)

Description: Invokes SCB_Init() and SCB_Enable(). After this function call the component is enabled and ready for operation. This is the preferred method to begin component operation.

When configuration is set to “Unconfigured SCB”, the component must first be initialized to operate in one of the following configurations: I^2C, SPI, UART or EZ I^2C. Otherwise this function does not enable component.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Init(void)

Description: Initializes the SCB component to operate in one of the selected configurations: I^2C, SPI, UART or EZ I^2C.

When configuration is set to “Unconfigured SCB”, this function does not do any initialization. Use mode-specific initialization functions instead: SCB_I2CInit, SCB_SpiInit, SCB_UartInit or SCB_EzI2CInit.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Enable(void)

Description: Enables SCB component operation: activates the hardware and internal interrupt.

For I^2C and EZ I^2C modes the interrupt is internal and mandatory for operation. For SPI and UART modes the interrupt can be configured as none, internal or external. The SCB configuration should be not changed when the component is enabled. Any configuration changes should be made after disabling the component.

When configuration is set to “Unconfigured SCB”, the component must first be initialized to operate in one of the following configurations: I^2C, SPI, UART or EZ I^2C using the mode-specific functions: SCB_I2CInit, SCB_SpiInit, SCB_UartInit or SCB_EzI2CInit. Otherwise this function does not enable component.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Stop(void)

Description: Disables the SCB component: disable the hardware and internal interrupt. Refer to the function SCB_Enable() for the interrupt configuration details.

This function disables the SCB component without checking to see if communication is in progress. Before calling this function it may be necessary to check the status of communication to make sure communication is complete. If this is not done then communication could be stopped mid byte and corrupted data could result.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Sleep(void)

Description: Prepares component to enter Deep Sleep.

The “Enable wakeup from Deep Sleep Mode” selection has an influence on this function implementation:

- Checked: configures the component to be wakeup source from Deep Sleep.
- Unchecked: stores the current component state (enabled or disabled) and disables the component. See SCB_Stop() function for details about component disabling.

Call the SCB_Sleep() function before calling the CyPmSysDeepSleep() function.

Refer to the PSoC Creator *System Reference Guide* for more information about power-management functions.

This function should not be called before entering Sleep.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Wakeup(void)

Description: Prepares component for Active mode operation after Deep Sleep.

The “Enable wakeup from Deep Sleep Mode” selection has influence to on this function implementation:

- Checked: restores the component Active mode configuration.
- Unchecked: enables the component if it was enabled before enter Deep Sleep.

This function should not be called after exiting Sleep.

Parameters: None

Return Value: None

Side Effects: Calling the SCB_Wakeup() function without first calling the SCB_Sleep() function may produce unexpected behavior.

void SCB_UartInit(SCB_UART_INIT_STRUCT *config)

Description: Configures the SCB for UART operation.

This function is **intended specifically** to be used when the SCB configuration is set to "Unconfigured SCB" in the customizer. After initializing the SCB in UART mode, the component can be enabled using the SCB_Start() or SCB_Enable() function.

This function uses a pointer to a structure that provides the configuration settings. This structure contains the same information that would otherwise be provided by the customizer settings.

Parameters: config: pointer to a structure that contains the following ordered list of fields. These fields match the selections available in the customizer. Refer to the customizer for further description of the settings.

Field	Description
uint32 mode	Mode of operation for the UART. The following defines are available choices: SCB_UART_MODE_STD SCB_UART_MODE_SMARTCARD SCB_UART_MODE_IRDA
uint32 direction	Direction of operation for the UART. The following defines are available choices: SCB_UART_TX_RX SCB_UART_RX SCB_UART_TX
uint32 dataBits	Number of data bits
uint32 parity	Determines the parity. The following defines are available choices: SCB_UART_PARITY_EVEN SCB_UART_PARITY_ODD SCB_UART_PARITY_NONE
uint32 stopBits	Determines the number of stop bits. The following defines are available choices: SCB_UART_STOP_BITS_1 SCB_UART_STOP_BITS_1_5 SCB_UART_STOP_BITS_2
uint32 oversample	Oversampling factor for the UART. **Note** The oversampling factor values are changed when enableIrdaLowPower is enabled: SCB_UART_IRDA_LP_OVS16 SCB_UART_IRDA_LP_OVS32 SCB_UART_IRDA_LP_OVS48 SCB_UART_IRDA_LP_OVS96 SCB_UART_IRDA_LP_OVS192 SCB_UART_IRDA_LP_OVS768 SCB_UART_IRDA_LP_OVS1536
uint32 enableIrdaLowPower	IrDA low power RX mode is enabled. 0 – disable 1 – enable The TX functionality does not work when enabled.

uint32 enableMedianFilter	0 – disable 1 – enable
uint32 enableRetryNack	0 – disable 1 – enable Ignored for modes other than SmartCard.
uint32 enableInvertedRx	0 – disable 1 – enable Ignored for modes other than IrDA.
uint32 dropOnParityErr	Drop data from RX FIFO if parity error is detected. 0 – disable 1 – enable
uint32 dropOnFrameErr	Drop data from RX FIFO if a frame error is detected. 0 – disable 1 – enable
uint32 enableWake	0 – disable 1 – enable Ignored for modes other than standard UART. The RX functionality has to be enabled.
uint32 rxBufferSize	Size of the RX buffer in words: • The value equal to the RX FIFO depth implies the usage of buffering in hardware. • A value greater than the RX FIFO depth results in a software buffer. The SCB_INTR _RX_NOT_EMPTY interrupt has to be enabled to transfer data into the software buffer. For PSoC 4100/PSoC 4200 devices, the RX FIFO and TX FIFO depth is equal to 8 bytes/words. For PSoC 4100 BLE/PSoC 4200 BLE devices, the RX FIFO and TX FIFO depth is equal to 8 bytes/words or 16 bytes (Byte mode is enabled).
uint8* rxBuffer	Buffer space provided for a RX software buffer: • A NULL pointer must be provided to use hardware buffering. • A pointer to an allocated buffer must be provided to use software buffering. The buffer size must equal (rxBufferSize + 1) in bytes if dataBits is less or equal to 8, otherwise (2 * (rxBufferSize + 1)) in bytes. The software RX buffer always keeps one element empty. For correct operation allocated RX buffer has to be one element greater than maximum packet size expected to be received.
uint32 txBufferSize	Size of the TX buffer in words: • The value equal to the RX FIFO depth implies the usage of buffering in hardware. • A value greater than the RX FIFO depth results in a software buffer. For 4100/PSoC 4200 devices, the RX and TX FIFO depth is equal to 8 bytes/words. For PSoC 4100 BLE/PSoC 4200 BLE devices, the RX and TX FIFO depth is equal to 8 bytes/words or 16 bytes (Byte mode is enabled).

uint8* txBuffer	Buffer space provided for a TX software buffer: • A NULL pointer must be provided to use hardware buffering. • A pointer to an allocated buffer must be provided to use software buffering. The buffer size must equal txBufferSize if dataBits is less or equal to 8, otherwise (2* rxBufferSize).
uint32 enableMultiproc	Enables multiprocessor mode. 0 – disable 1 – enable
uint32 multiprocAcceptAddr	Enables matched address to be accepted. 0 – disable 1 – enable
uint32 multiprocAddr	8 bit address to match in Multiprocessor mode. Ignored for other modes.
uint32 multiprocAddrMask	8 bit mask of address bits that are compared for a Multiprocessor address match. Ignored for other modes.
uint32 enableInterrupt	0 – disable 1 – enable The interrupt has to be enabled if software buffer will be used.
uint32 rxInterruptMask	Mask of interrupt sources to enable in the RX direction. This mask is written regardless of the setting of the enableInterrupt field. Multiple sources are enabled by providing a value that is the OR of all of the following sources to enable: SCB_INTR_RX_FIFO_LEVEL SCB_INTR_RX_NOT_EMPTY SCB_INTR_RX_FULL SCB_INTR_RX_OVERFLOW SCB_INTR_RX_UNDERFLOW SCB_INTR_RX_FRAME_ERROR SCB_INTR_RX_PARITY_ERROR
uint32 rxTriggerLevel	FIFO level for an RX FIFO level interrupt. This value is written regardless of whether the RX FIFO level interrupt source is enabled.
uint32 txInterruptMask	Mask of interrupt sources to enable in the TX direction. This mask is written regardless of the setting of the enableInterrupt field. Multiple sources are enabled by providing a value that is the OR of all of the following sources to enable: SCB_INTR_TX_FIFO_LEVEL SCB_INTR_TX_NOT_FULL SCB_INTR_TX_EMPTY SCB_INTR_TX_OVERFLOW SCB_INTR_TX_UNDERFLOW SCB_INTR_TX_UART_DONE SCB_INTR_TX_UART_NACK SCB_INTR_TX_UART_ARB_LOST
uint32 txTriggerLevel	FIFO level for a TX FIFO level interrupt. This value is written regardless of whether the TX FIFO level interrupt source is enabled.

uint8 enableByteMode	Ignored for devices other than PSoC 4100 BLE/PSoC 4200 BLE. 0 – disable 1 – enable When enabled the TX and RX FIFO depth is 16 bytes. This implies that number of Data bits must be less than or equal to 8.
uint8 enableCts	Ignored for all devices other than PSoC 4100 BLE/PSoC 4200 BLE. Enables usage of CTS input signal by the UART transmitter. 0 – disable 1 – enable
uint8 ctsPolarity	Ignored for all devices other than PSoC 4100 BLE/PSoC 4200 BLE. Sets active polarity of CTS input signal. SCB_UART_CTS_ACTIVE_LOW SCB_UART_CTS_ACTIVE_HIGH
uint8 rtsRxFifoLevel	Ignored for all devices other than PSoC 4100 BLE/PSoC 4200 BLE. RX FIFO level for RTS signal activation. While the RX FIFO has fewer entries than the RTS FIFO level value the RTS signal remains active, otherwise the RTS signal becomes inactive. By setting this field to 0, RTS signal activation is disabled.
uint8 rtsPolarity	Ignored for all devices other than PSoC 4100 BLE/PSoC 4200 BLE. Sets active polarity of RTS output signal. SCB_UART_RTS_ ACTIVE_LOW SCB_UART_RTS_ACTIVE_HIGH

Return Value: None

Side Effects: None

void SCB_UartPutChar(uint32 txDataByte)

Description: Places a byte of data in the transmit buffer to be sent at the next available bus time. This function is blocking and waits until there is a space available to put requested data in the transmit buffer.

For UART Multi Processor mode this function can send 9-bits data as well. Use SCB_UART_MP_MARK to add a mark to create an address byte.

Note This function is implemented as macro which calls SCB_SpiUartWriteTxData().

Parameters: uint32 txDataByte: the data to be transmitted.

Return Value: None

Side Effects: None

void SCB_UartPutString(const char8 string[])

Description: Places a NULL terminated string in the transmit buffer to be sent at the next available bus time.

This function is blocking and waits until there is a space available to put requested data in transmit buffer.

Parameters: const char8 string[]: pointer to the null terminated string array to be placed in the transmit buffer.

Return Value: None

Side Effects: None

void SCB_UartPutCRLF(uint32 txDataByte)

Description: Places byte of data followed by a carriage return (0x0D) and line feed (0x0A) in the transmit buffer

This function is blocking and waits until there is a space available to put all requested data in transmit buffer.

Parameters: uint32 txDataByte : the data to be transmitted

Return Value: None

Side Effects: None

uint32 SCB_UartGetChar(void)

Description: Retrieves next data element from receive buffer. This function is designed for ASCII characters and returns a char where 1 to 255 are valid characters and 0 indicates an error occurred or no data is present.

RX software buffer is disabled: Returns data element retrieved from RX FIFO.

RX software buffer is enabled: Returns data element from the software receive buffer.

Parameters: None

Return Value: uint32: Next data element from the receive buffer. ASCII character values from 1 to 255 are valid. A returned zero signifies an error condition or no data available.

Side Effects: The errors bits may not correspond with reading characters due to RX FIFO and software buffer usage.

RX software buffer is enabled: The internal software buffer overflow is not treated as an error condition. Check SCB_rxBufferOverflow to capture that error condition.

uint32 SCB_UartGetByte(void)

Description: Retrieves next data element from the receive buffer, returns received byte and error condition.

RX software buffer disabled: Returns data element retrieved from RX FIFO. Undefined data will be returned if the RX FIFO is empty

RX software buffer enabled: Returns data element from the software receive buffer

Parameters: None

Return Value: uint32: Bits 7-0 contain the next data element from the receive buffer and other bits contain the error condition. The error condition constants are provided below:

RX error conditions	Description
SCB_UART_RX_OVERFLOW	Attempt to write to a full receiver FIFO.
SCB_UART_RX_UNDERFLOW	Attempt to read from an empty receiver FIFO.
SCB_UART_RX_FRAME_ERROR	UART framing error detected.
SCB_UART_RX_PARITY_ERROR	UART parity error detected.

Side Effects: The errors bits may not correspond with reading characters due to RX FIFO and software buffer usage.

RX software buffer is disabled: Internal software buffer overflow is not returned as status by this function. Check SCB_rxBufferOverflow to capture that error condition.

void SCB_UartSetRxAddress(uint32 address)

Description: Sets the hardware detectable receiver address for the UART in Multiprocessor mode.

Parameters: uint32 address: Address for hardware address detection.

Return Value: None

Side Effects: None

void SCB_UartSetRxAddressMask(uint32 addressMask)

Description: Sets the hardware address mask for the UART in Multiprocessor mode.

Parameters: uint32 addressMask: Address mask.

Bit value 0 – excludes bit from address comparison.

Bit value 1 – the bit needs to match with the corresponding bit of the address.

Return Value: None

Side Effects: None

void SCB_UartSetRtsPolarity(uint32 polarity)

Description: Sets active polarity of RTS input signal.
Only available for PSoC 4100 BLE/PSoC 4200 BLE devices.

Parameters: uint32 polarity: Active polarity of RTS input signal.

Active RTS polarity constants	Description
SCB_UART_RTS_ACTIVE_LOW	RTS signal is active low
SCB_UART_RTS_ACTIVE_HIGH	RTS signal is active high

Return Value: None

Side Effects: None

void SCB_UartSetRtsFifoLevel (uint32 level)

Description: Sets level in the RX FIFO for RTS signal activation. While the RX FIFO has fewer entries than the RTS FIFO level the RTS signal remains active, otherwise the RTS signal becomes inactive.
Only available for PSoC 4100 BLE/PSoC 4200 BLE devices.

Parameters: uint32 level: Level in the RX FIFO for RTS signal activation.
The range of valid level values is between 0 and RX FIFO depth - 1. Setting level value to 0 disables RTS signal activation.

Return Value: None

Side Effects: None

void SCB_UartEnableCts(void)

Description: Enables usage of CTS input signal by the UART transmitter.
Only available for PSoC 4100 BLE/PSoC 4200 BLE devices.

Parameters: None

Return Value: None

Side Effects: None

void SCB_UartDisableCts(void)

Description: Disables usage of CTS input signal by the UART transmitter.
Only available for PSoC 4100 BLE/PSoC 4200 BLE devices.

Parameters: None

Return Value: None

Side Effects: None

void SCB_UartSetCtsPolarity(uint32 polarity)

Description: Sets active polarity of CTS input signal.

Only available for PSoC 4100 BLE/PSoC 4200 BLE devices.

Parameters: uint32 polarity: Active polarity of CTS input signal.

Active CTS polarity constants	Description
SCB_UART_CTS_ACTIVE_LOW	CTS signal is active low
SCB_UART_CTS_ACTIVE_HIGH	CTS signal is active high.

Return Value: None

Side Effects: None

void SCB_SpiUartWriteTxData(uint32 txData)

Description: Places a data entry into the transmit buffer to be sent at the next available bus time. The data transmit direction is LSB.

This function is blocking and waits until there is space available to put the requested data in the transmit buffer.

For UART Multi Processor mode this function can send 9-bits data. Use SCB_UART_MP_MARK to add a mark to create an address byte.

Parameters: uint32 txData: the data to be transmitted.

The amount of data bits to be transmitted depends on Data bits selection (the data bit counting starts from LSB of txDataByte).

Return Value: None

Side Effects: None

void SCB_SpiUartPutArray(const uint16/uint8 wrBuf[], uint32 count)

Description: Places an array of data into the transmit buffer to be sent.

This function is blocking and waits until there is a space available to put all the requested data in the transmit buffer.

The array size can be greater than transmit buffer size.

Parameters: const uint16/uint8 wrBuf[]: pointer to an array with data to be placed in transmit buffer. The amount of data bits to be transmitted as one array entry depends on Data bits selection (the data bit counting starts from LSB for each array entry).

uint32 count: number of data elements to be placed in the transmit buffer.

Return Value: None

Side Effects: None

uint32 SCB_SpiUartGetTxBufferSize(void)

Description: Returns the number of elements currently in the transmit buffer.

TX software buffer is disabled: Returns the number of used entries in TX FIFO.

TX software buffer is enabled: Returns the number of elements currently used in the transmit buffer. This number does not include used entries in the TX FIFO. The transmit buffer size is zero until the TX FIFO is full.

Parameters: None

Return Value: uint32: Number of data elements ready to transmit.

Side Effects: None

void SCB_SpiUartClearTxBuffer(void)

Description: Clears the transmit buffer and TX FIFO.

Parameters: None

Return Value: None

Side Effects: None

uint32 SCB_SpiUartReadRxData(void)

Description: Retrieves the next data element from the receive buffer.

RX software buffer is disabled: Returns data element retrieved from RX FIFO. Undefined data will be returned if the RX FIFO is empty.

RX software buffer is enabled: Returns data element from the software receive buffer. Zero value will be returned if receive software buffer is empty.

Parameters: None

Return Value: uint32: Next data element from the receive buffer.

The amount of data bits to be received depends on Data bits selection (the data bit counting starts from LSB of return value).

Side Effects: None

uint32 SCB_SpiUartGetRxBufferSize(void)

Description: Returns the number of received data elements in the receive buffer.

RX software buffer is disabled: Returns the number of used entries in RX FIFO.

RX software buffer is enabled: Returns the number of elements that were placed in the receive buffer. This does not include the hardware RX FIFO.

Parameters: None

Return Value: uint32: Number of received data elements

Side Effects: None

void SCB_SpiUartClearRxBuffer(void)

Description: Clears the receive buffer and RX FIFO.

Parameters: None

Return Value: None

Side Effects: None

Global Variables

Knowledge of these variables is not required for normal operations.

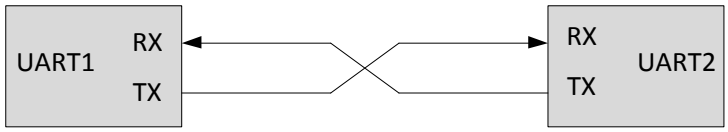
Variable	Description
SCB_initVar	SCB_initVar indicates whether the SCB component has been initialized. The variable is initialized to 0 and set to 1 the first time SCB_Start() is called. This allows the component to restart without reinitialization after the first call to the SCB_Start() routine. If re-initialization of the component is required, then the SCB_Init() function can be called before the SCB_Start() or SCB_Enable() function.
SCB_rxBufferOverflow	SCB_rxBufferOverflow sets when internal software receive buffer overflow was occurred.

UART Functional Description

The Universal Asynchronous Receiver/Transmitter (UART) protocol is an asynchronous serial interface. The UART transmit and receive interfaces consists of 2 signals:

- **TX** – Transmitter
- **RX** – Receiver

Figure 26. UART typical connection



Standard mode operation

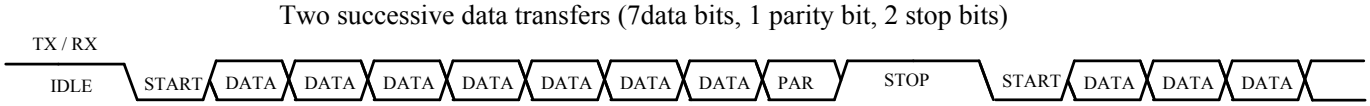
Standard UART is defined with “peer to peer” topology.

A typical UART transfer consists of a “Start Bit” followed by multiple “Data Bits”, optionally followed by a “Parity Bit” and finally completed by one or more “Stop Bits”. The “Start Bit” value is always ‘0’, the “Data Bits” values are dependent on the data transferred, the “Parity Bit” value is set to a value guaranteeing an even or odd parity over the “Data Bits” and the “Stop Bits” value is ‘1’. The “Parity Bit” is generated by the transmitter and can be used by the receiver to detect single bit transmission errors. When not transmitting data, the TX line is ‘1’; i.e. the same value as the “Stop Bits”.

The transition of a “Stop Bit” to a “Start Bit” is represented by a change from ‘1’ to ‘0’ on the TX line. This transition can be used by the receiver to synchronize with the transmitter clock. Synchronization at the start of each data transfer allows error-free transmission even in the presence of frequency drift between transmitter and receiver clocks. The required clock accuracy is dependent on the data transfer size.

The stop period or the amount of “Stop Bits” between successive data transfers is typically agreed upon between transmitter and receiver, and is typically in the range of 1 to 3 bit transfer periods.

Figure 27. UART Protocol

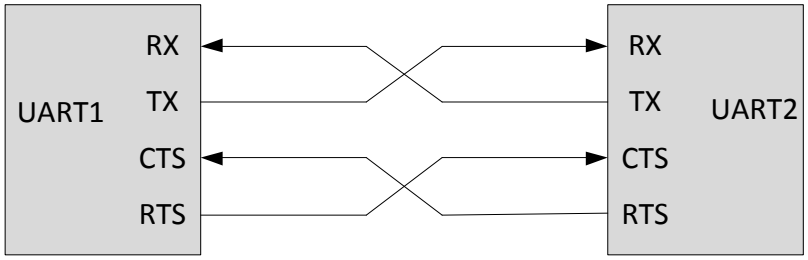


Flow control

Flow control is a method used to provide reliable communication between the receiver and transmitter without data loss. This method implies that a receiver tells a transmitter to stop

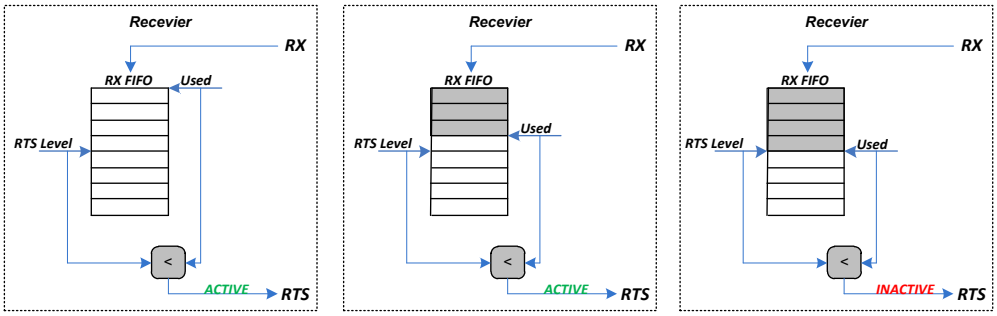
(suspend) or start (resume) transmitting. Hardware flow control is supported by the UART in Standard mode by PSoC 4100 BLE/PSoC 4200 BLE devices. The two extra lines for hardware flow control are needed in addition to data lines. They are called RTS and CTS. These lines are cross-coupled between the two devices, so the RTS line on one device is connected to the CTS line on the other device and vice versa. The Configure dialog provides independent control of RTS and CTS signals.

Figure 28. UART hardware flow control typical connection



As long as the receiver is ready to accept more data it will keep the RTS signal active. The RTS FIFO level parameter determines how the RTS remains active as follows: while the RX FIFO has fewer entries than the RX FIFO level the RTS signal remains active, otherwise the RTS signal becomes inactive. The RTS remains inactive unit data from RX FIFO is read to match RTS activation condition.

Figure 29. UART RTS signal activation



Initial state: RTS is active | Data reception: RTS is active | Data reception: RTS is inactive

The transmitter checks whether the CTS signal is active before sending data from the TX FIFO on the bus. The transmission of data is suspended if the CTS signal is inactive and will be resumed when CTS signal becomes active again.

Typically, the RTS and CTS signals are active low. However, there is a possibility to change the active polarity of these signals.

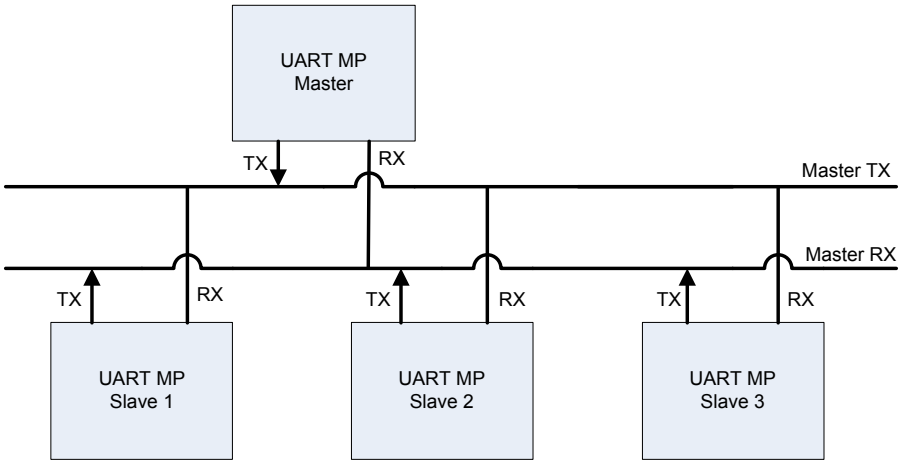
Multiprocessor mode operation

This mode is defined with “single-master-multi-slave” topology. The multiprocessor mode is also known as UART 9-bits protocol, while standard UART protocol uses a 5-bit to 8-bit data field.

The main properties of multiprocessor mode are:

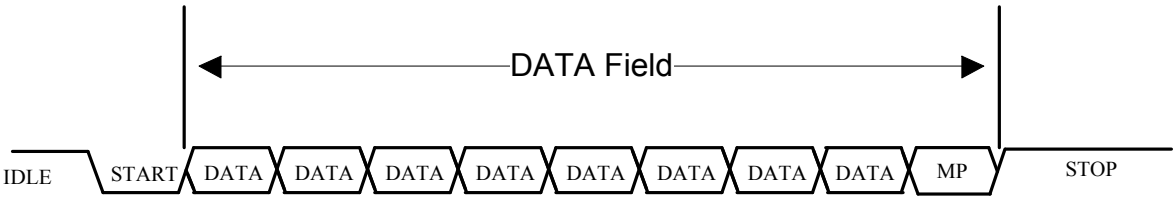
- Single master with multiple slave concept (multi-drop network)
- Each slave is identified by a unique address
- Using 9 bits data field, with the 9th bit (MSB) as address/data flag. When set '1', it indicates an address byte; when set '0' it indicates a data byte.
- Parity bit is disabled

Figure 30. Multiprocessor Bus Connections



To enable Multiprocessor mode, configure the UART with the following options: **Mode**: Standard, **Data bits**: 9 bits, **Parity**: None.

Figure 31. UART data frame in Multiprocessor mode



Because the data link layer of a multi-drop network is a user-defined protocol, it offers a flexible way of composing the data field.

All the bits in an address frame can be used to represent a device address. Alternatively, some bit can be used to represent the address, while the remaining bits can represent a command to the slave device, and some bits can represent the length of data in following data frames.

The SCB can be used as a master or slave device in multiprocessor mode.

When UART works as slave device, the received address is matched with Address and Mask. The matched address is written in the RX FIFO when Accept matching address in RX FIFO is checked. In the case of a match, subsequent received data are sent to the RX FIFO. In the case of no match, subsequent received data is dropped, until next address received for compare.

UART 9th data bit usage

The 9th bit is sent in the parity bit position and most typically used to define whether the data sent was an address or standard data. A mark (1) in the parity bit indicates an address was sent and a space (0) in the parity bit indicates data was sent. The data flow is "Start Bit, Data Bits, Parity, Stop Bits," similar to the other parity modes but this bit has to be controlled by user firmware before the transfer rather than being calculated based on the data bit values.

```
tx_data = 0x31;
tx_data |= UART_UART_MP_MARK; /* Set 9th bit to indicate address */
UART_SpiUartWriteTxData(tx_data);
```

SmartCard (ISO7816) mode operation

ISO7816 is an asynchronous serial interface, defined with “single-master-single-slave” topology. Only the master (reader) function is supported in the component.

SCB provides the basic physical layer support with asynchronous character transmission, and only “I/O” pin interface of standard ISO7816 [8] pin list is provided. SCB UART TX line will be connected to SmartCard I/O line, by internally multiplexing between TX and RX control modules.

The higher level protocol implementation is left for firmware to handle from the user level.

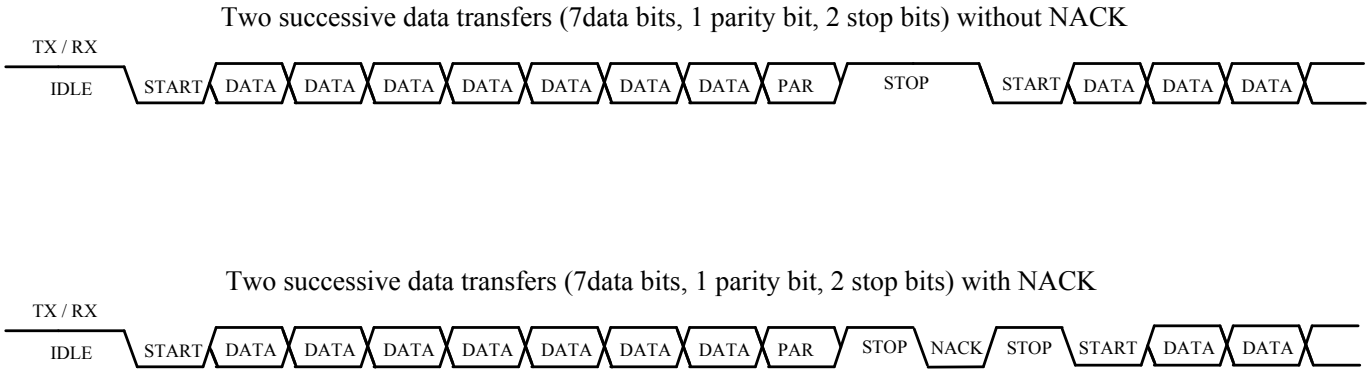
SmartCard data transfer

The SmartCard transfer is similar to a UART transfer, with the addition of a negative acknowledgement (NACK) that may be sent from the receiver to the transmitter. A NACK is always ‘0’. Both transmitter and receiver may drive the same I/O line, although never at the same time. Figure 32 illustrates the SmartCard protocol.

Typically, implementations use a tri-state driver with a pull-up resistor, such that when the line is not driven, its value is ‘1’ (the same value as when not transmitting data or the value of the “Stop Bit”).

[8] Refer to *the ISO/IEC 7816-3:2006 – Identification cards – Integrated circuit cards – Part 3: Cards with contacts – Electrical interface and transmission protocols* (1997) on the ISO web site at www.iso.org

Figure 32. SmartCard Data Transfer Example



A SmartCard transfer has the transmitter drive the “Start Bit” and “Data Bits” and a “Parity Bit”. After these bits, it enters its stop period by releasing the bus. Releasing results in the line being ‘1’ (the value of a “Stop Bit”). After half bit transfer period into the stop period, the receiver may drive a NACK on the line (a value of ‘0’) for one to two bit transfer period. This NACK is observed by the transmitter, which reacts by extending its stop period by one bit transfer period. For this protocol to work, the stop period should be larger than one bit transfer period.

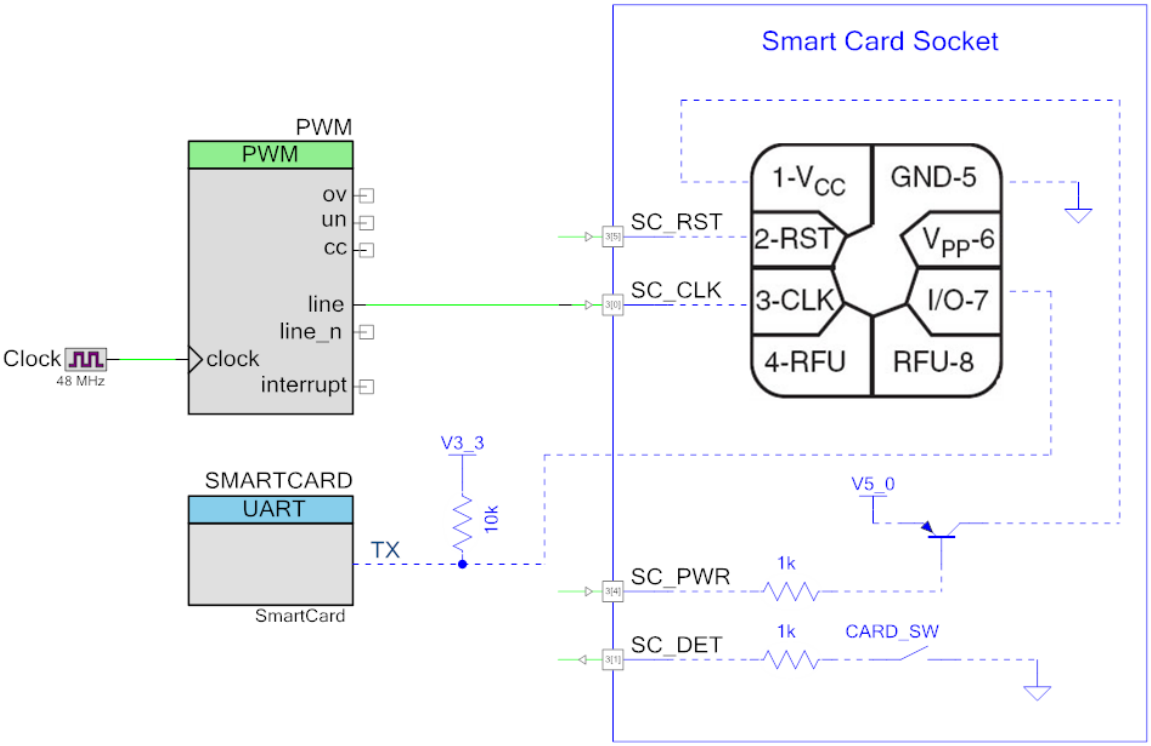
Note Data transfer with a NACK takes one bit transfer period longer than a data transfer without a NACK.

Example implementation of SmartCard reader

You have to consider how to implement a complete SmartCard system with other available system resources for “RST” signal, “CLK” signal, card detect signal, card power supply control signals.

Figure 33 is example of implementing SmartCard reader function with TCPWM and pins components.

Figure 33. SmartCard reader implementation example



The UART component is connected to I/O card contact, a pull-up resistor must be connected to this line. SC_RST, SC_CLK are standard card contacts. SC_DET is for card insertion detection. SC_PWR is for control of card power on or off. Refer to the *ISO7816 specification* for more details.

IrDA mode operation

IrDA is defined with “peer to peer” topology. SCB only provides support [9] for IrDA from the basic physical layer with rates from 1200 bps to 115200 bps. The physical layer is responsible for the definition of hardware transceivers for the data transmission. The higher level protocol implementation is left for firmware to handle from the user level.

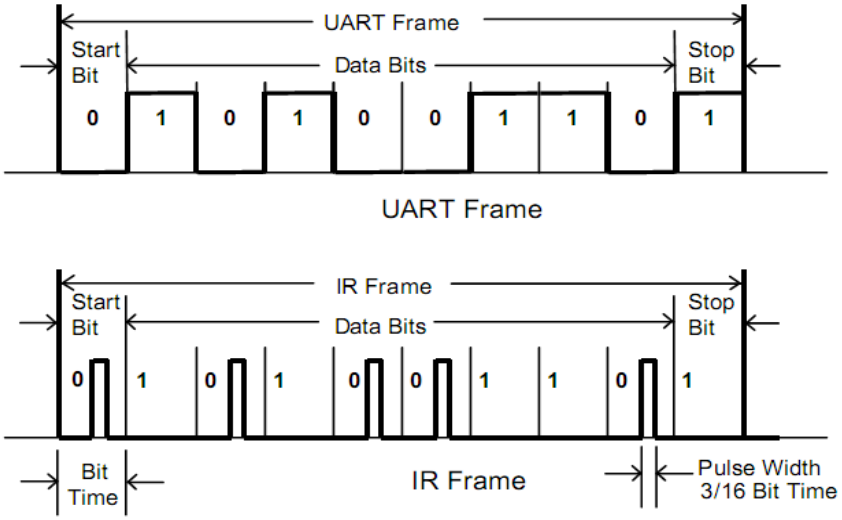
The minimum demand for transmission rates for IrDA is only 9600 bps. All transmissions must be started at this rate to enable compatibility. Higher rates are a matter of negotiation of the ports after establishing the links.

The IrDA protocol adds a modulation scheme to the UART signaling. At the transmitter, bits are modulated. At the receiver, bits are demodulated. The modulation scheme uses a Return-to-Zero-Inverted (RZI) format. A bit value of ‘0’ is signaled by a short ‘1’ pulse on the line and a bit value of ‘1’ is signaled by holding the line to ‘0’. IrDA is using 3/16 RZI modulation.

9 Refer to the *IrPHY (IrDA Physical Layer Link Specification)* (Rev. 1.4 from May 2001) on the IrDA web site at www.irda.org

The Figure 34 shows UART frame and IR frame, comprised a Start Bit, 8 Data Bits, no Parity Bit and ending with a Stop Bit.

Figure 34. UART Frame and IR Frame example



Oversampling Selection

IrDA is using 3/16 RZI modulation, so the sampling clock frequency should be set 16x of selected **Baud rate**, by configuring **Oversampling**. **Oversampling** should always be 16 for IrDA.

Normal versus Low power transmitting

There are two modes of IrDA operation:

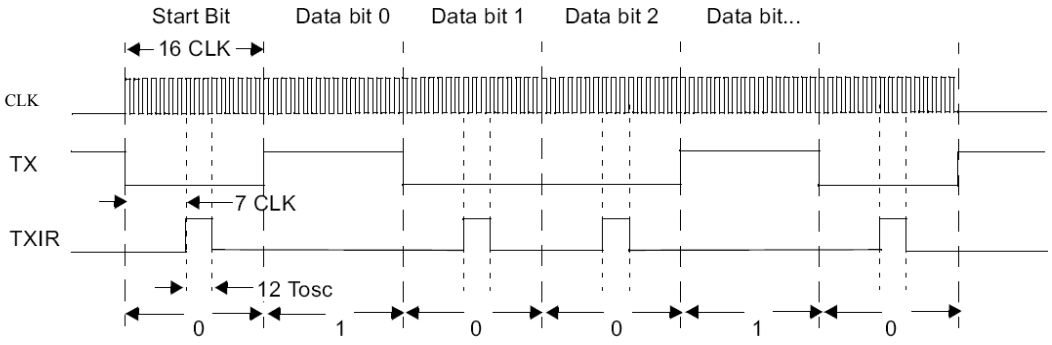
- **Normal transmission** – pulse width is roughly 3/16 of the bit period (for all baud rates)
- **Low power transmission** – pulse width is potentially smaller (down to 1.62 µs typical and 1.41µs minimal) than 3/16 of the bit period (for rates less 115200 bps). Supported only for **RX only** direction.

Inverting RX

This option is used to support two possible demodulation schemes described below.

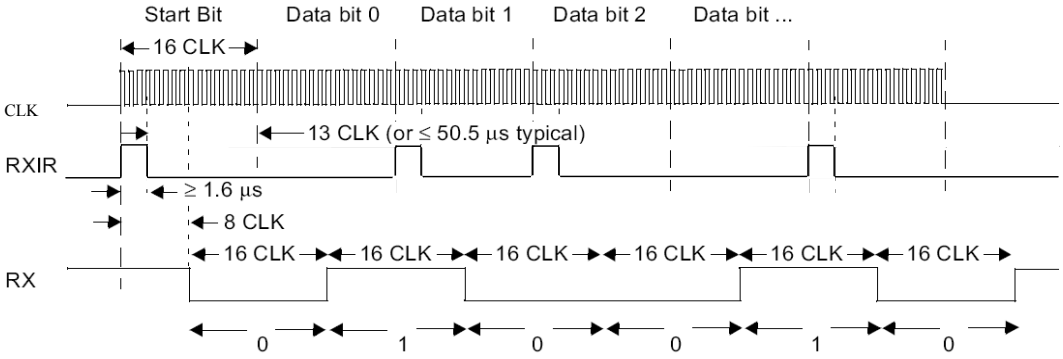
According to the IrPHY specification, the IR frame modulation (encoding) scheme is shown in Figure 35.

Figure 35. IR frame modulation scheme



The IR frame demodulation (decoding) scheme is shown in Figure 36. RXIR line voltage level is default low, active high.

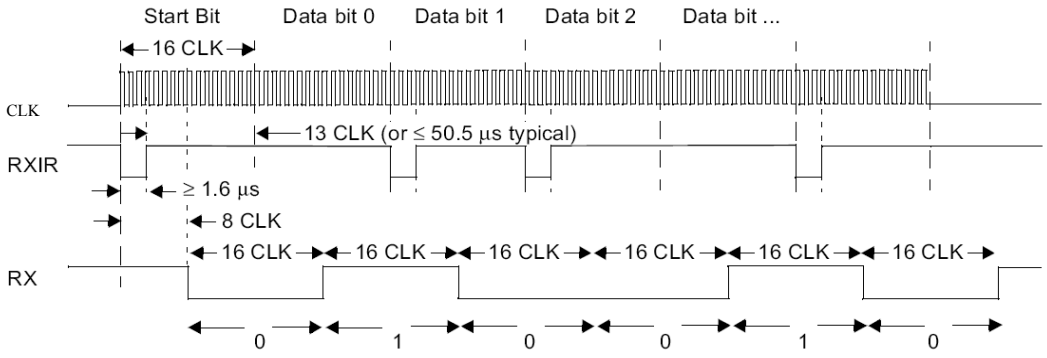
Figure 36. IR frame demodulation scheme 1 (active high)



Note There is a delay from receiving RXIR and decoding RX.

In an application, the RXIR frame output from IrDA transceiver is often pull-up, default high, active low. Figure 37 shows another demodulation scheme.

Figure 37. IR frame demodulation scheme 2 (active low)



FIFO depth

The hardware provides two FIFOs. One is used for receive direction, RX FIFO, and the other for transmit direction, TX FIFO. The FIFO depth is 8 data elements. The width of each data element is 16 bits. The data frame width is configurable from 4-16 bits. One FIFO element is consumed regardless of the data frame width.

PSoC 4100 BLE/PSoC 4200 BLE devices provide the ability to double the FIFO depth to 16 data elements when the data frame width is 4-8 bits.

Software Buffer

Selecting RX or TX Buffer Size values greater than the FIFO depth enables usage of the RX or TX FIFO and a circular software buffer. The array of requested size is allocated internally by the component for the TX software buffer. The allocated array for the RX software buffer has one extra element that remains empty while in operation. Keeping this element empty simplifies circular buffer operation. The interrupt option is automatically updated to Internal, and the RX or TX interrupt source are reserved to provide software buffer operation. The internal interrupt handler is hooked up to the interrupt. Its main purpose is to provide interaction between software buffers and the hardware RX or TX FIFO. The software buffer overflow can happen only for the RX direction. This event is reported via global variable SCB_rxBufferOverflow. For the TX direction, the provided APIs do not allow the software buffer overflow.

Interrupts

When **RX buffer size** or **TX buffer size** is greater than the FIFO depth, the **RX FIFO not empty** or **TX FIFO not full** interrupt sources are reserved by the component for internal software buffer operations. **Do not clear or disable them** because it can cause incorrect software buffer operation. However, it is the user's responsibility to clear interrupts from other enabled interrupt because they are not cleared automatically. Create a custom function that clears these interrupt sources and register it using SCB_SetCustomInterruptHandler(). Each time internal interrupt

handler executes the custom function is called before handling software buffer operation. In case **RX buffer size** or **TX buffer size** is equal to the FIFO depth only the hardware TX or RX FIFO is used. In the **Internal** interrupt mode the interrupts are not cleared automatically. It is the user's responsibility to do this. The **External** interrupt mode is preferred in this case.

Low power modes

The component in UART mode is able to be a wakeup source from Sleep and Deep Sleep low power modes.

Sleep mode is identical to Active from a peripheral point of view. No configuration changes are required in the component or code before entering/exiting this mode. Any UART activity in TX or RX direction that involves an interrupt to occur leads to wakeup.

Deep Sleep mode requires that the UART be properly configured to be a wakeup source. The "Enable wakeup from Deep Sleep Mode" must be checked in the UART configuration dialog and RX direction enabled. The SCB_Sleep() and SCB_Wakeup() functions must be called before/after entering/exiting Deep Sleep.

The device wakes up by the RX GPIO falling edge event that is generated by the incoming start bit. There are two constraints for wakeup:

- The 1st data bit of wakeup transfer has to be '1'. The UART skips the start bit and synchronizes on the 1st data bit.
- The wakeup time of the device must be less than one bit duration; otherwise, the received data will be incorrect.

Note RX GPIO interrupt restricts usage of all GPIO interrupts from the port where the UART rx pin is placed.

Printf() function Usage Model

The printf() function formats a series of strings and numeric values and builds a string to write to an output stream. This function can be used in conjunction with a UART to simplify the formatting and transmission of data. This section describes the code required to allow the use of the printf() function with a UART component.

The printf() function has different implementations in different compilers. Each compiler provides a function that is responsible to send data. These functions are listed below for the supported compilers:

Compiler	Function Name
GCC	_write()
MDK	fputc()
RVDS	fputc()
IAR	__write()

The application should revise these functions to call the communication component API to send data via the selected interface (in this case, the UART interface).

Note The following code example uses an instance name of "SCB" for the SCB UART component:

```
#include <project.h>
#include <stdio.h>

#if defined (__GNUC__)
    /* Add an explicit reference to the floating point printf library to allow
       the usage of floating point conversion specifier. */
    asm (".global _printf_float");

    /* For GCC compiler revise _write() function for printf functionality */
    int _write(int file, char *ptr, int len)
    {
        int i;
        for (i = 0; i < len; i++)
        {
            SCB_UartPutChar(*ptr++);
        }

        return(len);
    }

#elif defined(__ARMCC_VERSION)
    /* For MDK/RVDS compiler revise fputc() function for printf functionality */
    struct __FILE
    {
        int handle;
    };

    enum
    {
        STDIN_HANDLE,
        STDOUT_HANDLE,
        STDERR_HANDLE
    };

    FILE __stdin = {STDIN_HANDLE};
    FILE __stdout = {STDOUT_HANDLE};
    FILE __stderr = {STDERR_HANDLE};

    int fputc(int ch, FILE *file)
    {
        int ret = EOF;

        switch(file->handle)
        {
            case STDOUT_HANDLE:
                SCB_UartPutChar(ch);
                ret = ch;
                break;
```

```
            case STDERR_HANDLE:
                ret = ch;
                break;

            default:
                file = file;
                break;
        }

        return(ret);
    }

#elif defined (__ICCARM__)
    /* For IAR compiler revise __write() function for printf functionality */
    size_t __write(int handle, const unsigned char * buffer, size_t size)
    {
        size_t nChars = 0;

        for (/* Empty */; size != 0; --size)
        {
            SCB_UartPutChar(*buffer++);
            ++nChars;
        }

        return (nChars);
    }

#endif /* (__GNUC__) */

int main()
{
    uint32 i = 444444444;
    float f = 55.555f;

    CyGlobalIntEnable;  /* Enable interrupts */

    SCB_Start();        /* Start communication component */

    /* Use printf() function which will send formatted data through
    * UART (SCB mode) */
    printf("Test printf function. long: %ld, float: %f \n",i,f);

    for(;;)
    {
        /* Place your application code here. */
    }
}
```

The log from terminal software:

```
Test printf function. long: 444444444, float: 55.555000
```

Notes

The printf() function prepares the text stream in the buffer and executes it when it receives new-line character ‘\n’.

Common SCB Component Information

Interrupt APIs

These functions are common for most SCB modes.

By default, PSoC Creator assigns the instance name “SCB_1” to the first instance of a component in a given design. You can rename it to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “SCB.”

Function	Description
SCB_EnableInt()	Enables the interrupt in the NVIC (when an internal interrupt is used).
SCB_DisableInt()	Disables the interrupt in the NVIC (when an internal interrupt is used).
SCB_GetInterruptCause()	Returns a mask of bits showing the source of the current triggered interrupt.
SCB_SetCustomInterruptHandler()	Registers a function to be called by the internal interrupt handler.
SCB_GetTxInterruptSource()	Returns TX interrupt request register. This register contains current status of TX interrupt sources.
SCB_SetTxInterruptMode()	Writes TX interrupt mask register. This register configures which bits from TX interrupt request register will trigger an interrupt event.
SCB_GetTxInterruptMode()	Returns TX interrupt mask register.
SCB_GetTxInterruptSourceMasked()	Returns TX interrupt masked request register. This register contains logical AND of corresponding bits from TX interrupt request and mask registers.
SCB_ClearTxInterruptSource()	Clears TX interrupt sources in the interrupt request register.
SCB_SetTxInterrupt()	Sets TX interrupt sources in the interrupt request register.
SCB_SetTxFifoLevel()	Sets level in the TX FIFO to generate TX level interrupt.
SCB_GetRxInterruptSource()	Returns RX interrupt request register. This register contains current status of RX interrupt sources.
SCB_SetRxInterruptMode()	Writes RX interrupt mask register. This register configures which bits from RX interrupt request register will trigger an interrupt event.
SCB_GetRxInterruptMode()	Returns RX interrupt mask register.
SCB_GetRxInterruptSourceMasked()	Returns RX interrupt masked request register. This register contains logical AND of corresponding bits from RX interrupt request and mask registers.
SCB_ClearRxInterruptSource()	Clears RX interrupt sources in the interrupt request register.
SCB_SetRxInterrupt()	Sets RX interrupt sources in the interrupt request register.

Function	Description
SCB_SetRxFifoLevel()	Sets level in the RX FIFO to generate RX level interrupt.
SCB_GetMasterInterruptSource()	Returns Master interrupt request register. This register contains current status of Master interrupt sources.
SCB_SetMasterInterruptMode()	Writes Master interrupt mask register. This register configures which bits from Master interrupt request register will trigger an interrupt event.
SCB_GetMasterInterruptMode()	Returns Master interrupt mask register.
SCB_GetMasterInterruptSourceMasked()	Returns Master interrupt masked request register. This register contains logical AND of corresponding bits from Master interrupt request and mask registers.
SCB_ClearMasterInterruptSource()	Clears Master interrupt sources in the interrupt request register.
SCB_SetMasterInterrupt()	Sets Master interrupt sources in the interrupt request register.
SCB_ClearSlaveInterruptSource()	Returns Slave interrupt request register. This register contains current status of Slave interrupt sources.
SCB_SetSlaveInterruptMode()	Writes Slave interrupt mask register. This register configures which bits from Slave interrupt request register will trigger an interrupt event.
SCB_GetSlaveInterruptMode()	Returns Slave interrupt mask register.
SCB_GetSlaveInterruptSourceMasked()	Returns Slave interrupt masked request register. This register contains logical AND of corresponding bits from Slave interrupt request and mask registers.
SCB_ClearSlaveInterruptSource()	Clears Slave interrupt sources in the interrupt request register.
SCB_SetSlaveInterrupt()	Sets Slave interrupt sources in the interrupt request register.

void SCB_EnableInt(void)

Description: When using an Internal interrupt, this enables the interrupt in the NVIC. When using an external interrupt the API for the interrupt component must be used to enable the interrupt.

Parameters: None

Return Value: None

Side Effects: None

void SCB_DisableInt(void)

Description: When using an Internal interrupt, this disables the interrupt in the NVIC. When using an external interrupt the API for the interrupt component must be used to disable the interrupt.

Parameters: None

Return Value: None

Side Effects: None

uint32 SCB_GetInterruptCause(void)

Description: Returns a mask of bits showing the source of the current triggered interrupt. This is useful for modes of operation where an interrupt can be generated by conditions in multiple interrupt source registers.

Parameters: None

Return Value: uint32: Mask with the OR of the following conditions that have been triggered:

Interrupt causes constants	Description
SCB_INTR_CAUSE_MASTER	Interrupt from Master
SCB_INTR_CAUSE_SLAVE	Interrupt from Slave
SCB_INTR_CAUSE_TX	Interrupt from TX
SCB_INTR_CAUSE_RX	Interrupt from RX

Side Effects: None

void SCB_SetCustomInterruptHandler(void (*func) (void))

Description: Registers a function to be called by the internal interrupt handler. First the function that is registered is called, and then the internal interrupt handler performs any operations such as software buffer management functions. It is user's responsibility to not break the software buffer operations. Only one custom handler is supported; which is the function provided by the most recent call. At initialization time no custom handler is registered.

Parameters: func: Pointer to the function to register. The value NULL indicates to remove the current custom interrupt handler.

Return Value: None

Side Effects: None

uint32 SCB_GetTxInterruptSource(void)

Description: Returns TX interrupt request register. This register contains current status of TX interrupt sources.

Parameters: None

Return Value: uint32: Current status of TX interrupt sources.

Each constant is a bit field value. The value returned may have multiple bits set to indicate the current status.

TX interrupt sources	Description
SCB_INTR_TX_FIFO_LEVEL	Transmitter FIFO has fewer entries than the value specified by level.
SCB_INTR_TX_NOT_FULL	Transmitter FIFO is not full.
SCB_INTR_TX_EMPTY	Transmitter FIFO is empty.
SCB_INTR_TX_OVERFLOW	Attempt to write to a full transmitter FIFO.
SCB_INTR_TX_UNDERFLOW	Attempt to read from an empty transmitter FIFO.
SCB_INTR_TX_UART_NACK	UART received a NACK in SmartCard mode.
SCB_INTR_TX_UART_DONE	UART transfer is complete. All data elements from the TX FIFO are sent.
SCB_INTR_TX_UART_ARB_LOST	Value on the TX line of the UART does not match the value on the RX line.

Side Effects: None

void SCB_SetTxInterruptMode(uint32 interruptMask)

Description: Writes TX interrupt mask register. This register configures which bits from TX interrupt request register will trigger an interrupt event.

Parameters: uint32 interruptMask: TX interrupt sources to be enabled (refer to SCB_GetTxInterruptSource() function for bit field values).

Return Value: None

Side Effects: None

uint32 SCB_GetTxInterruptMode(void)

Description: Returns TX interrupt mask register This register specifies which bits from TX interrupt request register will trigger an interrupt event.

Parameters: None

Return Value: uint32: Enabled TX interrupt sources (refer to SCB_GetTxInterruptSource() function for return values).

Side Effects: None

uint32 SCB_GetTxInterruptSourceMasked(void)

Description: Returns TX interrupt masked request register. This register contains logical AND of corresponding bits from TX interrupt request and mask registers.

This function is intended to be used in the interrupt service routine to identify which of enabled TX interrupt sources cause interrupt event.

Parameters: None

Return Value: uint32: Current status of enabled TX interrupt sources (refer to SCB_GetTxInterruptSource() function for return values).

Side Effects: None

void SCB_ClearTxInterruptSource(uint32 interruptMask)

Description: Clears TX interrupt sources in the interrupt request register.

Parameters: uint32 interruptMask: TX interrupt sources to be cleared (refer to SCB_GetTxInterruptSource() function for return values).

Return Value: None

Side Effects: The side effects are listed in the table below for each affected interrupt source. Refer to section TX FIFO interrupt sources for detailed description.

TX interrupt sources	Description
SCB_INTR_TX_FIFO_LEVEL	Interrupt source is not cleared when transmitter FIFO has less entries than level.
SCB_INTR_TX_NOT_FULL	Interrupt source is not cleared when transmitter FIFO has empty entries.
SCB_INTR_TX_EMPTY	Interrupt source is not cleared when transmitter FIFO is empty.
SCB_INTR_TX_UNDERFLOW	Interrupt source is not cleared when transmitter FIFO is empty and I2C mode with clock stretching is selected. Put data into the transmitter FIFO before clearing it. This behavior only applicable for PSoC 4100/PSoC 4200 devices.

void SCB_SetTxInterrupt(uint32 interruptMask)

Description: Sets TX interrupt sources in the interrupt request register.

Parameters: uint32 interruptMask: TX interrupt sources to set in the TX interrupt request register (refer to SCB_GetTxInterruptSource() function for return values).

Return Value: None

Side Effects: None

void SCB_ SetTxFifoLevel(uint32 level)

Description: Sets level in the TX FIFO to generate a TX level interrupt.

When the TX FIFO has more entries than the TX FIFO level an TX level interrupt request is generated.

Parameters: uint32 level: Level in the TX FIFO to generate TX level interrupt

The range of valid level values is between 0 and TX FIFO depth - 1.

Return Value: None

Side Effects: None

uint32 SCB_GetRxInterruptSource(void)

Description: Returns RX interrupt request register. This register contains current status of RX interrupt sources.

Parameters: None

Return Value: uint32: Current status of RX interrupt sources.

Each constant is a bit field value. The value returned may have multiple bits set to indicate the current status.

RX interrupt sources	Description
SCB_INTR_RX_FIFO_LEVEL	Receiver FIFO has more entries than the value specified by level.
SCB_INTR_RX_NOT_EMPTY	Receiver FIFO is not empty.
SCB_INTR_RX_FULL	Receiver FIFO is full.
SCB_INTR_RX_OVERFLOW	Attempt to write to a full receiver FIFO.
SCB_INTR_RX_UNDERFLOW	Attempt to read from an empty receiver FIFO.
SCB_INTR_RX_FRAME_ERROR	UART framing error detected.
SCB_INTR_RX_PARITY_ERROR	UART parity error detected.

Side Effects: None

void SCB_SetRxInterruptMode(uint32 interruptMask)

Description: Writes RX interrupt mask register. This register configures which bits from RX interrupt request register will trigger an interrupt event.

Parameters: uint32 interruptMask: RX interrupt sources to be enabled (refer to SCB_GetRxInterruptSource() function for bit fields values).

Return Value: None

Side Effects: None

uint32 SCB_GetRxInterruptMode(void)

Description: Returns RX interrupt mask register This register specifies which bits from RX interrupt request register will trigger an interrupt event.

Parameters: None

Return Value: uint32: Enabled RX interrupt sources (refer to SCB_GetRxInterruptSource() function for return values).

Side Effects: None

uint32 SCB_GetRxInterruptSourceMasked(void)

Description: Returns RX interrupt masked request register. This register contains logical AND of corresponding bits from RX interrupt request and mask registers.

This function is intended to be used in the interrupt service routine to identify which of enabled RX interrupt sources cause interrupt event.

Parameters: None

Return Value: uint32: Current status of enabled RX interrupt sources (refer to SCB_GetRxInterruptSource() function for return values).

Side Effects: None

void SCB_ClearRxInterruptSource(uint32 interruptMask)

Description: Clears RX interrupt sources in the interrupt request register.

Parameters: uint32 interruptMask: RX interrupt sources to be cleared (refer to SCB_GetRxInterruptSource() function for return values).

Return Value: None

Side Effects: The side effects are listed in the table below for each affected interrupt source. Refer to section RX FIFO interrupt sources for detailed description.

RX interrupt sources	Description
SCB_INTR_RX_FIFO_LEVEL	Interrupt source is not cleared when the receiver FIFO has more entries than level.
SCB_INTR_RX_NOT_EMPTY	Interrupt source is not cleared when receiver FIFO is not empty.
SCB_INTR_RX_FULL	Interrupt source is not cleared when receiver FIFO is full.

void SCB_SetRxInterrupt(uint32 interruptMask)

Description: Sets RX interrupt sources in the interrupt request register.

Parameters: uint32 interruptMask: RX interrupt sources to set in the RX interrupt request register (refer to SCB_GetRxInterruptSource() function for return values).

Return Value: None

Side Effects: None

void SCB_SetRxFifoLevel (uint32 level)

Description: Sets level in the RX FIFO to generate a RX level interrupt.

When the RX FIFO has more entries than the RX FIFO level an RX level interrupt request is generated.

Parameters: uint32 level: Level in the RX FIFO to generate RX level interrupt.

The range of valid level values is between 0 and RX FIFO depth - 1.

Return Value: None

Side Effects: None

uint32 SCB_GetMasterInterruptSource(void)

Description: Returns Master interrupt request register. This register contains current status of Master interrupt sources.

Parameters: None

Return Value: uint32: Current status of Master interrupt sources.

Each constant is a bit field value. The value returned may have multiple bits set to indicate the current status.

Master interrupt sources	Description
SCB_INTR_MASTER_SPI_DONE	SPI master transfer is complete. Refer to Interrupt sources section for detailed description.
SCB_INTR_MASTER_I2C_ARB_LOST	I2C master lost arbitration.
SCB_INTR_MASTER_I2C_NACK	I2C master received negative acknowledgement (NAK).
SCB_INTR_MASTER_I2C_ACK	I2C master received acknowledgement.
SCB_INTR_MASTER_I2C_STOP	I2C master generated STOP.
SCB_INTR_MASTER_I2C_BUS_ERROR	I2C master bus error (detection of unexpected START or STOP condition).

Side Effects: None

void SCB_SetMasterInterruptMode(uint32 interruptMask)

Description: Writes Master interrupt mask register. This register configures which bits from Master interrupt request register will trigger an interrupt event.

Parameters: uint32 interruptMask: Master interrupt sources to be enabled (refer to SCB_GetMasterInterruptSource() function for bit field values).

Return Value: None

Side Effects: None

uint32 SCB_GetMasterInterruptMode(void)

Description: Returns Master interrupt mask register This register specifies which bits from Master interrupt request register will trigger an interrupt event.

Parameters: None

Return Value: uint32: Enabled Master interrupt sources (refer to SCB_GetMasterInterruptSource() function for return values).

Side Effects: None

uint32 SCB_GetMasterInterruptSourceMasked(void)

Description: Returns Master interrupt masked request register. This register contains logical AND of corresponding bits from Master interrupt request and mask registers.

This function is intended to be used in the interrupt service routine to identify which of enabled Master interrupt sources cause interrupt event.

Parameters: None

Return Value: uint32: Current status of enabled Master interrupt sources (refer to SCB_GetMasterInterruptSource() function for return values).

Side Effects: None

void SCB_ClearMasterInterruptSource(uint32 interruptMask)

Description: Clears Master interrupt sources in the interrupt request register.

Parameters: uint32 interruptMask: Master interrupt sources to be cleared (refer to SCB_GetMasterInterruptSource() function for return values).

Return Value: None

Side Effects: None

void SCB_SetMasterInterrupt(uint32 interruptMask)

Description: Sets Master interrupt sources in the interrupt request register.

Parameters: uint32 interruptMask: Master interrupt sources to set in the Master interrupt request register (refer to SCB_GetMasterInterruptSource() function for return values).

Return Value: None

Side Effects: None

uint32 SCB_GetSlaveInterruptSource(void)

Description: Returns Slave interrupt request register. This register contains current status of Slave interrupt sources.

Parameters: None

Return Value: uint32: Current status of Slave interrupt sources.

Each constant is a bit field value. The value returned may have multiple bits set to indicate the current status.

Slave interrupt sources	Description
SCB_INTR_SLAVE_I2C_ARB_LOST	I2C slave lost arbitration: the value driven on the SDA line is not the same as the value observed on the SDA line.
SCB_INTR_SLAVE_I2C_NACK	I2C slave received negative acknowledgement (NAK).
SCB_INTR_SLAVE_I2C_ACK	I2C slave received acknowledgement (ACK).
SCB_INTR_SLAVE_I2C_WRITE_STOP	Stop or Repeated Start event for write transfer intended for this slave (address matching is performed).
SCB_INTR_SLAVE_I2C_STOP	Stop or Repeated Start event for (read or write) transfer intended for this slave (address matching is performed).
SCB_INTR_SLAVE_I2C_START	I2C slave received Start condition.
SCB_INTR_SLAVE_I2C_ADDR_MATCH	I2C slave received matching address.
SCB_INTR_SLAVE_I2C_GENERAL	I2C Slave received general call address.
SCB_INTR_SLAVE_I2C_BUS_ERROR	I2C slave bus error (detection of unexpected START or STOP condition).
SCB_INTR_SLAVE_SPI_BUS_ERROR	SPI slave deselected at an expected time in the SPI transfer.

Side Effects: None

void SCB_SetSlaveInterruptMode(uint32 interruptMask)

Description: Writes Slave interrupt mask register. This register configures which bits from Slave interrupt request register will trigger an interrupt event.

Parameters: uint32 interruptMask: Slave interrupt sources to be enabled (refer to SCB_GetSlaveInterruptSource() function for bit field values).

Return Value: None

Side Effects: None

uint32 SCB_GetSlaveInterruptMode(void)

Description: Returns Slave interrupt mask register This register specifies which bits from Slave interrupt request register will trigger an interrupt event.

Parameters: None

Return Value: uint32: Enabled Slave interrupt sources (refer to SCB_GetSlaveInterruptSource() function for return values).

Side Effects: None

uint32 SCB_GetSlaveInterruptSourceMasked(void)

Description: Returns Slave interrupt masked request register. This register contains logical AND of corresponding bits from Slave interrupt request and mask registers.

This function is intended to be used in the interrupt service routine to identify which of enabled Slave interrupt sources cause interrupt event.

Parameters: None

Return Value: uint32: Current status of enabled Slave interrupt sources (refer to SCB_GetSlaveInterruptSource() function for return values).

Side Effects: None

void SCB_ClearSlaveInterruptSource(uint32 interruptMask)

Description: Clears Slave interrupt sources in the interrupt request register.

Parameters: uint32 interruptMask: Slave interrupt sources to be cleared (refer to SCB_GetSlaveInterruptSource() function for return values).

Return Value: None

Side Effects: None

void SCB_SetSlaveInterrupt(uint32 interruptMask)

Description: Sets Slave interrupt sources in the interrupt request register.

Parameters: uint32 interruptMask: Slave interrupt sources to set in the Slave interrupt request register (refer to SCB_GetSlaveInterruptSource() function for return values).

Return Value: None

Side Effects: None

Interrupt Function Appliance

Function	I2C	SPI	UART	EZI2C
SCB_EnableInt()	+	+	+	+
SCB_DisableInt()	+	+	+	+
SCB_GetInterruptCause()	+	+	+	+
SCB_SetCustomInterruptHandler()	+	+	+	+
SCB_GetTxInterruptSource()	+	+	+	+
SCB_SetTxInterruptMode()	+	+	+	+
SCB_GetTxInterruptMode()	+	+	+	+
SCB_GetTxInterruptSourceMasked()	+	+	+	+
SCB_ClearTxInterruptSource()	+	+	+	+
SCB_SetTxInterrupt()	+	+	+	+
SCB_SetTxFifoLevel()	+	+	+	+
SCB_GetRxInterruptSource()	+	+	+	+
SCB_SetRxInterruptMode()	+	+	+	+
SCB_GetRxInterruptMode()	+	+	+	+
SCB_GetRxInterruptSourceMasked()	+	+	+	+
SCB_ClearRxInterruptSource()	+	+	+	+
SCB_SetRxInterrupt()	+	+	+	+
SCB_SetRxFifoLevel()	+	+	+	+
SCB_GetMasterInterruptSource()	+	+	–	–
SCB_SetMasterInterruptMode()	+	+	–	–
SCB_GetMasterInterruptMode()	+	+	–	–
SCB_GetMasterInterruptSourceMasked()	+	+	–	–
SCB_ClearMasterInterruptSource()	+	+	–	–
SCB_SetMasterInterrupt()	+	+	–	–
SCB_GetSlaveInterruptSource()	+	+	–	+
SCB_SetSlaveInterruptMode()	+	+	–	+
SCB_GetSlaveInterruptMode()	+	+	–	+
SCB_GetSlaveInterruptSourceMasked()	+	+	–	+
SCB_ClearSlaveInterruptSource()	+	+	–	+
SCB_SetSlaveInterrupt()	+	+	–	+

Clock Selection

The SCB is clocked by a single dedicated clock connection. Depending on the mode of operation the frequency of this clock may be calculated by the component based on the customizer configuration or may be provided externally.

Since the Unconfigured mode customizer is not aware of the end mode of operation the clock must be provided externally in this case.

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the component. There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator components
- specific deviations – deviations that are applicable only for this component

This section provides information on component-specific deviations. Project deviations are described in the MISRA Compliance section of the *System Reference Guide* along with information on the MISRA compliance verification environment.

The SCB component has the following specific deviations:

MISRA-C: 2004 Rule	Rule Class [10]	Rule Description	Description of Deviation(s)
1.1	R	This rule states that code shall conform to C ISO/IEC 9899:1990 standard.	Nesting of control structures (statements) exceeds 15 - program does not conform strictly to ISO:C90. In practice, most compilers will support a much more liberal nesting limit and therefore this limit may only be relevant when strict conformance is required. By comparison, ISO:C99 specifies a limit of 127 "nesting levels of blocks. The supported compilers (GCC 4.1.1, RVDS and MDK) support larger number nesting of control structures.
17.4	R	Array indexing shall be the only allowed form of pointer arithmetic.	Component uses array indexing operation to access buffers. The buffer size is checked before access. It is safe operation unless user provides incorrect buffer size.
19.7	A	A function should be used in preference to a function-like macro.	Deviated since function-like macros are used to allow more efficient code.

[10] Required / Advisory

This component has the following embedded components: Pins and Interrupt. Refer to the corresponding component datasheets for information on their MISRA compliance and specific deviations.

Sample Firmware Source Code

PSoC Creator provides many example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the “Find Example Project” topic in the PSoC Creator Help for more information.

Interrupt Service Routine

The SCB supports interrupts on the various events, depending on the mode of operation. All of the interrupt events are ORed together before being sent to the interrupt controller, so the SCB can only generate a single interrupt request to the controller at any given time. This signal goes high when any of the enabled interrupt sources are true.

Some of the modes expose this signal as terminal when it is not needed for internal operation as described in the Input/Output Connections section. If it is needed for internal operation the terminal is not present.

Software can service multiple interrupt events in a single interrupt service routine by using various interrupt APIs.

PSoC Creator generates the necessary interrupt service routines for handling internal operation. However it is possible to register a custom function using *SCB_SetCustomInterruptHandler()* function. This user function will be called first, before the internal interrupt handler performs any operations such as software buffer management functions. Only one custom handler is supported.

Note Interrupt sources managed by user are not cleared automatically. It is user responsibility to do that. Interrupt sources are cleared by writing a ‘1’ in corresponding bit position. The preferred way to clear interrupt sources is usage APIs (for example: `SCB_ClearRxInterruptSource()`).

```
void CustomInterruptHandler(void);
void main()
{
    /* Register custom function */
    SCB_SetCustomInterruptHandler(&CustomInterruptHandler);

    /* Initialize SCB component in UART mode.
    * The SCB_INTR_RX_PARITY_ERROR is already enabled in GUI:
    * UART Advanced Tab.
    */
    SCB_Start();
    CyGlobalIntEnable; /* Enable global interrupts. */
    for(;;)
```

```
        {
            /* Place your application code here. */
        }
    }

    /* User interrupt handler to insert into SCB interrupt handler.
    * Note: SCB interrupt set to Internal in GUI.
    */
    void CustomInterruptHandler(void)
    {
        if(0u != (SCB_GetRxInterruptSourceMasked() & SCB_INTR_RX_PARITY_ERROR))
        {
            /* Interrupt sources does not clear automatically if it is managed by
            * user. The interrupt sources clearing becomes user responsibility.
            */
            SCB_ClearRxInterruptSource(SCB_INTR_RX_PARITY_ERROR);

            /*
            * Add user interrupt code to manage SCB_INTR_RX_PARITY_ERROR.
            */
        }
    }
```

TX FIFO interrupt sources

The following TX interrupt sources have level-sensitive behavior:

- TX FIFO empty
- TX FIFO not full
- TX FIFO level

These interrupt sources trigger the current status of the TX FIFO, and keep it until a clear operation. Clearing these interrupt sources does not make any sense if the current status of the TX FIFO still triggers them, because they are restored back.

The restore operation takes one clock cycle; therefore, the interrupt source is cleared during this time. The restore time causes false clearing of interrupt sources, which might have an undesired impact on the design.

The suggested flow to start TX FIFO interrupt sources processing is as follows:

1. Fill the TX FIFO with data. While filling the TX FIFO, it is better to monitor the number of entries in it rather than try to clear the TX FIFO interrupt source after each byte put in the TX FIFO.
2. Clear the “old” triggered interrupt source. To clear interrupt source, write ‘1’ to the corresponding bit position.
3. Enable the interrupt source.

Note The TX FIFO level interrupt source behavior depends on the level value.

Note The described behavior applies only to PSoC 4100/PSoC 4200 devices. After the SCB component is disabled, the TX FIFO becomes empty. It is not possible to clear them due to the restore nature. Therefore, the component interrupt or TX interrupt sources must be disabled to not cause locking in the interrupt handler after the component was disabled. For other devices, this behavior is fixed; after the SCB component is disabled all interrupts are cleared.

RX FIFO interrupt sources

The following RX interrupt sources have level-sensitive behavior:

- RX FIFO not empty
- RX FIFO full
- RX FIFO level

These interrupt sources trigger the current status of the RX FIFO, and keep it until a clear operation. Clearing these interrupt sources does not make any sense if the current status of the RX FIFO still triggers them, because they are restored back. The restore operation takes one clock cycle; therefore, the interrupt source is cleared during this time.

The restore time causes false clearing of RX interrupt sources, which might have an undesired impact on the design.

The suggested flow to start RX FIFO interrupt sources processing is as follows:

1. Clear the “old” triggered interrupt source. To clear interrupt source, write ‘1’ to the corresponding bit position.
2. Then enable the interrupt source.

In most cases the clear operation is not required. While getting data from the RX FIFO, it is better to monitor the number of entries in it rather than try to clear the RX FIFO interrupt source after each byte read from the RX FIFO.

Note The RX FIFO level interrupt source behavior depends on level value.

Note The described behavior applies only to PSoC 4100/PSoC 4200 devices. After the SCB component is disabled, the triggered RX FIFO interrupt sources are not cleared automatically. The explicit clear operation must be executed.

Placement

The SCB is placed as Fixed Function block and all placement information is provided to the API through the *cyfitter.h* file.

The SCB pins placement information is available in the **Pins** tab of the PSoC Creator Design-Wide Resources (DWR) file. See the *Device datasheet* section *Pinout* for pins functions and placement information. The pins which provide SCB interfaces functionality might be combined

with debug functionality. To utilize these pins for SCB functionality debug capability needs to be disabled in the **System** tab of the PSoC Creator Design-Wide Resources (DWR) file.

Registers

See the chip *Technical Reference Manual (TRM)* for more information about registers.

Component Debug Window

PSoC Creator allows you to view debug information about components in your design. Each component window lists the memory and registers for the instance. For detailed hardware registers descriptions, refer to the appropriate device technical reference manual.

To open the Component Debug window:

1. Make sure the debugger is running or in break mode.
2. Choose **Windows > Components…** from the **Debug** menu.
3. In the Component Window Selector dialog, select the component instances to view and click **OK**.

The selected Component Debug window(s) will open within the debugger framework. Refer to the "Component Debug Window" topic in the PSoC Creator Help for more information.

Resources

SCB is implemented as a fixed-function block.

Mode		Resource Type		
		SCB Fixed Blocks		Interrupts
Unconfigured SCB		1		1
I^2C	Slave	1		1
	Master	1		1
	Multi-Master	1		1
	Multi-Master-Slave	1		1
SPI	Slave	Hardware buffers	1	–
		Software buffers	1	1
	Master	Hardware buffers	1	–
		Software buffers	1	1
UART	Standard	Hardware buffers	1	–
		Software buffers	1	1
	Standard (Multiprocessor mode)	Hardware buffers	1	–
		Software buffers	1	1

Mode		Resource Type		
		SCB Fixed Blocks		Interrupts
	SmartCard	Hardware buffers	1	–
		Software buffers	1	1
	IrDA	Hardware buffers	1	–
		Software buffers	1	1
EZ I^2C	Clock stretching enabled	One address	1	1
		Two addresses	1	1
	Clock stretching disabled	One address	1	1

API Memory Usage

The component memory usage varies significantly, depending on the compiler, device, number of APIs used and component configuration. The following table provides the memory usage for all APIs available in the given component configuration.

The measurements have been done with the associated compiler configured in Release mode with optimization set for Size. For a specific design the map file generated by the compiler can be analyzed to determine the memory usage.

Configuration			PSoC 4 (GCC)					
			PSoC 4000		PSoC 4100 / PSoC 4200		PSoC 4100 BLE / PSoC 4200 BLE	
			Flash	RAM	Flash	RAM	Flash	RAM
Unconfigured SCB			7316	154	10602	171	11330	171
I^2C	Slave		1604	42	1680	43	1552	43
	Master		2564	46	2804	46	2572	46
	Multi-Master		2564	46	2804	46	2572	46
	Multi-Master-Slave		3860	79	4100	79	3808	79
SPI	Slave	Hardware buffers [11]	N/A	N/A	562	9	602	9
		Software buffers [12]	N/A	N/A	966	50	1006	50
	Master	Hardware buffers [11]	N/A	N/A	606	9	646	9

[11] Hardware buffers – Only hardware TX and RX FIFO are used. The TX and RX FIFO depth is 8 data elements when Byte mode option is disabled and 16 data elements otherwise. Interrupt mode is None.

[12] Software buffers – RX and TX buffers size equal to 10 data elements, the internal RAM buffers are used as well as hardware FIFO. Internal interrupt is automatically enabled in this case.

Configuration			PSoC 4 (GCC)					
			PSoC 4000		PSoC 4100 / PSoC 4200		PSoC 4100 BLE / PSoC 4200 BLE	
			Flash	RAM	Flash	RAM	Flash	RAM
		Software buffers [12]	N/A	N/A	1006	50	1046	50
UART	Standard [13]	Hardware buffers [11]	N/A	N/A	708	9	848	9
		Software buffers [12]	N/A	N/A	1152	50	1288	50
	Standard (Multiprocessor mode)	Hardware buffers [11]	N/A	N/A	764	9	776	9
		Software buffers [12]	N/A	N/A	1224	70	1232	70
	SmartCard	Hardware buffers [11]	N/A	N/A	712	9	720	9
		Software buffers [12]	N/A	N/A	1156	50	1164	50
	IrDA	Hardware buffers [11]	N/A	N/A	716	9	724	9
		Software buffers [12]	N/A	N/A	1160	50	1168	50
EZ I^2C	Clock stretching enabled	One address	1316	26	1360	26	1264	26
		Two addresses	1688	50	1768	50	1632	50
	Clock stretching disabled [14]	One address	1272	23	1056	24	1228	23

DC and AC Electrical Characteristics

Specifications are valid for –40 °C ≤ T_A ≤ 85 °C and T_J ≤ 100 °C, except where noted.
Specifications are valid for 1.71 V to 5.5 V, except where noted.

PSoC 4000

I^2C DC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
I_{I2C1}	Block current consumption at 100 KHz	–	–	10.5	µA	
I_{I2C2}	Block current consumption at 400 KHz	–	–	135	µA	
I_{I2C4}	I^2C enabled in Deep Sleep mode	–	–	2.5	µA	

[13] The hardware flow control feature is enabled for PSoC 4100 BLE/PSoC 4200 BLE devices.

[14] The “Enable wakeup from Deep Sleep Mode” is enabled for all devices other than PSoC 4100/PSoC 4200 devices.

I²C AC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
F_{I2C1}	Bit rate	–	–	400	Kbps	

PSoC 4100/PSoC 4200

I²C DC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
I_{I2C1}	Block current consumption at 100 KHz	–	–	10.5	µA	
I_{I2C2}	Block current consumption at 400 KHz	–	–	135	µA	
I_{I2C3}	Block current consumption at 1 Mbps	–	–	310	µA	
I_{I2C4}	I^2C enabled in Deep Sleep mode	–	–	1.4	µA	

I²C AC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
F_{I2C1}	Bit rate	–	–	1	Mbps	

UART DC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
I_{UART1}	Block current consumption at 100 Kbits/sec	–	–	9	µA	
I_{UART2}	Block current consumption at 1000 Kbits/sec	–	–	312	µA	

UART AC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
F_{UART}	Bit rate	–	–	1	Mbps	

SPI DC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
I_{SPI1}	Block current consumption at 1 Mbits/sec	–	–	360	µA	
I_{SPI2}	Block current consumption at 4 Mbits/sec	–	–	560	µA	

Parameter	Description	Min	Typ	Max	Units	Conditions
I_{SPI3}	Block current consumption at 8 Mbits/sec	–	–	600	µA	

SPI AC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
F_{SPI}	SPI operating frequency (master; 6X oversampling)	–	–	8	MHz	

SPI Master AC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
T_{DMO}	MOSI valid after Sclock driving edge	–	–	15	ns	
T_{DSI}	MISO valid before Sclock capturing edge. Full clock, late MISO Sampling used	20	–	–	ns	
T_{HMO}	Previous MOSI data hold time with respect to capturing edge at Slave	0	–	–	ns	

SPI Slave AC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
T_{DMI}	MOSI valid before Sclock capturing edge	40	–	–	ns	
T_{DSO}	MISO valid after Sclock driving edge	–	–	42 + 3 × Tscb	ns	
T_{HSO}	Previous MISO data hold time	0	–	–	ns	
$T_{SSELSCK}$	SSEL Valid to first SCK Valid edge	100	–	–	ns	

PSoC 4100 BLE/PSoC 4200 BLE

I^2C DC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
I_{I2C1}	Block current consumption at 100 KHz	–	–	50	µA	
I_{I2C2}	Block current consumption at 400 KHz	–	–	155	µA	
I_{I2C3}	Block current consumption at 1 Mbps	–	–	390	µA	
I_{I2C4}	I^2C enabled in Deep Sleep mode	–	–	1.4	µA	

I²C AC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
F_{I2C1}	Bit rate	–	–	1	Mbps	

UART DC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
I_{UART1}	Block current consumption at 100 Kbits/sec	–	–	55	µA	
I_{UART2}	Block current consumption at 1000 Kbits/sec	–	–	312	µA	

UART AC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
F_{UART}	Bit rate	–	–	1	Mbps	

SPI DC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
I_{SPI1}	Block current consumption at 1 Mbits/sec	–	–	360	µA	
I_{SPI2}	Block current consumption at 4 Mbits/sec	–	–	560	µA	
I_{SPI3}	Block current consumption at 8 Mbits/sec	–	–	600	µA	

SPI AC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
F_{SPI}	SPI operating frequency (master; 6X oversampling)	–	–	8	MHz	

SPI Master AC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
T_{DMO}	MOSI valid after Sclock driving edge	–	–	18	ns	
T_{DSI}	MISO valid before Sclock capturing edge. Full clock, late MISO Sampling used	20	–	–	ns	

Parameter	Description	Min	Typ	Max	Units	Conditions
T_{HMO}	Previous MOSI data hold time with respect to capturing edge at Slave	0	–	–	ns	

SPI Slave AC Specifications

Parameter	Description	Min	Typ	Max	Units	Conditions
T_{DMI}	MOSI valid before Sclock capturing edge	40	–	–	ns	
T_{DSO}	MISO valid after Sclock driving edge	–	–	42 + 3 × Tscb	ns	
T_{HSO}	Previous MISO data hold time	0	–	–	ns	
$T_{SSELSCK}$	SSEL Valid to first SCK Valid edge	100	–	–	ns	

Component Changes

This section lists the major changes in the component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
2.0.a	Datasheet updates.	Fixed a few summary tables to add/remove tables, plus a few minor edits. Updated descriptions for a few APIs. Updated the characterization data.
2.0	Added support for Bluetooth Low Energy devices.	
	Improved internal clock selection logic for all modes. Restricted the I2C master and slave clock frequency requirements to meet I2C specification parameters.	Version 2.0 is not completely backward compatible with the previous component version 1.20. Internal clock selection has changed for I2C, EZI2C, SPI and UART modes. To make version 2.0 backward compatible the “Clock from terminal” option with previous clock settings should be used. To do this you need to discover the clock frequency for the SCB that was used in version 1.20. This can be found by looking at the Clock tab of the .cydwr file. Find a clock that ends in _SCBCLK. In version 2.0 enable the “Clock from terminal” option in the component. Next, connect a clock component to the clock terminal on the SCB component. Set this clock frequency to the same frequency as the clock used in version 1.20. The I2C Slave restricts the input clock frequency to be no less than 1.58 MHz. The clock frequency must be increased to reach minimum requirement to build the project.
	Removed Median filter option from the I2C and EZ I2C modes. The median filter option is set depends on the selected data rate.	The analog filters applied to I2C lines for most data rate modes. There is no reason to apply both filters: analog and median (digital). For I2C master modes with data rate greater than 400 kbps (Fast Plus) only median (digital) filter is applied. If Unconfigured SCB was utilized with previous version of the component and configured to I2C mode the dataRate field of configuration structure has to be initialized for correct filter selection.
	The SCB_Sleep() and SCB_Wakeup() functions were modified for I2C and EZI2C slave (clock stretching enabled) modes for PSoC 4000. For more information refer to Low power modes section of the appropriate mode.	This change intended to address the SCL lock up after wakeup from Deep Sleep on address match event.
	Fixed the I2C master operation in the Multi-Master-Slave mode when Enable wakeup from Deep Sleep Mode option is enabled for PSoC 4000.	The master was not able to able to start communication.

Version	Description of Changes	Reason for Changes / Impact

	Added APIs to set level in the RX and TX FIFO to generate appropriate level interrupt:
	`void SCB_SetRxFifoLevel(uint32 level)`
	`void SCB_SetTxFifoLevel(uint32 level)`	

	Removed SPI Master slave select routing constrains.	SPI Master slave select output pin can be placed to the any allowed slave select location, ss0-ss3.
	The Number of SS lines is allowed to be 0 for SPI Master.	This allows removal of all hardware slave select lines in Master mode. It is useful when slave select control required to be implemented in firmware.
	Changed SPI Master minimum Oversampling value to 6 and removed dependencies from other parameters.	Fixed incorrect oversampling limitations.
	Added API function to access SPI bus state: `SCB_SpiIsBusBusy()`	This function facilitates detection that SPI transfer is completed.

	Add protection from the component interruption to the following APIs:
	`SCB_I2CSlaveInitReadBuf()`
	`SCB_I2CSlaveInitWriteBuf()`
	`SCB_I2CSlaveClearReadStatus()`
	`SCB_I2CSlaveClearWriteStatus()`
	`SCB_I2CMasterClearStatus()`
	`SCB_I2CMasterStatus()`
	`SCB_I2CMasterClearWriteBuf()`
	`SCB_I2CMasterClearReadBuf()`
	`SCB_EzI2CSetBuffer1()`
	`SCB_EzI2CSetBuffer2()`
		I2C and EZI2C operations executed by the listed functions are atomic.

1.20	Fixed EZ I2C with clock stretching operation when SCB Unconfigured mode is selected.	The complier gives warning while compilation.
	Fixed EZ I2C mode with clock stretching buffer update when master writes number of bytes multiplied of 8 and starts from base address multiplied of 8.	The EZ I2C slave completes transfer too early and last 8 bytes remains in the RX FIFO. The buffer is not updated properly.
	Fixed EZ I2C current address behavior for buffer size equal to 256 bytes.	When buffer read or write overflow is occurred the current address wraps around to first element instead point to outside the buffer.
	Improved interrupt handling timings for EZ I2C mode without clock stretching.	
	Added support of PSoC 4000 devices.	
1.10	EZ I2C mode added.	

Version	Description of Changes	Reason for Changes / Impact
	SPI/UART internal interrupt source to transfer data from the internal software buffer into the TX FIFO is changed from TX_EMPTY to TX_NOT_FULL.	This change will result in the TX FIFO being kept full when data is available in the software buffer. This will reduce the likelihood that the FIFO will become empty during a transmission due to a long interrupt response time.
1.0.a	Edits to the component datasheet to match the GUI.	
1.0	The first release of the SCB component	

© Cypress Semiconductor Corporation, 2014. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control, or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC® is a registered trademark, and PSoC Creator™ and Programmable System-on-Chip™ are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.