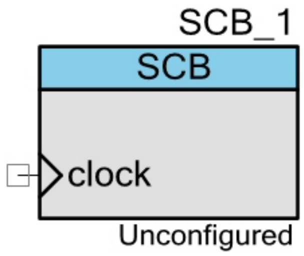
PSoC 4 Serial Communication Block (SCB)

1.20



Features

- Pre-configured components:
  - Industry-standard NXP® I^{2}C bus interface
  - Standard SPI Master and Slave functionalities with Motorola, Texas Instruments, and the National Semiconductor's Microwire protocol
  - Standard UART TX and RX functionalities with SmartCard reader and IrDA protocols
  - EZ I^{2}C mode which emulates a common I^{2}C EEPROM interface
- Supports wakeup from Deep Sleep mode
- Run-time customization
- I^{2}C Bootloader support

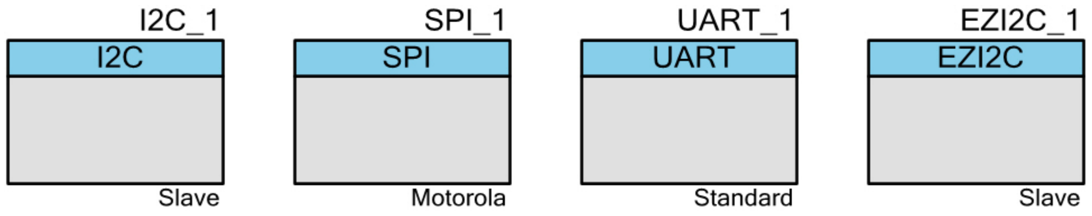
General Description

The PSoC 4 SCB component is a multifunction hardware block that implements the following components. 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 I^{2}C modes. The UART or SPI mode choice is not available.

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

I2C **SPI** **UART** **EZI2C**

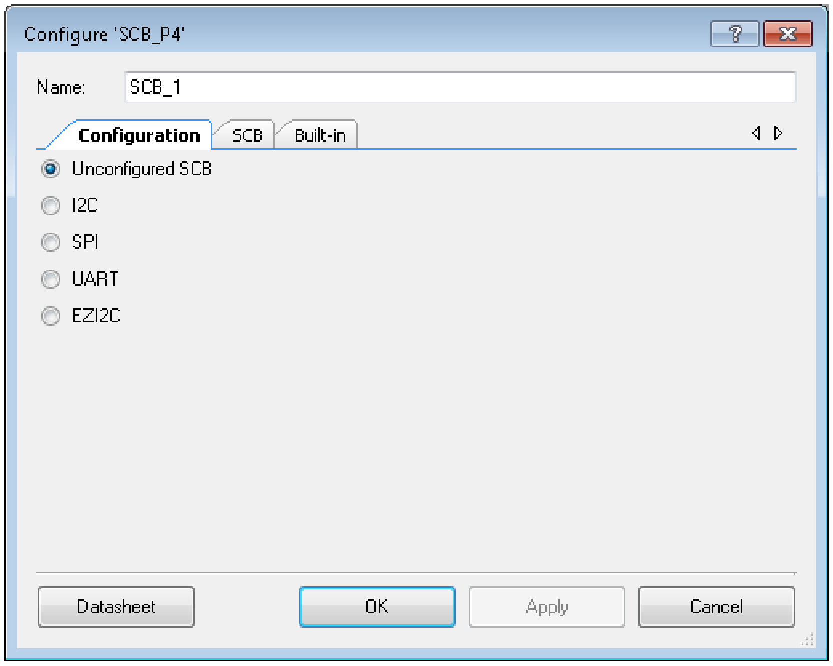


There is also an Unconfigured SCB component entry in the component catalog.

When to Use an SCB Component

The SCB can be used in a pre-configured mode: I2C, SPI, UART and EZI2C. Alternatively the SCB can be Unconfigured at build time and configured at run-time into any of the modes with any setting value using APIs. All configuration settings can be made at run time.

The following shows the base component Configure dialog to select the appropriate mode.



Name: SCB_1

Configuration | SCB | Built-in

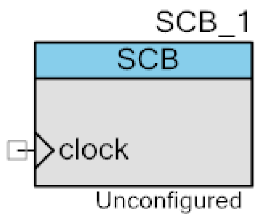
[✓ Unconfigured SCB]
I2C
SPI
UART
EZI2C

Note PSoC 4000 devices support only I^{2}C 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 that can be used 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 from the unconfigured mode.



Input/Output Connections

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

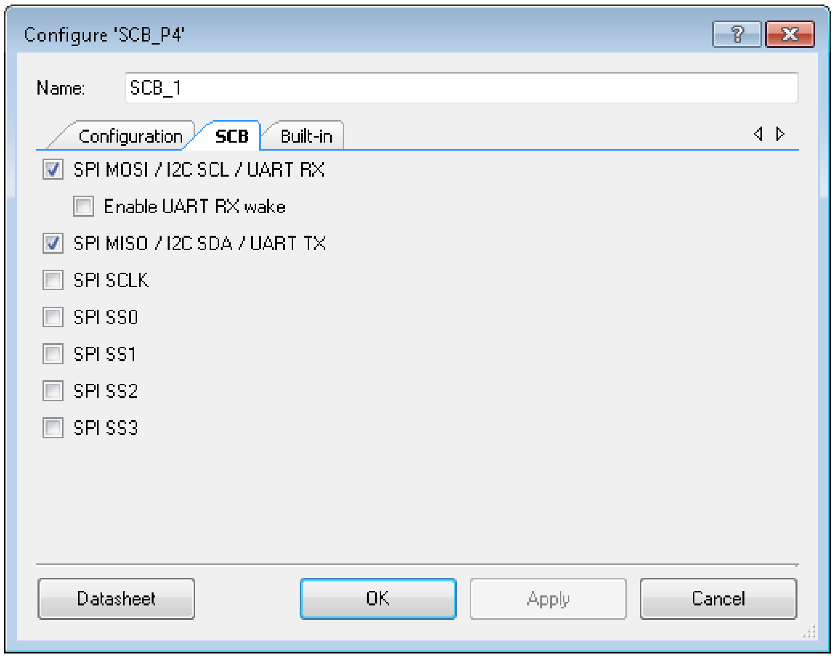
clock – Input

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

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 chip *Technical Reference Manual (TRM)* for more information.

Note The input buffer of buried output pins is disabled to not 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


Configure 'SCB_P4'

Name: SCB_1

- [✓] SPI MOSI / I2C SCL / UART RX
  - [ ] Enable UART RX wake
- [✓] SPI MISO / I2C SDA / UART TX
- [ ] SPI SCLK
- [ ] SPI SS0
- [ ] SPI SS1
- [ ] SPI SS2
- [ ] SPI SS3

Note PSoC 4000 devices support only I^{2}C modes. The following pin names are used for these devices:

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

The **SCB** tab options allow pin selection that will be utilized by the configured interface. The interface type along with the pin name is listed with a checkbox to enable that terminal on the symbol.

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.

Unconfigured mode operation

Before starting operation in Unconfigured mode, chose from the list of supported interfaces. Use the **SCB** parameters tab to select pins required ro implement the chosen interfaces.

The 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^{2}C interface; and as RX for the UART interface.

After selecting pins to utilize the range of selected interfaces, the clock component must be connected to the SCB clock input. This clock frequency along with the SCB oversampling configuration determines the speed of operation of the interface. The clock frequency will be configured later in the firmware setting clock divider. The possible choice of clock configuration is: source HFCLK with divider 1.

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

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

Interface data rate configuration

The data rate for each interface is calculated using a formula (where f_{SCBCLK} is the frequency of the clock component connected to the SCB):

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

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

f_{SCBCLK} = Data rate * Oversampling value

The oversampling values range is defined for every interface. Refer to the I2C / SPI / UART / EZI2C 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^{2}C 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 oversampling default value 16 is chosen to calculate Div_{SCBCLK} for the I^{2}C slave.

f_{SCBCLK} = Data rate * Oversampling value = 100 000 * 16 = 1.6MHz

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

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

f_{SCBCLK} = Data rate * Oversampling value = 115200 * 16 = ~1,843 MHz

$Div_{SCBCLK} = f_{HFCLK} / f_{SCBCLK}$ = 24 MHz / 1, 843 MHz = ~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}.

Actual $f_{SCBCLK} = f_{HFCLK} / Div_{SCBCLK}$ = 24 MHz / 13 = ~1,846 Hz.

The deviation of actual f_{SCBCLK} from desired must be calculated: (1,843MHz – 1,846 MHz) / 1,843 MHz = ~0.2%

Taking to 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:

- I2C slave data rate = 100 kbps: Oversampling = 16, Div_{SCBCLK} = 15;
- UART baud rate = 115200 bps: Oversampling = 16, Div_{SCBCLK} = 13;

Run-time Configuration

The SCB component Configure dialog is used to configure the component. Select an interface and set its configuration. To provide the same functionality in run time, the configuration structures are provided for every interface. These structures provide a number of the fields that match the selections available in the Configure dialog.

Allocate structures for the selected interface and fill it with configuration. A pointer to this structure is passed to the appropriate initialization function of the selected interface. The function contains a prefix of the interface to which it belongs to. For example function to configure in I^2C mode:

```
void SCB_I2CInit(SCB_I2C_INIT_STRUCT *config)
```

The following example provides configuration structures for:

- I2C slave, data rate 1.6 kbps, slave address is 0x08
- UART RX+TX, sub-mode Standard, buffer size 10 for TX and RX (implies software buffer utilization)

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

```
#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 */

/* Use defines from I2C customizer setup */
```

```
const SCB_I2C_INIT_STRUCT configI2C =
{
    SCB_I2C_MODE_SLAVE, /* mode: slave */
    0u,    /* oversampleLow: N/A for slave, SCBCLK provides oversampling */
    0u,    /* oversampleHigh: N/A for slave, SCBCLK provides oversampling */
    1u,    /* enableMedianFilter: enable */
    0x08u, /* slaveAddr: slave address */
    0xFEu, /* slaveAddrMask: signle slave address */
    0u,    /* acceptAddr: disable */
    0u     /* enableWake: disable */
};

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 */
};
```

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^{2}C and UART above.

```
/* The clock divider value which is written into the register has to be less for
one from calculated */
#define SCBCLK_I2C_DIVIDER  (14u)   /* I2C Slave: 100 kbps with OVS = 16. 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.

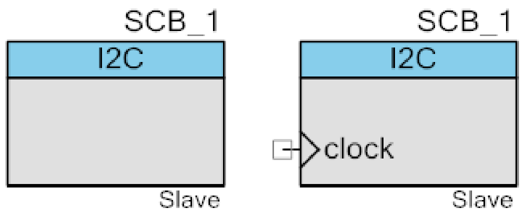
API Names

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 belongs to 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() belongs to the SPI or UART interface.

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

I2C



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

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

The component supports standard clock speeds up to 1000 kbps. It is compatible with I^{2}C Standard-mode, Fast-mode, and Fast-mode Plus devices as defined in the NXP I^{2}C-bus specification [1] 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 I/Os indicates that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.

clock – Input*

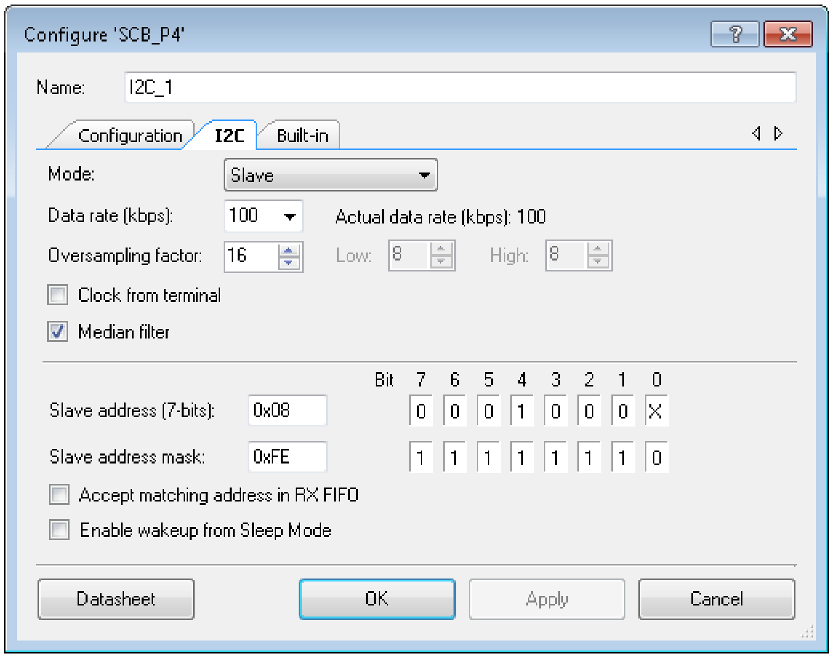
Clock that operates this block. The presence of this terminal varies depending on the **Clock from terminal** parameter.

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 chip *Technical Reference Manual (TRM)* for more information.

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

I2C Parameters

Drag an SCB component onto your design and double click it to open the **Configure** dialog.



Configure 'SCB_P4'

Name: I2C_1

Mode: Slave

Data rate (kbps): 100 Actual data rate (kbps): 100

Oversampling factor: 16 Low: 8 High: 8

☐ Clock from terminal

☑ Median filter

	Bit	7	6	5	4	3	2	1	0
Slave address (7-bits):	0x08	0	0	0	1	0	0	0	X
Slave address mask:	0xFE	1	1	1	1	1	1	1	0

☐ Accept matching address in RX FIFO

☐ Enable wakeup from Sleep Mode

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

Mode

This option determines what modes are 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 speed may differ based on available clock speed and divider range. The standard data rates are 50, 100 (default), 400, and 1000 kbps. If **Clock from terminal** is set, the **Data Rate** parameter is ignored; the input clock and **Oversampling factor** determines the actual data rate.

Actual data rate

Actual data rate displays data rate on which component will operate with current settings. The selected data rate could be differing from actual data rate. The factors that have effect on actual data rate calculation are: oversampling factor, HFCLK clock and accuracy of internal or external component clock.

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. **Oversampling factor** is used to calculate the internal component clock frequency required to achieve this amount of oversampling for the defined **Data rate**.

- For **Master** modes, the **Oversampling factor** is the sum of Low and High oversampling values. These values are written into the register and used for Low and High phases of I^2C clock generation.
- For **Slave** modes, only the component clock source is important. This has to provide the required amount of oversampling.

An oversampling factor maximum value is 32 and the minimum value depends on **Median filter** settings. Median filter is unchecked – 12, Median filter is checked – 14. The default is 16.

Low

This parameter defines the oversampling factor low of the I^2C SCL clock phase; the number of component clocks within one low period I^2C SCL clock. It is only applicable for **Master** modes. The minimum oversampling factor value depends on **Median filter** settings. Median filter is unchecked – 7, Median filter is checked – 8. The default is 8.

High

This parameter defines the oversampling factor high of the I^2C SCL clock phase; the number of component clocks within one high period I^2C SCL clock. It is only applicable for **Master** modes. The minimum oversampling factor value depends on **Median filter** settings. Median filter is unchecked – 5, Median filter is checked – 6. The default is 8.

Clock from terminal

This parameter allows choosing between an internally configured clock and an externally configured clock for data rate generation. When option is enabled, the component does not control the data rate, but displays the actual data rate based on the user-connected clock source and component oversampling factor. When this option is not enabled, PSoC Creator configures the required clock source. The clock source frequency is calculated by component based on the **Data rate** parameter and oversampling factor.

Note When setting the data rate or external clock frequency value, make sure that PSoC Creator can provide this value using the current system clock frequency. Otherwise, a warning about the clock accuracy range is generated while building the project. This warning contains the actual clock value set by PSoC Creator. Choose whether the system clock or component clock should be changed to fit the clocking system requirements and achieve an optimal value.

Median filter

This parameter applies digital 3 taps median filter on input path of I^2C SDA. This filter reduces the susceptibility to errors. However, minimum oversampling factor value is increased.

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 0 and 127 may be selected; the default is **8**.

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

Slave address mask

This parameter is used to mask bit of slave address while address match procedure. The bit 0 of address mask corresponds to read/write direction bit and always does not care in address match.

- 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.

For example: Slave address is 0x36 and Slave Address Mask is 0xDE (bit 0 is R/W bit and bit5 set as does not care in address match). The matched slave addresses are: 0x36 and 0x26.

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

Accept matching address in RX FIFO

This parameter determines whether to accept a match I^2C slave address in the RX FIFO or not. Enable this option if more than one I^2C address support desired. Implement the software address matching code which decides whether address from RX FIFO matches supported addresses range.

Enable wakeup from Sleep Mode

This option allows the system to be awakened from sleep when a slave address match occurs. This option is only available for Slave or Muti-Master-Slave mode.

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

- slave address must be even (bit 0 equal zero)
- median filter must be disabled

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

I2C APIs

Application Programming Interface (API) routines 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_Init()	Initialize the SCB component according to defined parameters in the customizer.
SCB_Enable()	Enables the SCB component operation.
SCB_Start()	Starts the SCB component.
SCB_Stop()	Disable the SCB component.
SCB_Sleep()	Prepares the SCB component to enter Deep Sleep.
SCB_Wakeup()	Prepares the SCB component to exit Deep Sleep.
SCB_I2CInit()	Configures the SCB component for operation in I2C mode.
SCB_I2CSlaveStatus()	Returns slave status flags.
SCB_I2CSlaveClearReadStatus()	Returns read status flags and clears slave read status flags.
SCB_I2CSlaveClearWriteStatus()	Returns the 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).

Function	Description
SCB_I2CSlaveGetReadBufSize()	Returns the number of bytes read by the master since SCB_I2CSlaveClearReadBuf() was called.
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 an 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.

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()	–	+	+	+
SCB_I2CMasterGetReadBufSize()	–	+	+	+
SCB_I2CMasterGetWriteBufSize()	–	+	+	+
SCB_I2CMasterClearReadBuf()	–	+	+	+
SCB_I2CMasterClearWriteBuf()	–	+	+	+

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. 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 the component.
Parameters:	None
Return Value:	None
Side Effects:	None

void SCB_Start(void)

Description:	Invokes SCB_Init() and SCB_Enable(). After this function call the component is enabled and ready for 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_Stop(void)

Description:	Disables the SCB component and its interrupt.
Parameters:	None
Return Value:	None
Side Effects:	None

void SCB_Sleep(void)

Description:	Prepares component to enter Deep Sleep. The “Enable wakeup from Sleep Mode” selection has an influence on this function implementation. 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 exit Deep Sleep. The “Enable wakeup from Sleep Mode” selection has influence on this function implementation. **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 I^2C 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, 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	0 – disable 1 – enable
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.

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^{2}C 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[2]. 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.
SCB_I2C_SSTAT_RD_ERR	Slave captured error on the bus while 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. Slave continually returns 0xFF byte in this case.
SCB_I2C_SSTAT_WR_ERR	Slave captured error on the bus while 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 byte when SCB_I2C_SSTAT_WR_ERR is set. It is recommended to discard write buffer content in this case.

Side Effects: None

2 NAK is an abbreviation for negative acknowledgment or not acknowledged. I^{2}C 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^{2}C slave address
Parameters:	uint32 address: I^{2}C 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 whether it violates the I2C 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^{2}C slave address
Parameters:	uint32 addressMask: I^{2}C 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 I2C slave address. This value may be any between 0 and 254 (0x00 to 0xFE). The LSB of address is R/W bit and it ignored independently of addressMask bit value.
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^{2}C 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^{2}C 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^{2}C master since an 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:	This function returns not valid value 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^{2}C master since an 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:	This function returns not valid value 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^{2}C 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^{2}C 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 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 was occurred while 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 performs start condition generation. As a result, master has automatically switched to slave mode and is responding. The master transaction was 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^{2}C 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.
SCB_I2C_MODE_NO_STOP	Execute transfer without a Stop. The following transfer expected to perform ReStart.

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

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. See the SCB_I2CMasterSendStart() function for constants.
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 and does not return until start condition and address byte are sent and ACK/NACK response is received or errors 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 occurred. Nothing was sent on the bus. The attempt has to be retried.
SCB_I2C_MSTR_NOT_READY	Master is not active master on the bus. The 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 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 begin of Slave operation. This error condition 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. This function is blocking and does not return until start condition and address are sent and ACK/NACK response is received or errors 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.

uint32 SCB_I2CMasterSendStop(void)

Description:	Generates Stop condition on the bus. At least one byte has to be read if 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 transfer, 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 byte is transmitted or error occurred.

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. The 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 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 ACKs or NAKs received byte.

This function does not generate NAK explicitly. The following call SCB_I2CMasterSendStop() or SCB_I2CMasterSendRestart() will generate NAK and Stop or ReStart condition appropriately.

This function is blocking and does not return until byte is received or error occurred.

Parameters: uint32 ackNack: Response to received byte.

Response constants	Description
SCB_I2C_ACK_DATA	Sends ACK response. The master notifies slave that transfer continues and next byte will be read.
SCB_I2C_NAK_DATA	Sends NAK response. The master notifies slave that transfer will be 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.

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 not valid value if SCB_I2C_MSTAT_ERR_ARB_LOST or SCB_I2C_MSTAT_ERR_BUS_ERROR occurred while 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 not valid value if SCB_I2C_MSTAT_ERR_ARB_LOST or SCB_I2C_MSTAT_ERR_BUS_ERROR occurred while 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

Bootloader Support

The SCB component can be used as a communication component for the Bootloader. Only SCB in I2C 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 caller 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 caller 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 I^2C component and enables its interrupt. Every incoming I^2C write transaction is treated as a command for the bootloader. Every incoming I^2C 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 I^{2}C component and disables its interrupt.
Parameters:	None
Return Value:	None
Side Effects:	None

void SCB_CyBtldrCommReset(void)

Description:	Sets read and write I^{2}C buffers to the initial state and resets the slave status.
Parameters:	None
Return Value:	None
Side Effects:	None

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

Description:	Allows the caller 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 send to the device. uint16 size: Number of bytes to write. uint16 *count: Pointer to variable to write the number of bytes actually written. 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_CyBtldrCommWrite(const uint8 pData[], uint16 size, uint16 * count, uint8 timeOut)

Description:	Allows the caller 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 device. uint16 size: Number of bytes to write. uint16 *count: Pointer to variable to write the number of bytes actually written. 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^{2}C Save, 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^{2}C 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^{2}C master/slave.

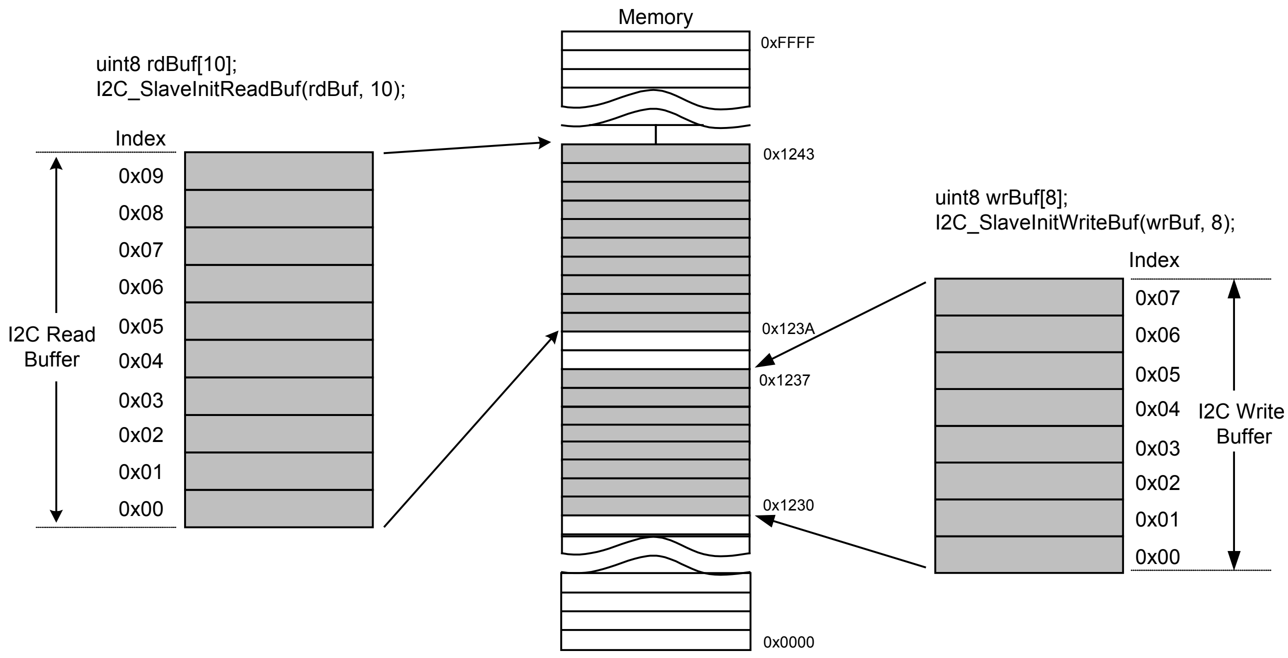
Slave Operation

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 I^{2}C master. The I^{2}C 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() functions are 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 I^2C 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.

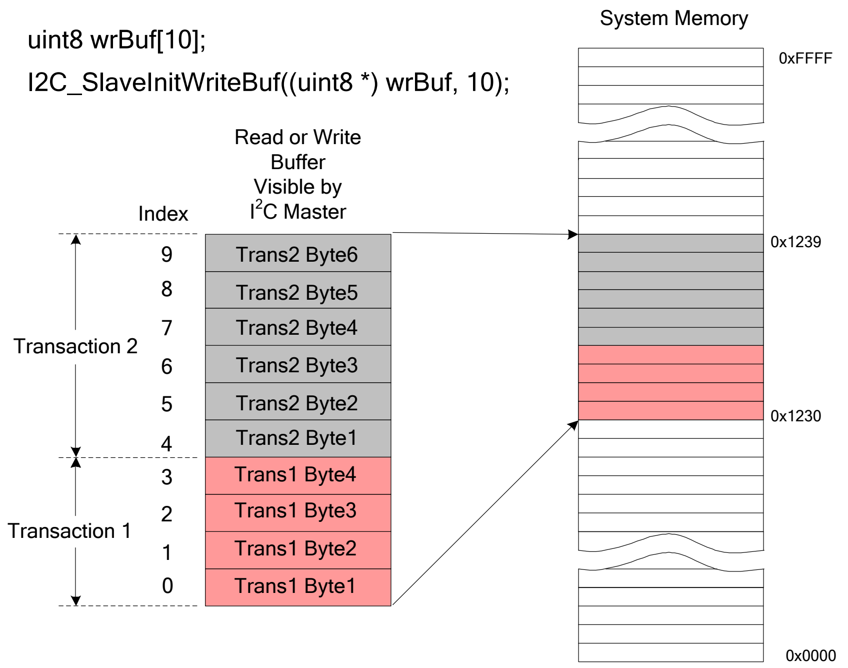
```
void SCB_I2CSlaveClearReadBuf(void)
void SCB_I2CSlaveClearWriteBuf(void)
```

This resets the index back to zero. The next byte read or written to 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[10];
uint8 userArray[10];
uint32 byteCnt;
/* Initialize write buffer before call SCB_Start */
SCB_I2CSlaveInitWriteBuf((uint8 *) wrBuf, 10);

/* Start I2C Slave operation */
SCB_I2CStart();

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

for(;;)  /* loop forever  */
{
   /* Wait for I2C master to complete a write */
   if(0u != (SCB_I2CSlaveStatus() & SCB_I2C_SSTAT_WR_CMPLT))
   {
      byteCnt = SCB_I2CSlaveGetWriteBufSize();
      SCB_I2CSlaveClearWriteStatus();
      for(i=0; i < byteCnt; i++)
      {
         userArray[i] = wrBuf[i]; /* Transfer data */
      }
      SCB_I2CSlaveClearWriteBuf();
   }
}
```

Note All slave status and buffer operation APIs are not interrupt protected and may be modified by I2C ISR. It is preferred to disable I2C interrupt while status and buffer processing. The interrupt disabling blocks slave operation and causes SCL clock stretching.

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 an 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 the 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, xferData. 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^{2}C 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 I^{2}C 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 whether wining master address it. In case of address match the slave becomes active.

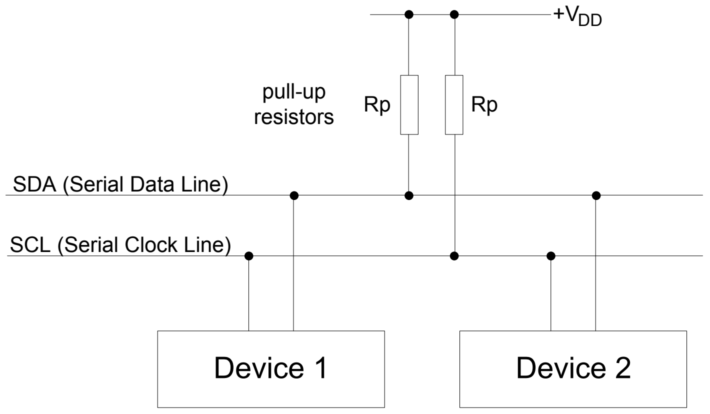
For Master and Slave operation examples look at the Slave Operation and Master sections.

Note All slave status and buffer operation APIs are not interrupt protected and may be modified by I2C ISR. It is preferred to disable I2C interrupt while status and buffer processing. The interrupt disabling blocks slave operation and causes SCL clock stretching.

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 I²C 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 is I^2C mode is able to be wakeup source from Sleep and Deep Sleep low power modes.

The Sleep mode is identical to Active from peripheral point of view. It is not required any configuration changes in component or the code to be called before enter/exit this mode. Any commination intended to the slave causes interrupt to occur and leads to wake up. Any master activity which involves interrupt to occur leads to wake up.

The master modes are not able to be wakeup source from Deep Sleep. This capability is only available for slave modes. The slave has to be configured properly to enable this functionality. The “Enable wakeup from Sleep mode” must be checked in the slave configuration dialog. The SCB_Sleep() and SCB_Wakeup() has to be called before/after enter/exit Deep Sleep.

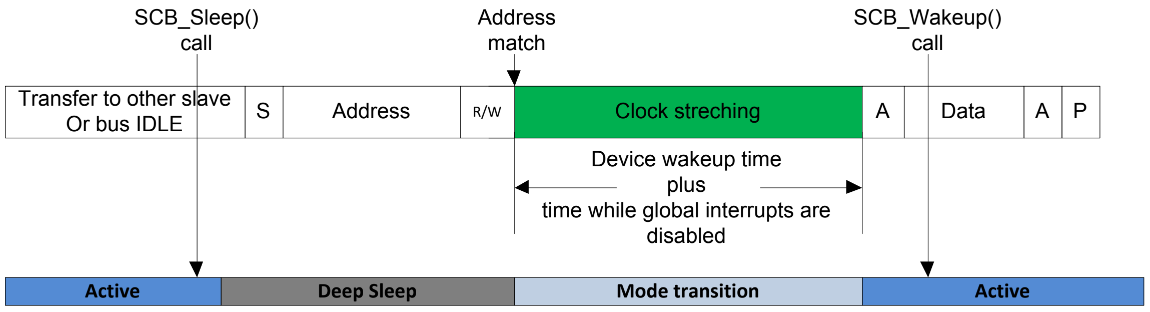
The wakeup event is slave address match. The externally clocked logic performs address matching and when it was occurred the wakeup interrupt triggers. The slave stretches SCL line until control is passed to its interrupt routine to ACK the address.

Before enter Deep Sleep the on-going transaction intended to the slave has to be completed therefore following code is suggested:

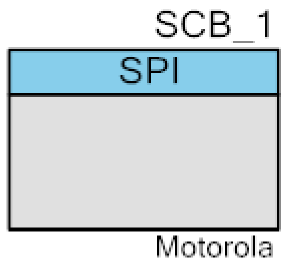
```
CyGlobalIntDisable; /* Disables all interrupts to lock the I2C bus state */

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

if(0u == status)
```

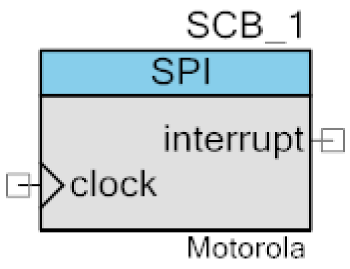
```
    {
        SCB_Sleep(); /* Configure the slave to be wakeup source */

        CySysPmDeepSleep();

        CyGlobalIntEnable; /* Enable all interrupts to unlock I2C bus state */

        SCB_Wakeup(); /* Configure the slave to active mode operation */
    }
    else
    {
        CyGlobalIntEnable; /* Transaction in progress: do not enter to Deep Sleep */
    }
```

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



SPI





The component provides an industry-standard, 4-wire SPI interface. The original SPI protocol was defined by Motorola. The component supports three additional modes, allowing communication with any SPI device. 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 I/Os indicates that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.

clock – Input*

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

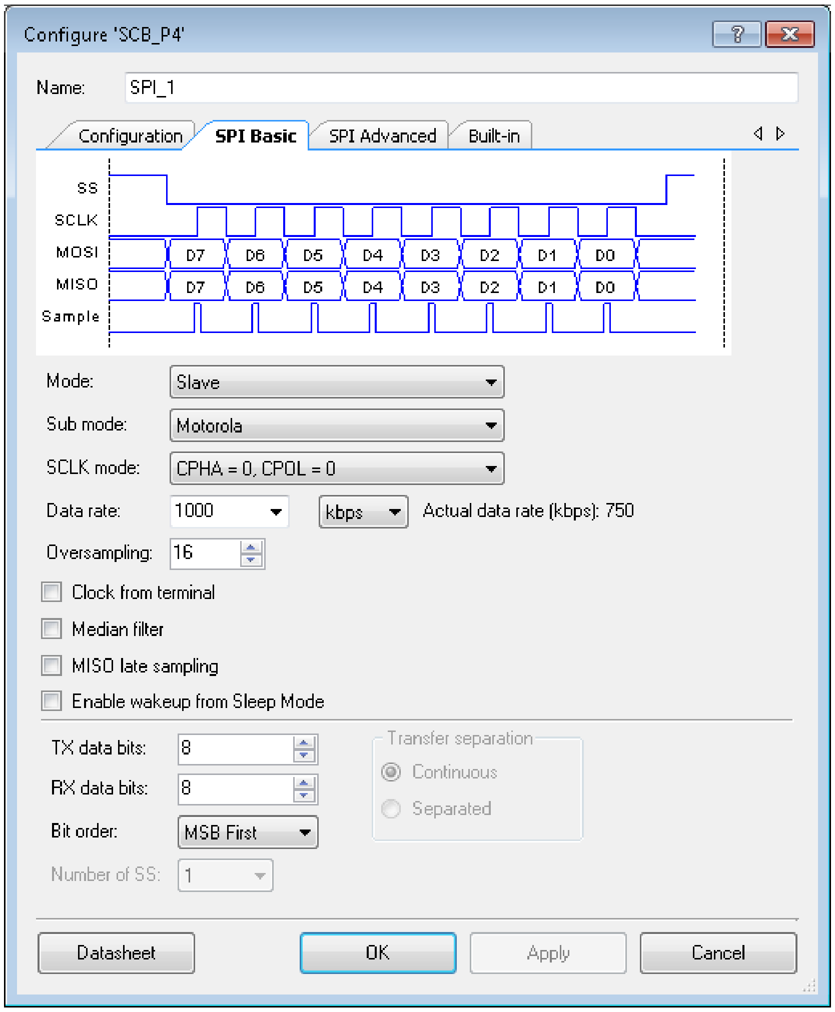
interrupt – Output*

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 chip *Technical Reference Manual (TRM)* for more information.

Note The input buffer of buried output pins is disabled to not 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



Name: SPI_1

Mode: Slave

Sub mode: Motorola

SCLK mode: CPHA = 0, CPOL = 0

Data rate: 1000 kbps Actual data rate (kbps): 750

Oversampling: 16

☐ Clock from terminal

☐ Median filter

☐ MISO late sampling

☐ Enable wakeup from Sleep Mode

TX data bits: 8

RX data bits: 8

Bit order: MSB First

Number of SS: 1

Transfer separation: [✓ Continuous] Separated

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

Mode

This option determines in what SPI mode the SCB will be operated.

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

Sub mode

This option determines what SPI sub-modes are supported, Motorola, TI (Start Coincides), TI (Start Precedes), or National Semiconductor's Microwire protocol.

- **Motorola** – The original SPI protocol is defined by Motorola (default).

- **TI (Start Coincides)** – The Texas Instruments' SPI protocol.
- **TI (Start Precedes)** – The Texas Instruments' SPI protocol.
- **National Semiconductor** – The National Semiconductor's Microwire protocol.

SCLK mode

The parameter defines the clock phase and clock polarity mode you want to use in the communication. The CPHA and CPOL selection provided.

- **CPHA = 0, CPOL= 0 –** Data is driven on a falling edge of SCLK. Data is captured on a rising edge of SCLK. 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.
- **CPHA = 1, CPOL= 0** – Data is driven on a rising edge of SCLK. Data is captured on a falling edge of SCLK.
- **CPHA = 1, CPOL= 1** – Data is driven on a falling edge of SCLK. Data is captured on a rising edge of SCLK.

Clock from terminal

This parameter allows choosing between an internally configured clock and an externally configured clock for data rate generation. When option is enabled, the component does not control the data rate but displays the actual data rate based on the user-connected clock source. Otherwise, PSoC Creator calculates and configures the required clock frequency based on the **Data rate** parameter, taking into account the oversampling.

Note When setting the data rate or external clock frequency value, make sure that PSoC Creator can provide this value using the current system clock frequency. Otherwise, a warning about the clock accuracy range is generated while building the project. This warning contains the actual clock value set by PSoC Creator. Choose whether the system clock or component clock should be changed to fit the clocking system requirements and achieve an optimal value.

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 speed and divider range. The standard bit rates are 500, 1000 (default), 2000, 4000 to 8000 in multiples of 2000 kbps. This parameter has no affect if the **Clock from terminal** parameter is enabled.

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 for defined **Data rate**. For **Master** mode the oversampling value is written into the register and used for SPI clock generation. The even **Oversampling factor** results that the first and second phase of the SPI clock period are the same. Otherwise the first phase of the SPI clock period is 1 one component clock cycle longer than the second phase. For **Slave** only the component clock source is important which has to provide required amount of the oversampling. An oversampling factor maximum value is 16 and minimum depends on component settings.

For **Master** the minimum oversampling factor value depends on **Median filter** and **Enable late MISO sample** settings:

- **Median filter** is unchecked, **Enable late MISO sample** is unchecked – 6
- **Median filter** is unchecked, **Enable late MISO sample** is checked – 3
- **Median filter** is checked, **Enable late MISO sample** is unchecked – 8
- **Median filter** is checked, **Enable late MISO sample** is checked – 4

For **Slave** the **Enable late MISO sample** is not applicable and minimum oversampling value only depends on **Median filter**. Median filter is unchecked – 6, Median filter is checked – 8.

The default is 16.

Median filter

This parameter applies 3 taps digital median filter on input path of MISO. This filter reduces the susceptibility to errors. However, minimum oversampling factor value is increased. The default value is a **Disabled**.

Enable late MISO sample

This option allows changing the SCLK edge on which MISO is captured (only applicable for **Master** mode). The default value is a **Disabled**.

Enable wakeup from Sleep Mode

This option allows the system to be awakened from sleep when slave select occurs.

Refer to the Low Power modes section under 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 same for **Motorola** and **Texas Instruments** sub-modes and 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

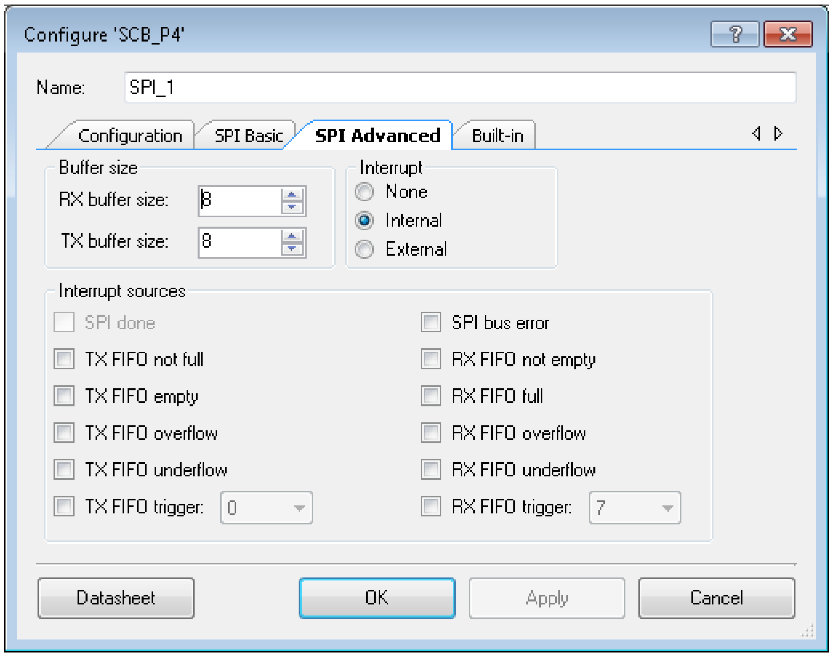
This parameter determines the number of SPI slave select lines. The default number of lines is a 1 line. Any integer from 1 to 4 is a valid setting. This option is only valid in **Master** mode.

Transfer separation

The **Transfer separation** parameter defines if individual data transfers are separated by slave select de-selection. These modes are defined in the following table. See table below for more information.

- **Continuous** – The SS goes low at the start of transfer and goes high when transfer completes (default)
- **Separated** – Every data frame 4 -16 bits is separated by SS de-selection by one SCLK period

Advanced SPI Parameters



Name: SPI_1

Buffer size
RX buffer size: 8
TX buffer size: 8

Interrupt
None [✓ Internal] External

Interrupt sources
- [ ] SPI done
- [ ] SPI bus error
- [ ] TX FIFO not full
- [ ] RX FIFO not empty
- [ ] TX FIFO empty
- [ ] RX FIFO full
- [ ] TX FIFO overflow
- [ ] RX FIFO overflow
- [ ] TX FIFO underflow
- [ ] RX FIFO underflow
- [ ] TX FIFO trigger: 0
- [ ] RX FIFO trigger: 7

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 circular receive data buffer. If this parameter is set to 8, the eight bytes/words RX FIFO is implemented in the hardware. All other values up to 2^{32} use the 8-bytes/words RX FIFO and a software buffer controlled by the supplied APIs and internal ISR. The buffer size is limited only available memory. Interrupt mode sets to internal automatically if RX buffer size is greater than 8.

TX buffer size

The **TX buffer size** parameter defines the size (in bytes/words) of memory allocated for a circular transmit data buffer. If this parameter is set to 8, the eight bytes/words TX FIFO is implemented in the hardware. All other values up to 2^{32} use the 8-bytes/words RX FIFO and a software buffer controlled by the supplied APIs and internal ISR. The buffer size is limited only available memory. Interrupt mode sets to internal automatically if TX buffer size is greater than 8.

Interrupt

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

- **None** – Removes internal interrupt component
- **Internal** – This option leaves interrupt component inside SCB component. The predefined internal ISR is hooked up to interrupt. The customer function can be registered to call on

every entry to ISR. The **Interrupt sources** option defines interrupt events to trigger interrupt.

- **External** – This option removes internal interrupt and provides output terminal. Only the interrupt component can be connected to it if customer interrupt handler desired. The **Interrupt sources** option sets interrupt source which triggers interrupt output.

Note For buffer sizes greater than 8 bytes/words, 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 SPI supports interrupts on the following events:

- **SPI done –** Master transfer done event: all data frames in the TX FIFO are sent and the TX FIFO is empty
- **TX FIFO not full –** TX FIFO is not full
- **TX FIFO empty –** TX FIFO is empty
- **TX FIFO overflow –** Attempt to write to a full TX FIFO
- **TX FIFO underflow –** Attempt to read from an empty TX FIFO
- **TX FIFO trigger –** When the TX FIFO has less entries than the amount of this field, a transmitter trigger event is generated
- **SPI bus error –** SPI slave deselected at an unexpected time in the SPI transfer
- **RX FIFO not empty –** RX FIFO is not empty
- **RX FIFO full –** RX FIFO is full
- **RX FIFO overflow –** Attempt to write to a full RX FIFO
- **RX FIFO underflow –** Attempt to read from an empty RX FIFO
- **RX FIFO trigger –** When the RX FIFO has more entries than the number of this field, a receiver trigger event is generated.

Note

When **RX buffer size** is greater than 8 bytes/words, the **RX FIFO not empty** interrupt source is reserved by the component and used for internal software buffer operations.

When **TX buffer size** is greater than 8 bytes/words, the **TX FIFO not full** interrupt source is reserved by the component and used for internal software buffer operations.

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_Init()	Initialize the SCB component according to defined parameters in the customizer.
SCB_Enable()	Enables SCB component operation.
SCB_Start()	Starts the SCB.
SCB_Stop()	Disable the SCB component.
SCB_Sleep()	Prepares component to enter Deep Sleep.
SCB_Wakeup()	Prepares component to exit Deep Sleep.
SCB_SpiInit()	Configures the SCB for SPI operation.
SCB_SpiSetActiveSlaveSelect()	Selects the active slave select line. Only applicable in Master mode.
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.

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

void SCB_Init(void)

Description:	Initializes the SCB component to operate in one of the selected configurations: I^{2}C, SPI, UART or EZ I^{2}C. 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. 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^{2}C, SPI, UART or EZ I^{2}C. Otherwise this function does not enable component.
Parameters:	None
Return Value:	None
Side Effects:	None

void SCB_Start(void)

Description:	Invokes SCB_Init() and SCB_Enable(). After this function call the component is enabled and ready for 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_Stop(void)

Description:	Disables the SCB component and its interrupt.
Parameters:	None
Return Value:	None
Side Effects:	None

void SCB_Sleep(void)

Description:	Prepares component to enter Deep Sleep. The “Enable wakeup from Sleep Mode” selection has an influence on this function implementation. 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 exit Deep Sleep.

The "Enable wakeup from Sleep Mode" selection has influence to on this function implementation.

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.

Parameters (cont.): config (cont.):

Field	Description
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 words: • The value 8 implies the usage of buffering in hardware. • A value greater than 8 results in a software buffer. The SCB_INTR _RX_NOT_EMPTY interrupt has to be enabled to transfer data into the software buffer.
uint8* rxBuffer	Buffer space provided for a RX software buffer: • The NULL pointer must be provided if hardware buffering implies. • The pointer to allocated buffer must be provided if software buffering implies. The buffer size equal to (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 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 8 implies the usage of buffering in hardware. • A value greater than 8 results in a software buffer.

Parameters (cont.): config (cont.):

Field	Description
	•
uint8* txBuffer	Buffer space provided for a TX software buffer: • The NULL pointer must be provided if hardware buffering implies. • The pointer to allocated buffer must be provided if software buffering implies. The buffer size equal to 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 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_TRIGGER • 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 trigger interrupt. This value is written regardless of whether the RX trigger 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_TRIGGER • 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 trigger interrupt. This value is written regardless of whether the TX trigger interrupt source is enabled.

Return Value: None

Side Effects: None

void SCB_SpiSetActiveSlaveSelect(uint32 activeSelect)

Description: Selects the active slave select line. This function is only applicable to SPI Master mode of operation. 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 all transactions (TX FIFO is empty and the SpiDone flag is set)
- This function does not check that these conditions are met. After initialization the active slave select line is 0.

Parameters: uint32 activeSelect: The four lines available to utilize Slave Select function.

Active Slave Select constants	Description
SCB_SPIM_ACTIVE_SS0	The Slave Select 0 line will be active on the following transaction
SCB_SPIM_ACTIVE_SS1	The Slave Select 1 line will be active on the following transaction
SCB_SPIM_ACTIVE_SS2	The Slave Select 2 line will be active on the following transaction
SCB_SPIM_ACTIVE_SS3	The Slave Select 3 line will be active on the following transaction

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 with data to be placed in transmit buffer. The amount of data bits to be transmitted as one array entry depends on TX 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 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 which were placed in the receive buffer.
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

SPI Functional Description

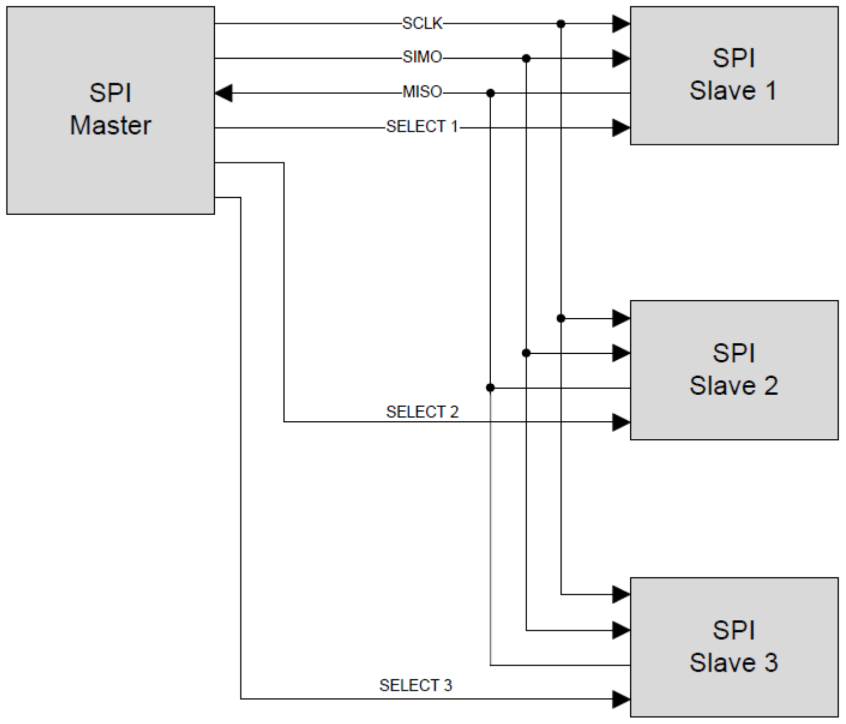
The Serial Peripheral Interface (SPI) protocol is a synchronous serial interface, with “single-master-multi-slave” topology. The original SPI protocol is defined by Motorola. Devices operate in either master or slave mode. The master initiates transfers of data frames. Multiple slaves are supported with 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. 5 SPI Bus Connections Example



Motorola sub mode operation

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

The Motorola SPI protocol has 4 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.
- **CPHA = 0, CPOL= 1** – Data is driven on a rising edge of SCLK. Data is captured on a falling edge of SCLK.
- **CPHA = 1, CPOL= 0** – Data is driven on a rising edge of SCLK. Data is captured on a falling edge of SCLK.

- **CPHA = 1, CPOL= 1** – Data is driven on a falling edge of SCLK. Data is captured on a rising edge of SCLK.

Figure 6 illustrates driving and capturing of MOSI/MISO data as a function of CPOL and CPHA.

Figure 6. SPI Motorola frame format

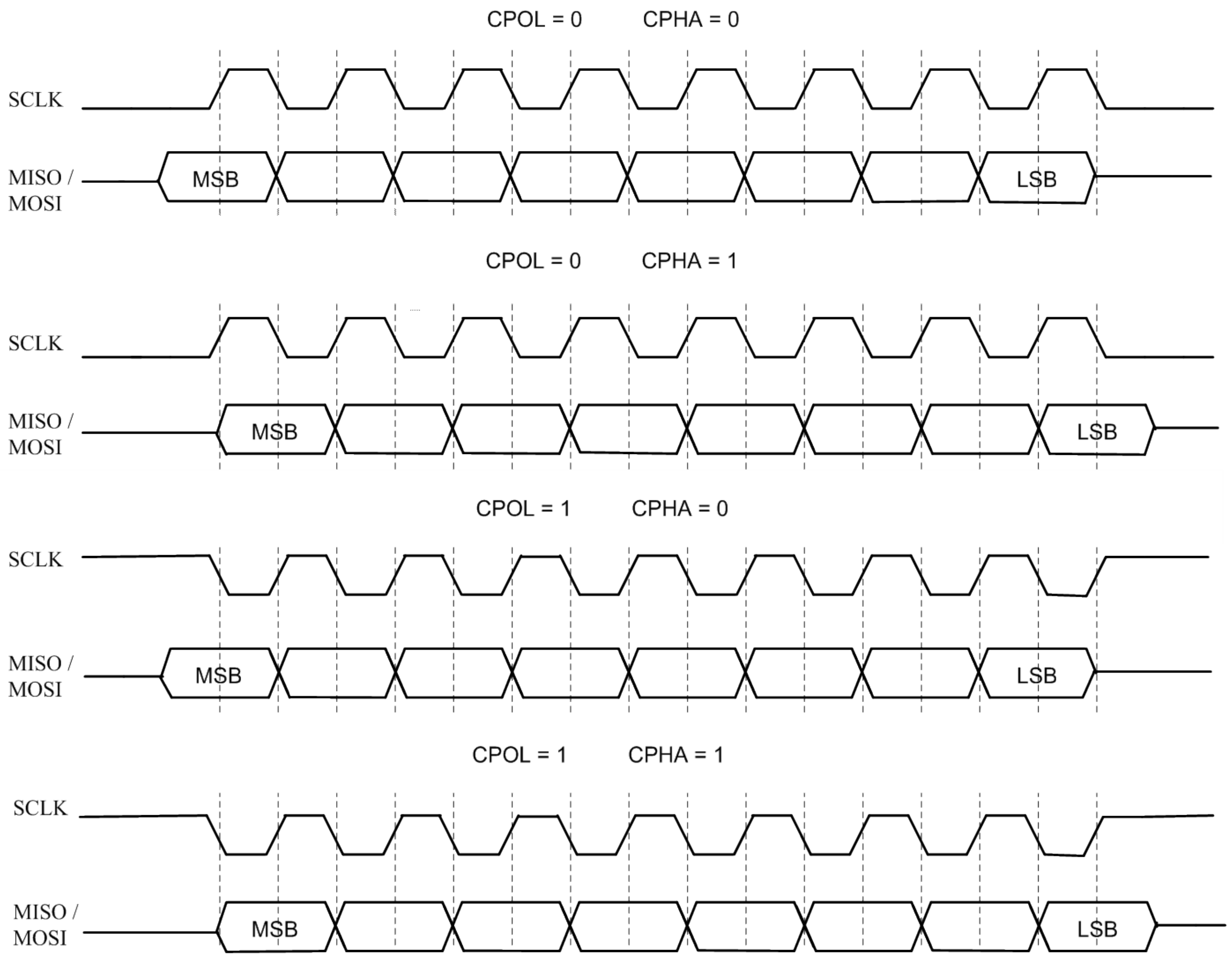
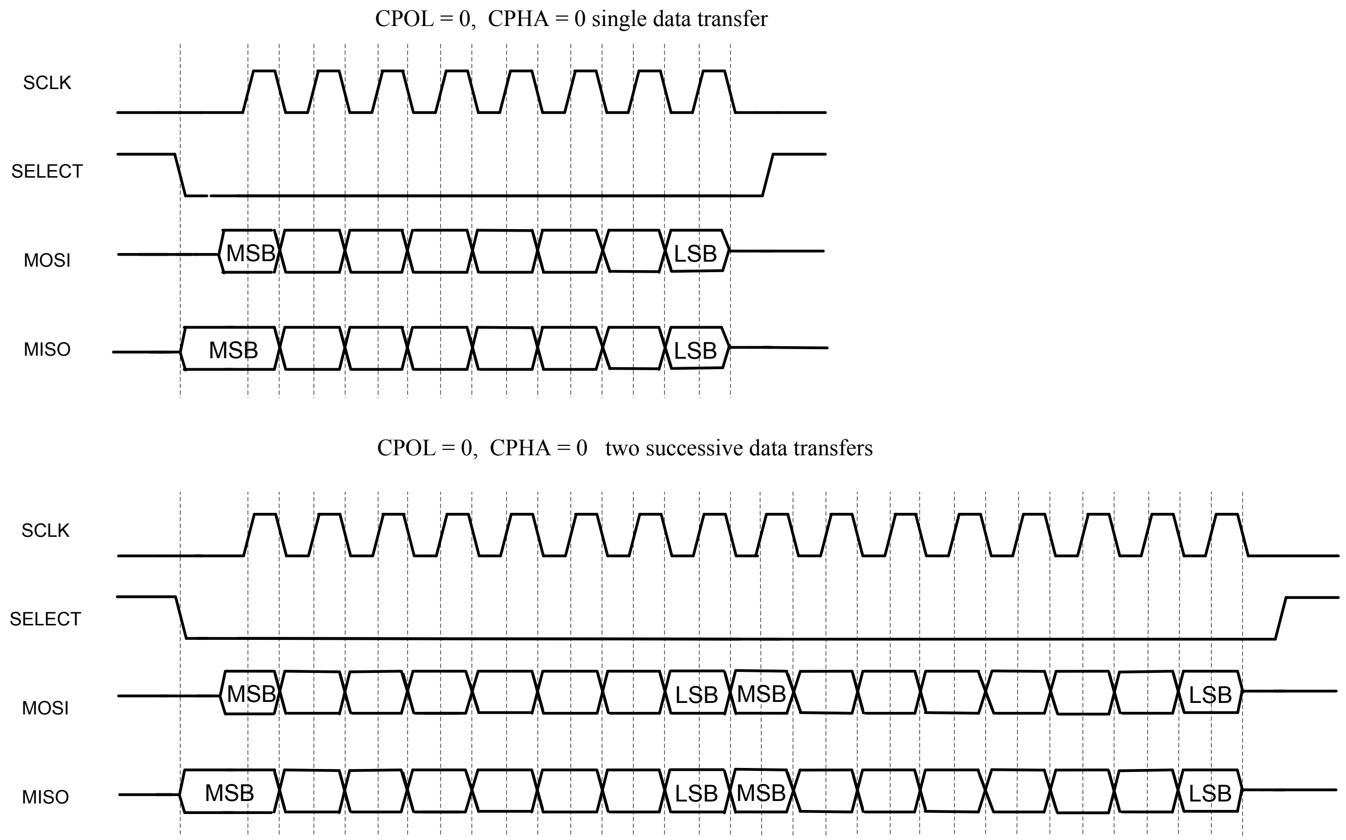


Figure 7 illustrates a single 8-bit data transfer and two successive 8-bit data transfers in mode 0 (CPOL is '0', CPHA is '0').

Figure 7. SPI Motorola Data Transfer Example


CPOL = 0, CPHA = 0 single data transfer
SCLK
SELECT
MOSI
MSB
LSB
MISO
MSB
LSB
CPOL = 0, CPHA = 0 two successive data transfers
SCLK
SELECT
MOSI
MSB
LSB
MSB
LSB
MISO
MSB
LSB
MSB
LSB

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

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

Figure 8. TI (Precede) Data Transfer Example

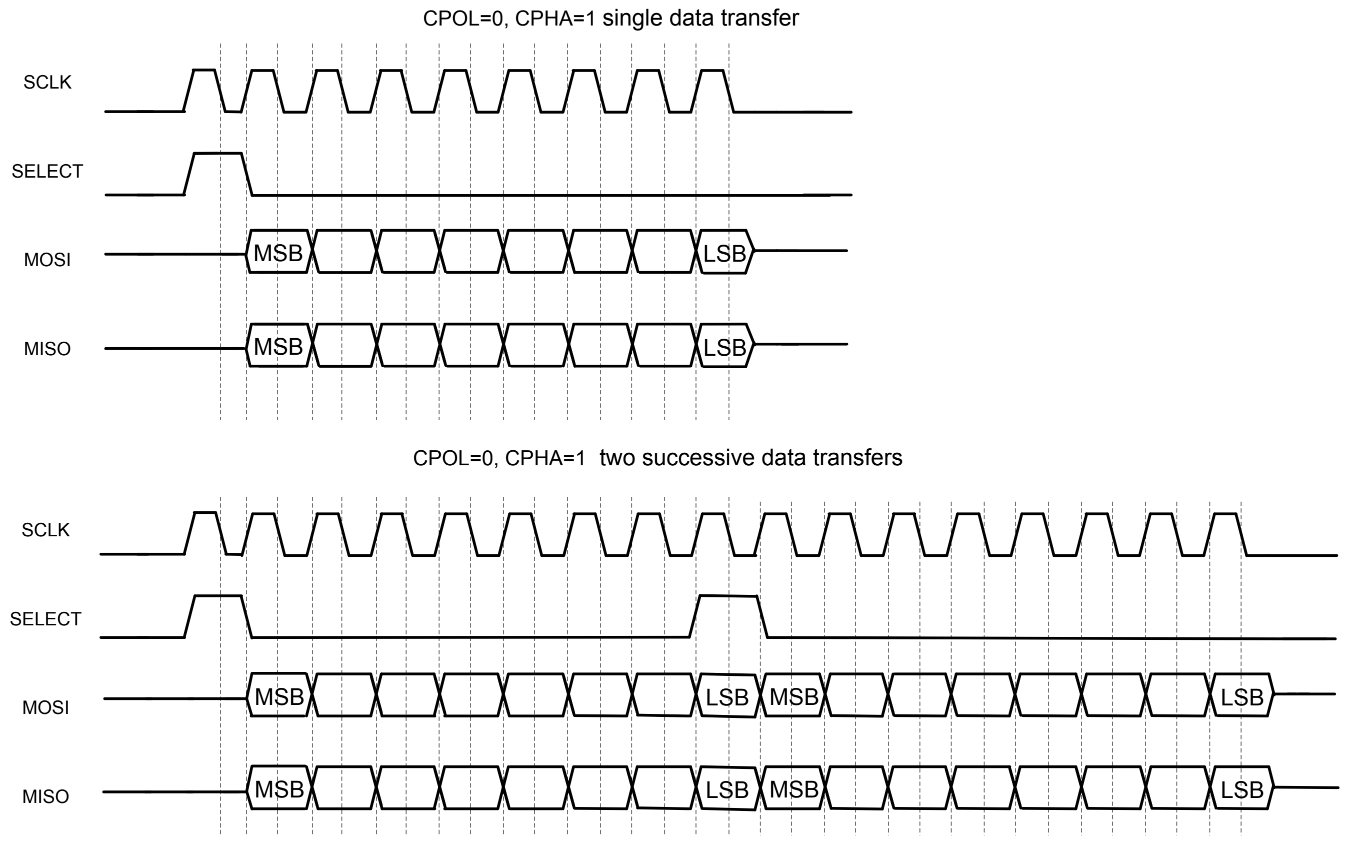
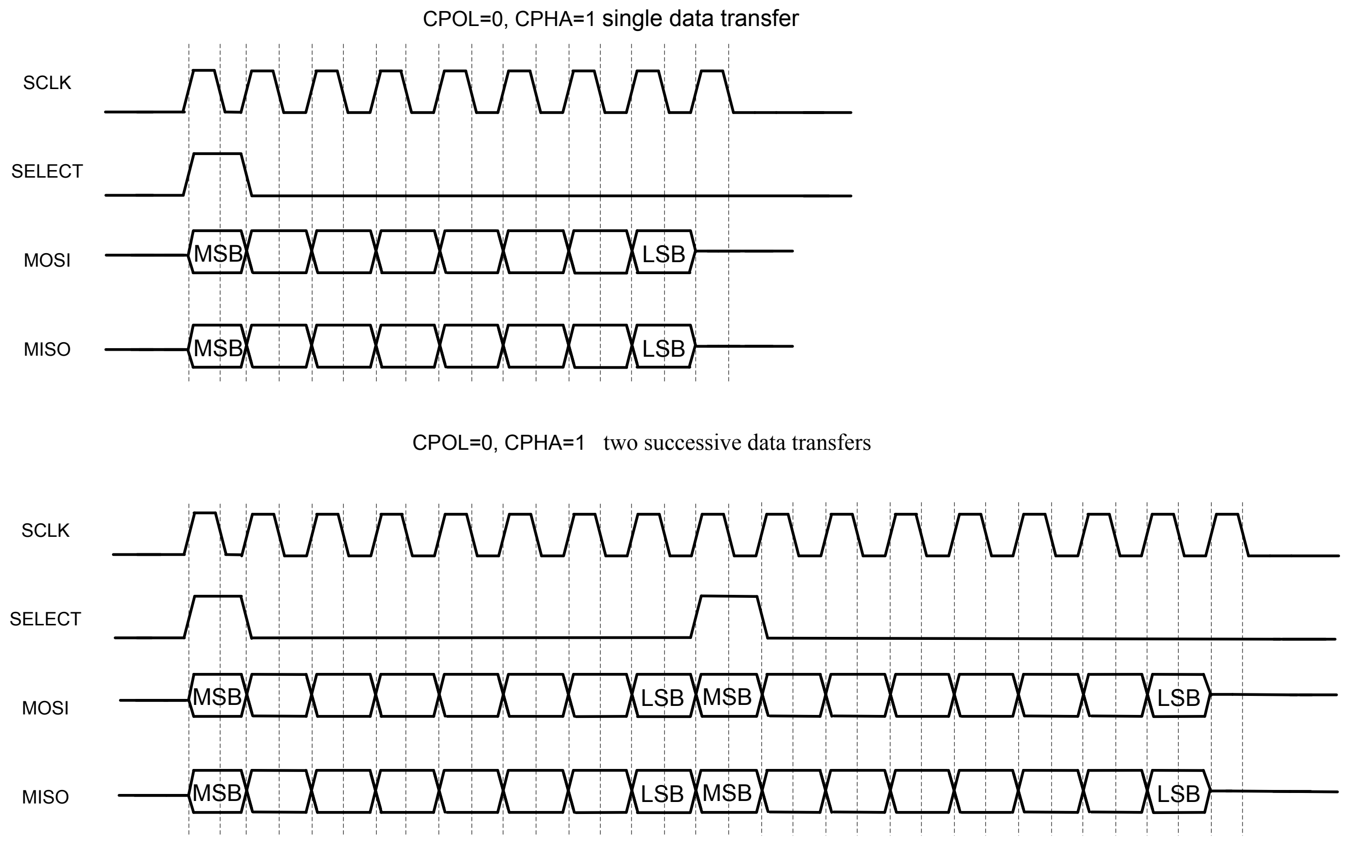

CPOL=0, CPHA=1 single data transfer
SCLK
SELECT
MOSI
MISO
MSB
LSB
MSB
LSB
CPOL=0, CPHA=1 two successive data transfers
SCLK
SELECT
MOSI
MISO
MSB
LSB
MSB
LSB
MSB
LSB
MSB
LSB

Figure 9 illustrates a single 8-bit data transfer and two successive 8-bit data transfers. The SS pulse coincides with the first data bit.

Figure 9. TI (Coincide) Data Transfer Example


CPOL=0, CPHA=1 single data transfer
SCLK
SELECT
MOSI
MISO
MSB
LSB
CPOL=0, CPHA=1 two successive data transfers
SCLK
SELECT
MOSI
MISO
MSB
LSB
MSB
LSB

National Semiconductor’s Microwire sub modes operation

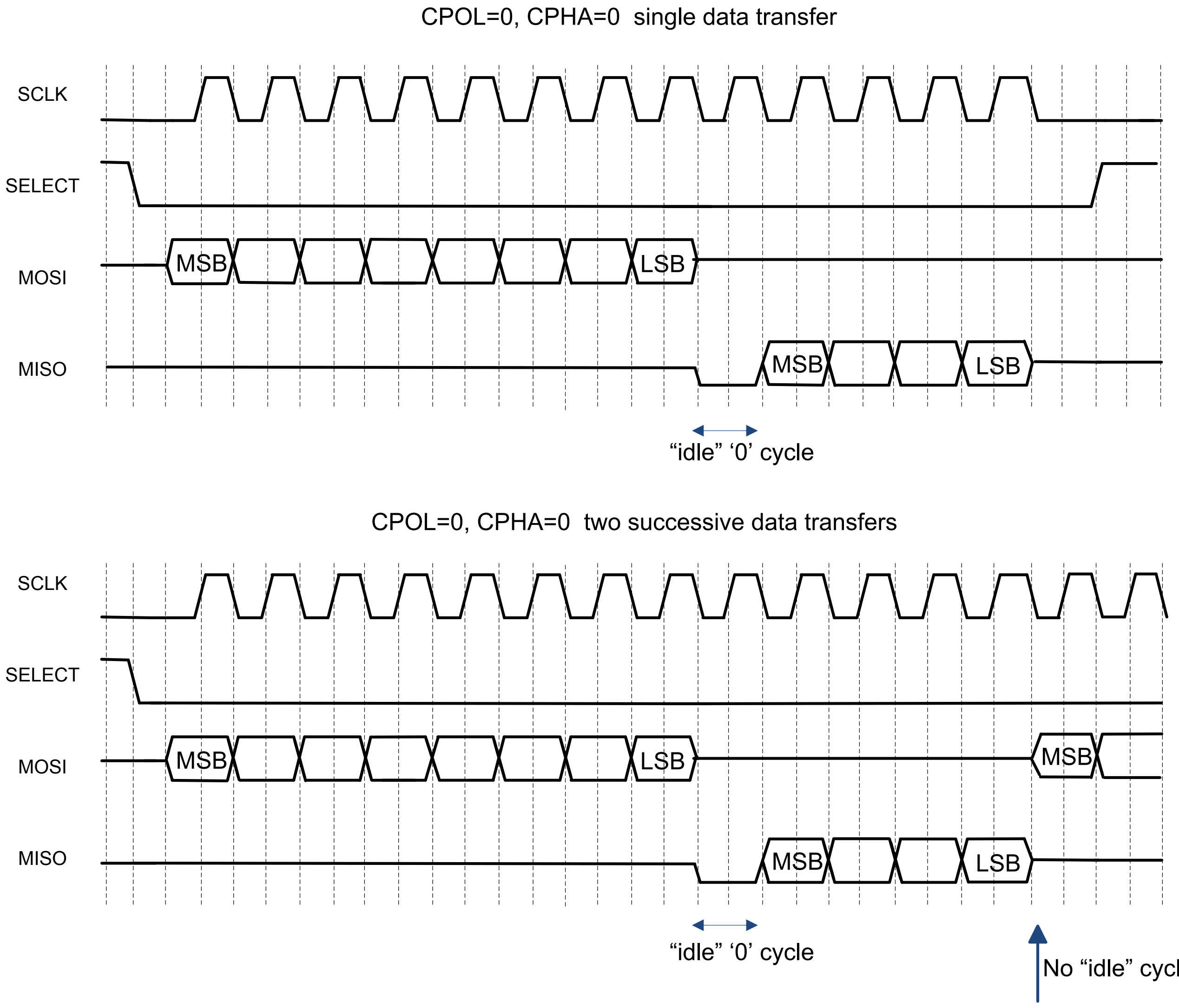
The National Semiconductor's Microwire protocol is a half-duplex protocol. Rather than transmission and reception occurring at the same time, transmission and reception take turns (transmission happens before reception). A single “idle” bit transfer period separates transmission from reception. This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

Note The successive data transfers are NOT separated by an “idle” bit transfer period.

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

Figure 10 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 10. National Semiconductor's Microwire Data Transfer Example



Continuous versus Separated Transfer Separation

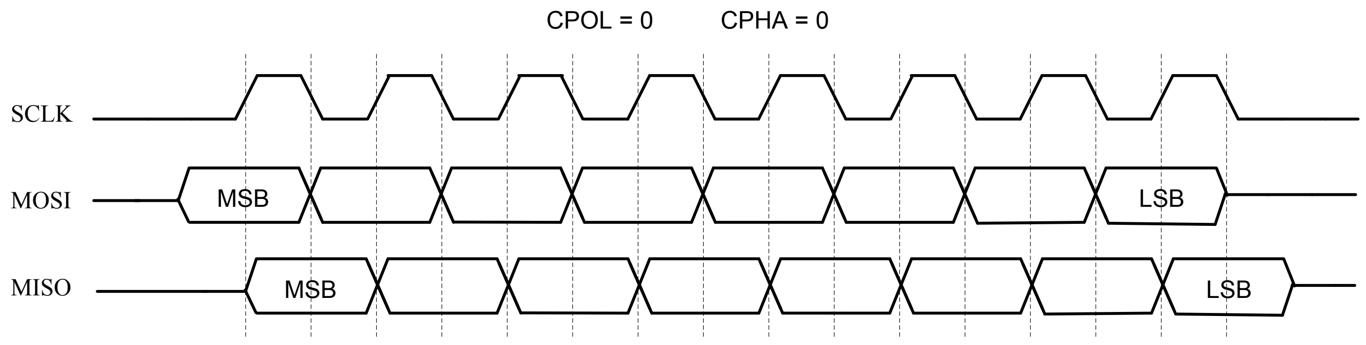
During separated data transfer the SELECT line always changing from '0' to '1' and back from '1' to '0' between the individual transfers. This process repeats for each individual data transfers.

Multiple data transfers may happen without that SELECT line toggling between the individual transfers. Figure 11 illustrates a two continuous 8-bit data transfers in SCLK mode CPHA=0, CPOL= 0.

MISO late sampling

The MISO is captured half a SCLK period later (only applicable in Master mode). 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 11. Late MISO sampling example



Software Buffer

The SCB has a FIFO memory, which is a 16-word by 16-bit SRAM, with byte write enable. In SPI mode, the FIFO is split into TX FIFO and RX FIFO. Each has eight entries of 16 bits per entry. The 16-bit width per entry is used to accommodate configurable data width.

The internal interrupt handler is used to provide interaction between software and hardware TX/RX buffers without any changes to your top-level firmware.

You should also consider that using the software buffer leads to greater timing intervals between transmitted words because of the extra time the interrupt handler needs to execute (depending on the selected HFCLK value).

Interrupts

When **RX buffer size** or **TX buffer size** is greater than 8 bytes/words, the **RX FIFO not empty** and **TX FIFO not full** interrupt sources are reserved by the component for internal software buffer operations. **Do not clear or disable them** because it causes incorrect software buffer operation. But it is user responsibility to clear interrupt events from other sources because they are not cleared automatically. The customer handler function exists to do that.

In case **RX buffer size** or **TX buffer size** is less or equal 8 bytes/words 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

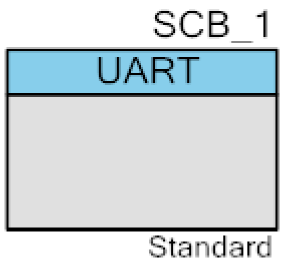
The component in SPI mode is able to be wakeup source from Sleep and Deep Sleep low power modes.

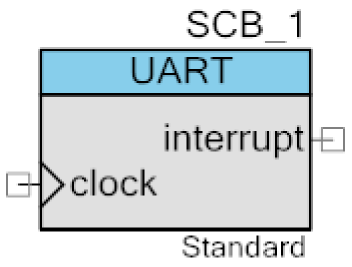
The Sleep mode is identical to Active from peripheral point of view. It implies that it is not required any configuration changes in component or code to be called before enter/exit this mode. Any commination intended to the slave causes interrupt to occur and leads to wake up. Any master activity which involves interrupt to occur leads to wake up.

The master mode is not able to be wakeup source from Deep Sleep. This capability is only available in slave mode. The Deep Sleep mode requires that slave has to be properly configured to be wakeup source. The “Enable wakeup from Sleep mode” must be checked in the slave configuration dialog. The SCB_Sleep() and SCB_Wakeup() has to be called before/after enter/exit Deep Sleep.

In the **Slave** mode operation the device wakes up from sleep on slave select. Waking up takes time and the ongoing SPI transfer is negatively acknowledged – "0xff" bytes are sent out on the MISO line. Master must poll the component again after device wakeup time is passed.

UART





The UART provides asynchronous communications commonly referred to as RS232. Three different UART-like serial interface protocols are supported:

- UART – this is the basic flavor.
- 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 I/Os indicates that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.

clock – Input*

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

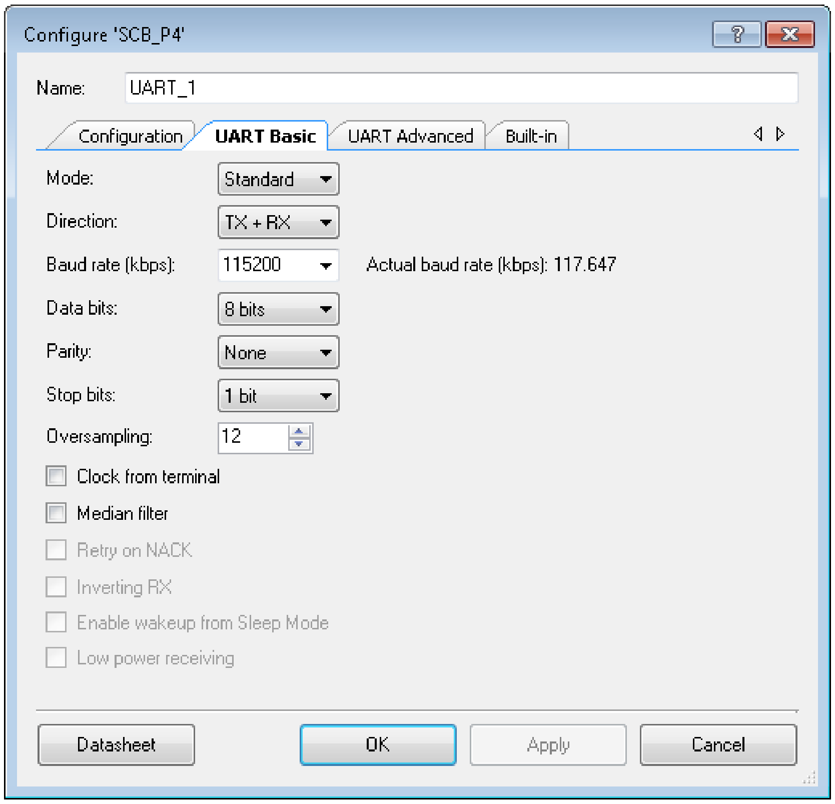
interrupt – Output*

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 chip *Technical Reference Manual (TRM)* for more information.

Note The input buffer of buried output pins is disabled to not 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 UART Parameters


Configure 'SCB_P4'

Name: UART_1

Mode: Standard

Direction: TX + RX

Baud rate (kbps): 115200 Actual baud rate (kbps): 117.647

Data bits: 8 bits

Parity: None

Stop bits: 1 bit

Oversampling: 12

☐ Clock from terminal

☐ Median filter

☐ Retry on NACK

☐ Inverting RX

☐ Enable wakeup from Sleep Mode

☐ Low power receiving

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 or bit-width configuration of the hardware for clock generation up to 921600; the actual rate may differ based on available clock speed and divider range. The default is **115200**.

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. **Oversampling factor** is used to calculate the internal component clock frequency required to achieve this amount of oversampling for defined **Data rate**. An oversampling factor maximum value is 16 and minimum depends on component settings.

For **Standard and Smart Card mode** the minimum oversampling factor value depends on **Median filter** settings. Median filter is unchecked – 6, Median filter is checked – 8. The default is 16.

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

The default is 16.

Clock from terminal

This parameter allows choosing between an internally configured clock and an externally configured clock for data rate generation. When option is enabled, the component does not control the data rate but displays the actual data rate based on the user-connected clock source. Otherwise, PSoC Creator calculates and configures the required clock frequency based on the **Baud rate** parameter, taking into account the oversampling.

Note When setting the data rate or external clock frequency value, make sure that PSoC Creator can provide this value using the current system clock frequency. Otherwise, a warning about the clock accuracy range is generated while building the project. This warning contains the actual clock value set by PSoC Creator. Choose whether the system clock or component clock should be changed to fit the clocking system requirements and achieve an optimal value.

Median filter

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

Retry on NACK

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

Inverting RX

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

Enable wakeup from Sleep Mode

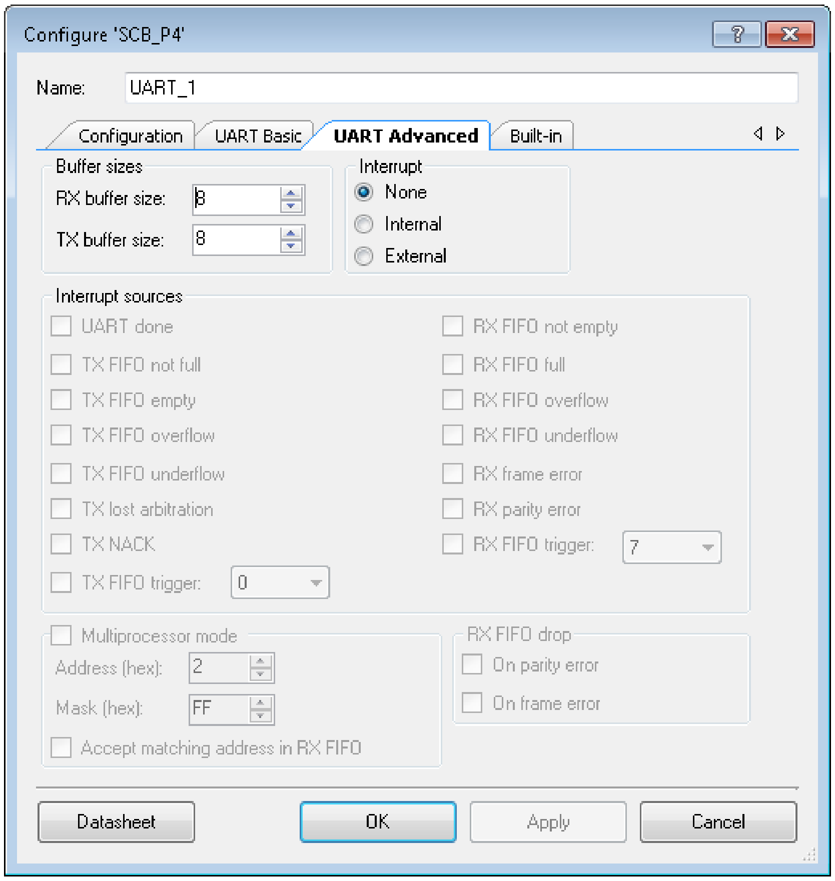
This option allows the system to be awakened from sleep on start bit. This option is applicable only when **RX Direction** is enabled.

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

Advanced UART Parameters


Configure 'SCB_P4'

Name: UART_1

Configuration | UART Basic | UART Advanced | Built-in

Buffer sizes
RX buffer size: 8
TX buffer size: 8

Interrupt
[✓ None] Internal External

Interrupt sources
- [ ] UART done
- [ ] TX FIFO not full
- [ ] TX FIFO empty
- [ ] TX FIFO overflow
- [ ] TX FIFO underflow
- [ ] TX lost arbitration
- [ ] TX NACK
- [ ] TX FIFO trigger: 0
- [ ] RX FIFO not empty
- [ ] RX FIFO full
- [ ] RX FIFO overflow
- [ ] RX FIFO underflow
- [ ] RX frame error
- [ ] RX parity error
- [ ] RX FIFO trigger: 7

- [ ] Multiprocessor mode
Address (hex): 2
Mask (hex): FF
- [ ] Accept matching address in RX FIFO

RX FIFO drop
- [ ] On parity error
- [ ] On frame error

RX buffer size

The **RX buffer size** parameter defines the size (in bytes/words) of memory allocated for a circular receive data buffer. If this parameter is set to 8, the eight bytes/words RX FIFO is implemented in the hardware. All other values up to 2^{32} use the 8-byte/word RX FIFO and a software buffer controlled by the supplied APIs and internal ISR. The buffer size is limited to available memory. Interrupt mode sets to internal automatically if RX buffer size is greater than 8.

TX buffer size

The **TX buffer size** parameter defines the size (in bytes/words) of memory allocated for a circular transmit data buffer. If this parameter is set to 8, the eight bytes/words TX FIFO is implemented in the hardware. All other values up to 2^{32} use the 8-byte/word TX FIFO and a software buffer controlled by the supplied APIs and internal ISR. The buffer size is limited to available memory. Interrupt mode sets to internal automatically if TX buffer size is greater than 8.

Interrupt

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

- **None** – Removes internal interrupt component
- **Internal** – This option leaves interrupt component inside SCB component. The predefined internal ISR is hooked up to interrupt. The customer function can be registered to call on every entry to ISR. The **Interrupt sources** option defines interrupt events to trigger interrupt.
- **External** – This option removes internal interrupt and provides output terminal. The interrupt component can be connected to it if customer interrupt handler desired. The **Interrupt sources** option sets interrupt source which triggers interrupt output.

Note For buffer sizes greater than 8 bytes/words, 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 SPI supports interrupts on the following events:

- **UART done –** UART transmitter done event: all data frames in the TX FIFO are sent and the TX FIFO is empty
- **TX FIFO not full –** TX FIFO is not full
- **TX FIFO empty –** TX FIFO is empty
- **TX FIFO overflow –** Attempt to write to a full 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 underflow –** Attempt to read from an empty TX FIFO.
- **TX FIFO trigger –** When the TX FIFO has less entries than the amount of this field, a transmitter trigger event is generated.
- **RX FIFO not empty –** RX FIFO is not empty.
- **RX FIFO full –** RX FIFO is full.
- **RX FIFO overflow** – Attempt to write to a full RX FIFO.
- **RX FIFO underflow** – Attempt 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 trigger –** When the RX FIFO has more entries than the number of this field, a receiver trigger event is generated.

Note

When **RX buffer size** is greater than 8 bytes/words, the **RX FIFO not empty** interrupt source is reserved by the component.

When **TX buffer size** is greater than 8 bytes/words, the **TX FIFO not full** interrupt is reserved by the component.

Multiprocessor mode

This parameter enables the multiprocessor mode where the first bit of 9 bits 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 or not 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 from 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 captured, received data is sent to the RX FIFO. Otherwise, received data is dropped and lost.

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 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_Init()	Initialize the SCB component according to defined parameters in the customizer.
SCB_Enable()	Enables SCB component operation.
SCB_Start()	Starts the SCB.
SCB_Stop()	Disable the SCB component.
SCB_Sleep()	Prepares component to enter Deep Sleep.
SCB_Wakeup()	Prepares component to exit Deep Sleep.
SCB_UartInit()	Configures the SCB for UART operation.
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.

Function	Description
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_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.

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.

void SCB_Init(void)

Description:	Initializes the SCB component to operate in one of the selected configurations: I^{2}C, SPI, UART or EZ I^{2}C. 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. 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^{2}C, SPI, UART or EZ I^{2}C. Otherwise this function does not enable component.
Parameters:	None
Return Value:	None
Side Effects:	None

void SCB_Start(void)

Description:	Invokes SCB_Init() and SCB_Enable(). After this function call the component is enabled and ready for operation. When configuration is set to “Unconfigured SCB”, the component must first be initialized to operate in one of the following configurations: I^{2}C, SPI, UART or EZ I^{2}C. Otherwise this function does not enable component.
Parameters:	None
Return Value:	None
Side Effects:	None

void SCB_Stop(void)

Description:	Disables the SCB component and its interrupt.
Parameters:	None
Return Value:	None
Side Effects:	None

void SCB_Sleep(void)

Description:	Prepares component to enter Deep Sleep. The “Enable wakeup from Sleep Mode” selection has an influence on this function implementation. 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 exit Deep Sleep. The “Enable wakeup from Sleep Mode” selection has influence to on this function implementation. **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 and lost it if parity error detected. 0 – disable 1 – enable
uint32 dropOnFrameErr	Drop data from RX FIFO and lost it if frame error 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 8 implies the usage of buffering in hardware. • A value greater than 8 results in a software buffer. The SCB_INTR _RX_NOT_EMPTY interrupt has to be enabled to transfer data into the software buffer.
uint8* rxBuffer	Buffer space provided for a RX software buffer: • The NULL pointer must be provided if hardware buffering implies. • The pointer to allocated buffer must be provided if software buffering implies. The buffer size equal to (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 8 implies the usage of buffering in hardware. • A value greater than 8 results in a software buffer.
uint8* txBuffer	Buffer space provided for a TX software buffer: • The NULL pointer must be provided if hardware buffering implies. • The pointer to allocated buffer must be provided if software buffering implies. The buffer size equal to 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_TRIGGER 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 trigger interrupt. This value is written regardless of whether the RX trigger 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_TRIGGER 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 trigger interrupt. This value is written regardless of whether the TX trigger interrupt source is enabled.

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 transmit buffer.

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

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 all 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) to 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 is 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 does not treat 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 15-8 contains status and bits 7-0 contains the next data element from receive buffer. If the bits 15-8 are nonzero, an error has occurred
Side Effects:	The errors bits may not correspond with reading characters due to RX FIFO and software buffer usage. RX software buffer is disabled: The internal software buffer overflow does not treat as an error condition. 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_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 mark to create 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 which were placed in the receive buffer.
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

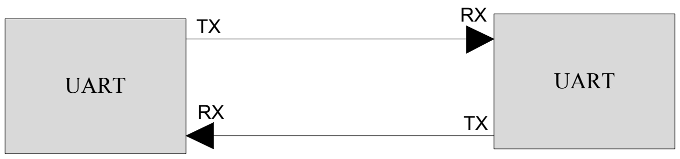
UART Functional Description

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

Note SCB does NOT support RS232 side band signals associated with flow control, such as DTR (Data Terminal Ready), DCD (Data Carrier Detect), etc.

Figure 12. UART typical connection



Standard mode operation

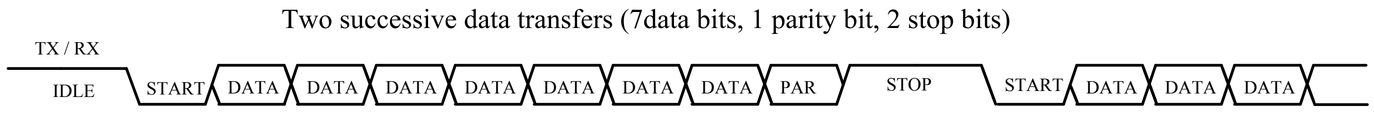
Standard UART is defined with "peer to peer" topology.

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 13. UART Protocol


Two successive data transfers (7data bits, 1 parity bit, 2 stop bits)
TX / RX
IDLE
START
DATA
DATA
DATA
DATA
DATA
DATA
DATA
PAR
STOP
START
DATA
DATA
DATA

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 control by user software before the transfer rather than being calculated based on the data bit values.

```
tx_data = 0x31;
tx_data |= 0x100; /* Set 9th bit to indicate 'address' */
UART_SpiUartWriteTxData(tx_data)
```

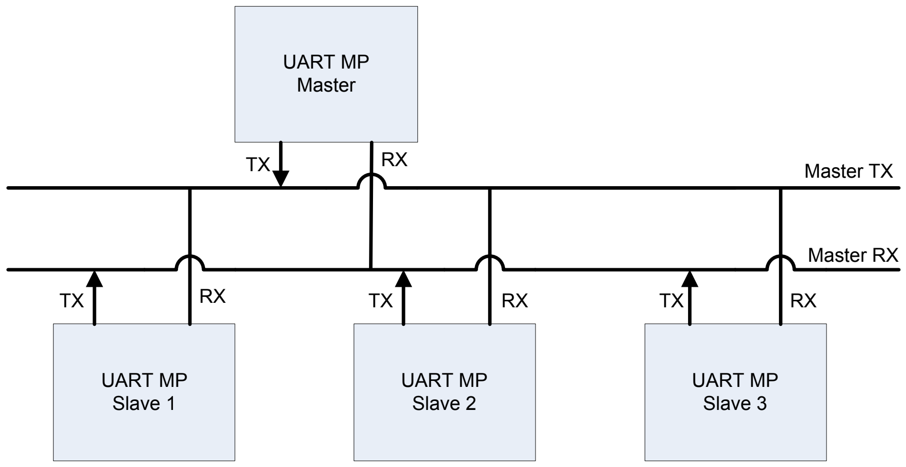
Multiprocessor mode operation

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 5 to 8-bits data field.

The main properties of multiprocessor mode are:

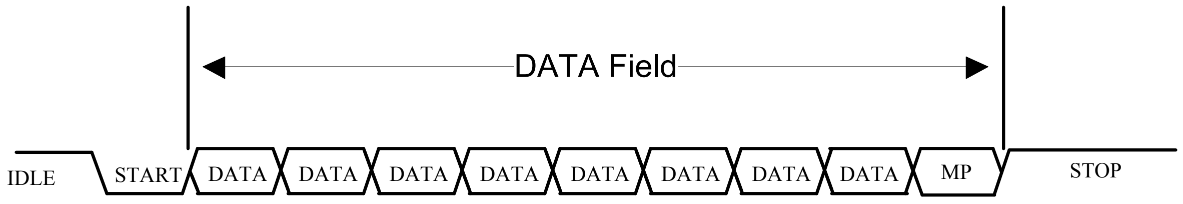
- Single master with multiple slave concept (multi-drop network)
- Each slave is identified by a unique address
- Using 9 bits data field, with the 9th bit (MSB) as address/data flag. When set ‘1’, it indicates an address byte; when set ‘0’ it indicates a data byte.
- Parity bit is disabled

Figure 14. Multiprocessor Bus Connections



To enable Multiprocessor mode configure the UART with the following options: **Mode**: Standard, **Data bits**: 9 bits, **Parity**: None.

Figure 15. 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 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 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 are dropped, until next address received for compare.

SmartCard (ISO7816) mode operation

ISO7816 is asynchronous serial interface, defined with “single-master-single-slave” topology. Only master (reader) function is supported in component.

SCB provides the basic physical layer support with asynchronous character transmission, and only “I/O” pin interface of standard ISO7816[3] 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 are left for firmware to handle from the user level.

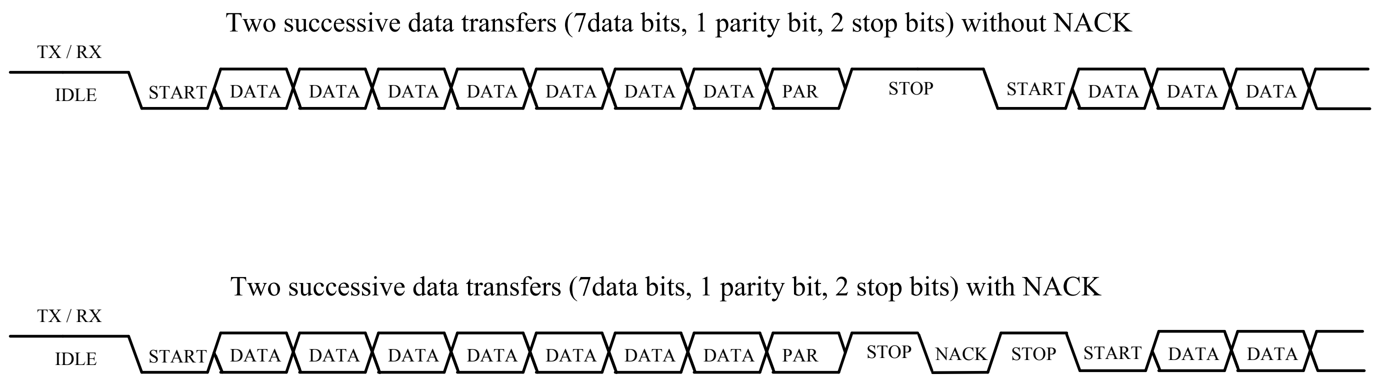
SmartCard data transfer

The SmartCard transfer is similar to a UART transfer, with the addition of a negative acknowledgement (NACK) that may be send 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 16 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”).

3 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 16. SmartCard Data Transfer Example


Two successive data transfers (7data bits, 1 parity bit, 2 stop bits) without NACK
TX / RX
IDLE
START
DATA
DATA
DATA
DATA
DATA
DATA
DATA
PAR
STOP
START
DATA
DATA
DATA
Two successive data transfers (7data bits, 1 parity bit, 2 stop bits) with NACK
TX / RX
IDLE
START
DATA
DATA
DATA
DATA
DATA
DATA
DATA
PAR
STOP
NACK
STOP
START
DATA
DATA

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.

Protocol T=0 and T=1

There are two transmission protocols in ISO7816 specification, T=0 (half-duplex transmission of asynchronous characters), and T=1 (half duplex transmission of asynchronous blocks). At physical layer, T=1 blocks are implemented with asynchronous characters.

Answer-to-reset (ATR)

By definition, the Answer-To-Reset is the value of the sequence of the bytes sent by the card (slave) to the interface device (reader or master), as the answer to a reset. On the I/O line, each byte is conveyed in an asynchronous character.

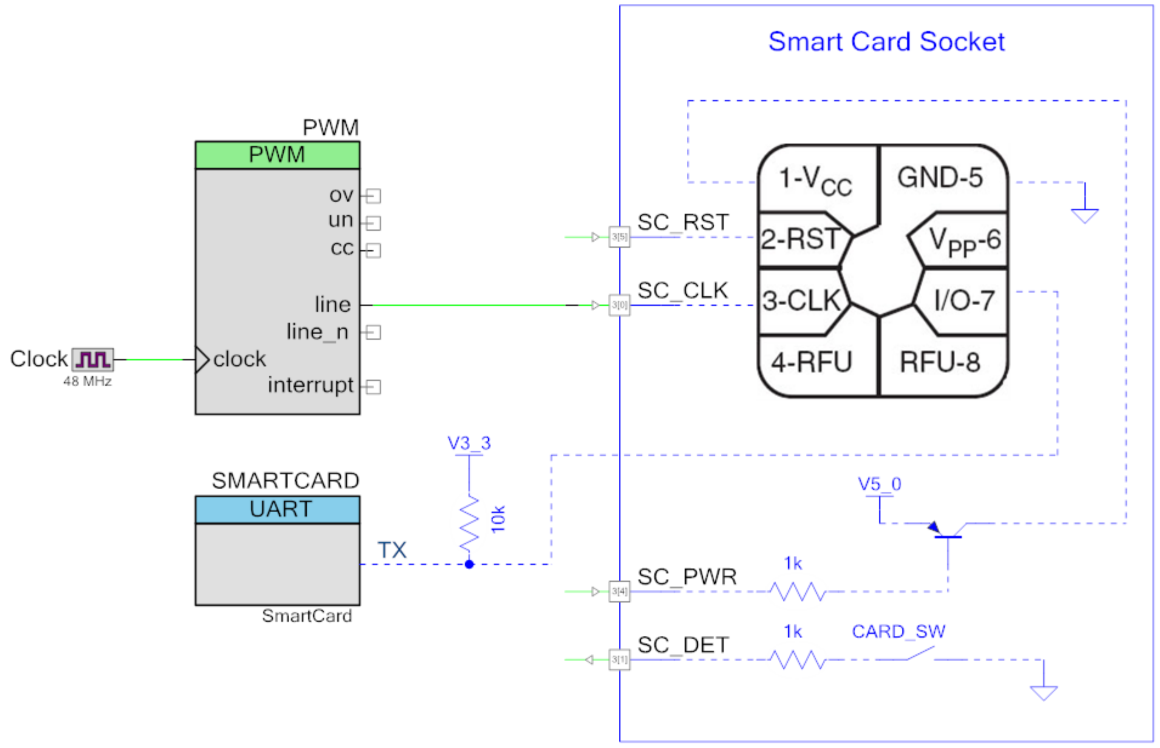
The process of communication setup (ATR, answer to reset), PPS (protocol and parameter selection), selection of classes of operating conditions, operation mode selection and switching, retransmission on NACK, and other high level protocol implementation is left for user firmware to handle.

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 17 is example of implementing SmartCard reader function with TCPWM and pins components.

Figure 17. SmartCard reader implementation example



The UART component is connected to I/O card contact. 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[4] 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 are left for firmware to handle from the user level.

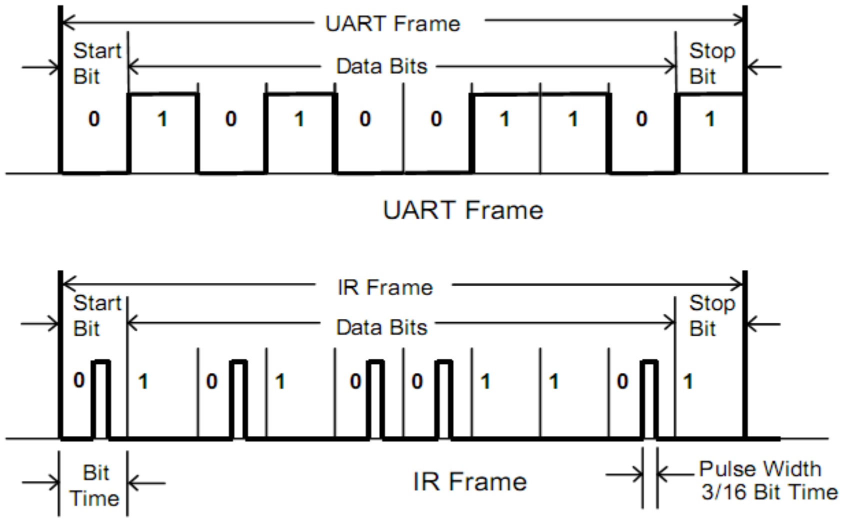
[4] Refer to the *IrPHY (IrDA Physical Layer Link Specification)* (Rev. 1.4 from May 2001) on the IrDA web site at www.irda.org

The 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.

The Figure 18 shows UART frame and IR frame, comprised a Start Bit, 8 Data Bits, no Parity Bit and ending with a Stop Bit.

Figure 18. UART Frame and IR Frame example



Oversempling 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:

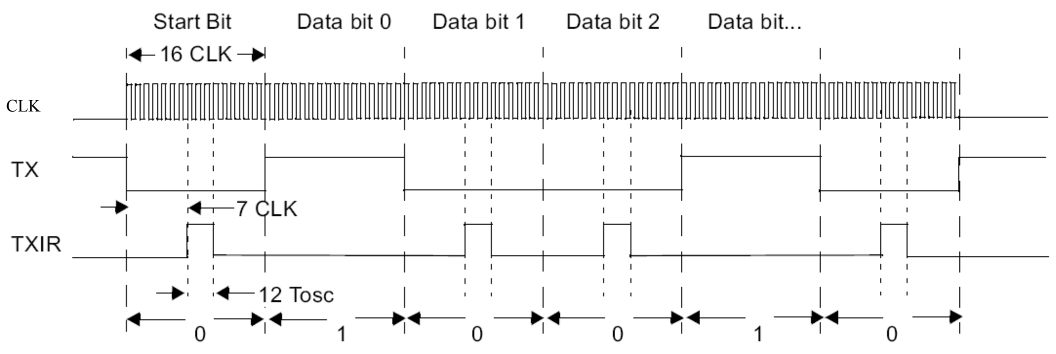
- **Normal transmission** – pulse width is roughly 3/16 of the bit period (for all baud rates)
- **Low power transmission** – pulse width is potentially smaller (down to 1.62µs typical and 1.41µs minimal) than 3/16 of the bit period (for rates less 115200 bps). Supported only for **RX only** direction.

Inverting RX

This option is used for support two possible demodulation scheme described below.

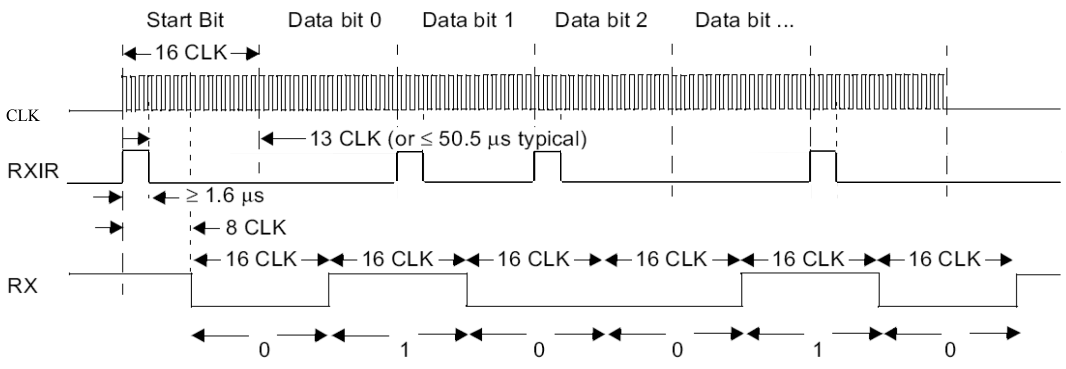
According to IrPHY specefication, IR frame modulation (encoding) scheme is shown on Figure 19.

Figure 19. IR frame modulation scheme



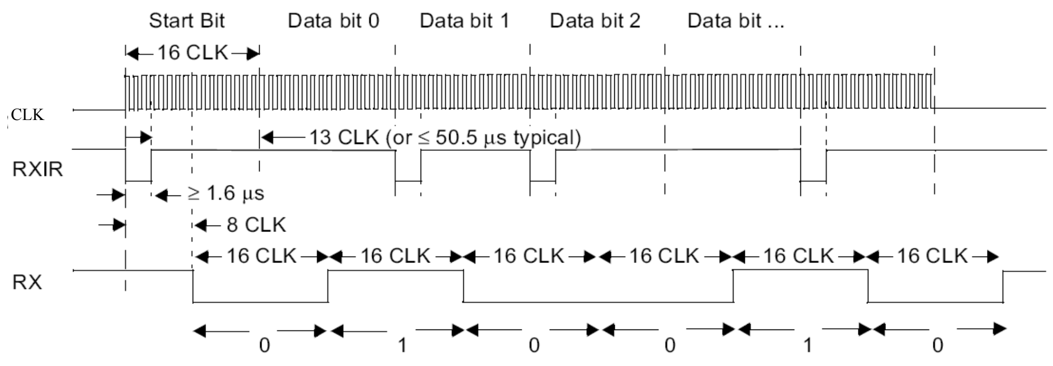
And IR frame demodulation (decoding) scheme is shown in Figure 20. RXIR line voltage level is default low, active high.

Figure 20. IR frame demodulation scheme 1 (active high)



In application, the RXIR frame output from IrDA transceiver is often pull-up, default high, active low. Figure 21 shows another demodulation scheme.

Figure 21. IR frame demodulation scheme 2 (active low)



Software Buffer

The SCB has a FIFO memory, which is a 16 word by 16 bit SRAM, with byte write enable. In UART mode, the FIFO is split into TX FIFO and RX FIFO. Each has eight entries of 16 bits per entry. The 16-bit width per entry is used to accommodate configurable data width.

The internal interrupt handler is used to provide interaction between software and hardware TX/RX buffers without any changes to your top-level firmware.

You should also consider that using the software buffer leads to greater timing intervals between transmitted words because of the extra time the interrupt handler needs to execute (depending on the selected HFCLK value).

Interrupts

When **RX buffer size** or **TX buffer size** is greater than 8 bytes/words, the **RX FIFO not empty** and **TX FIFO not full** interrupt sources are reserved by the component for internal software buffer operations. **Do not clear or disable them** because it causes incorrect software buffer operation. But it is user responsibility to clear interrupt from other sources because they are not cleared automatically. The customer handler function exists to do that.

In case **RX buffer size** or **TX buffer size** is less or equal 8 bytes/words 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** interrupt mode is preferred in this case.

Low power modes

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

The Sleep mode is identical to Active from peripheral point of view. It implies that it is not required any configuration changes in component or code to be called before enter/exit this mode. Any UART activity in TX or RX direction which involves interrupt to occur leads to wake up.

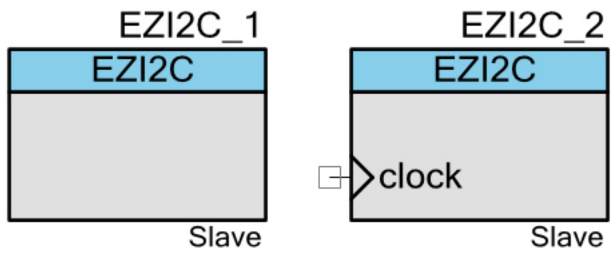
The Deep Sleep mode requires that UART has to be properly configured to be wakeup source. The “Enable wakeup from Sleep mode” must be checked in the slave configuration dialog and RX direction enabled. The SCB_Sleep() and SCB_Wakeup() has to be called before/after enter/exit Deep Sleep.

The device wakes up by the RX GPIO falling edge event that is generated by incoming start bit. There are two constrains for wakeup:

- the 1st data bit of wakeup transaction has to be ‘1’. The UART skips start bit and synchronizes on the 1st data bit.
- the wakeup time of device has to be less than one bit duration. In other case the received data will be incorrect.

Note RX GPIO interrupt restricts usage of all GPIO interrupts from the port where UART rx pin is placed.

EZI2C



The EZI2C Slave mode implements an I^2C register-based slave device. It is compatible with I^2C Standard-mode, Fast-mode, and Fast-mode Plus devices as defined in the NXP I2C-bus specification. 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. The EZ I^2C Slave supports standard data rates up to 1000 kbps and is compatible with multiple devices on the same bus.

The EZ I^2C Slave is a unique implementation of an I^2C 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 executed, there is little need to interact with the API.

Input/Output Connections

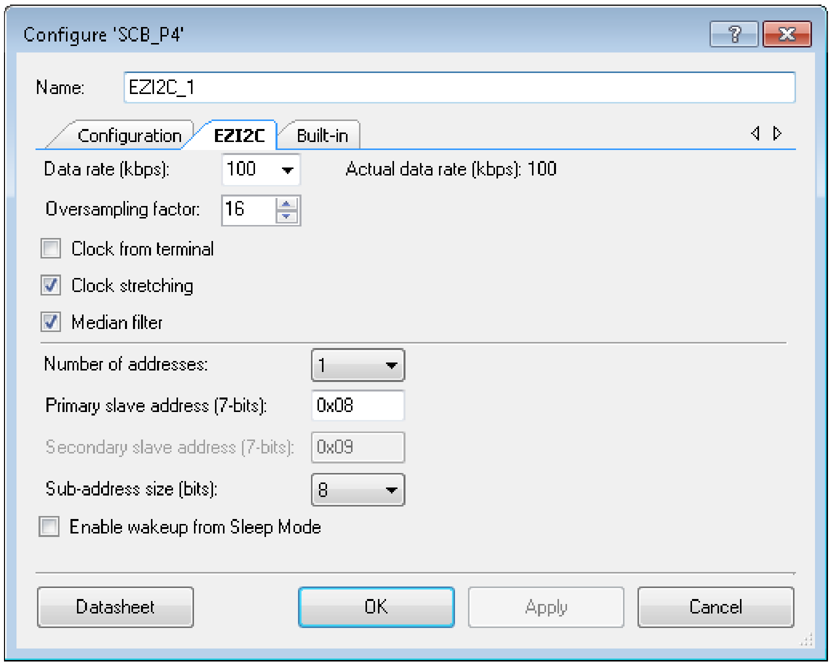
This section describes the various input and output connections for the SCB component. An asterisk (*) in the list of I/Os indicates that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.

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 chip *Technical Reference Manual (TRM)* for more information.

EZI2C Parameters


Configure 'SCB_P4'

Name: EZI2C_1

Data rate (kbps): 100
Actual data rate (kbps): 100
Oversampling factor: 16
☐ Clock from terminal
☑ Clock stretching
☑ Median filter
Number of addresses: 1
Primary slave address (7-bits): 0x08
Secondary slave address (7-bits): 0x09
Sub-address size (bits): 8
☐ Enable wakeup from Sleep Mode

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

Data rate

This parameter is used to set the I^{2}C data rate value up to 1000 kbps (400 kbps for PSoC 4000 family); the actual speed may differ based on available clock speed and divider range. The standard data rates are 50, 100 (default), 400, and 1000 kbps. The **Data Rate** is limited to a maximum of 400 kHz if the **Clock Stretching** option is disabled. If **Clock from terminal** is set, the **Data Rate** parameter is ignored; the input clock and **Oversampling factor** determines the actual data rate.

Actual data rate

Actual data rate displays the data rate which 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: oversampling factor, HFCLK clock and accuracy of the internal or external component clock.

Oversampling factor

This parameter defines the oversampling factor of the I^{2}C SCL clock; the number of component clocks within one I^{2}C SCL clock period. **Oversampling factor** is used to calculate the internal component clock frequency required to achieve this amount of oversampling for defined **Data rate**. An oversampling factor maximum value is 32 and minimum value depends on **Median filter** settings. Median filter is unchecked – 12, Median filter is checked – 14. The default is 16.

Clock from terminal

This parameter allows a choice between an internally configured clock and an externally configured clock for data rate generation. When the option is enabled, the component does not control the data rate but displays the actual data rate based on the user-connected clock source and component oversampling factor. When the option is not enabled, PSoC Creator configures the required clock source. The clock source frequency is calculated by the component based on the **Data rate** parameter and oversampling factor.

Note When setting the data rate or external clock frequency value, make sure that PSoC Creator can provide this value using the current system clock frequency. Otherwise, a warning about the clock accuracy range is generated while building the project. This warning contains the actual clock value set by PSoC Creator. Choose whether the system clock or component clock should be changed to fit the clocking system requirements and achieve an optimal value.

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 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.

Median filter

This parameter applies a digital 3 tap median filter on the input path of the I2C SDA signal. This filter reduces the susceptibility to errors. However, the minimum oversampling factor value is increased.

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 0 and 127 may be selected; the default is **8**.

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^{2}C 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 0 and 127 may be selected; the default is **9**. 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^{2}C master to access data arrays of up to 65,535 bytes.

Enable wakeup from Sleep Mode

This option allows the system to be awakened from sleep when a slave address match occurs.

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

- clock stretching must be enabled
- median filter must be disabled
- slave address must be even (bit 0 equal zero)

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

EZI2C APIs

Application Programming Interface (API) routines 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_Init()	Initialize the SCB component according to defined parameters in the customizer.
SCB_Enable()	Enables the SCB component operation.

Function	Description
SCB_Start()	Starts the SCB component.
SCB_Stop()	Disable the SCB component.
SCB_Sleep()	Prepares the SCB component to enter Deep Sleep.
SCB_Wakeup()	Prepares the SCB component to exit Deep Sleep.
SCB_EzI2CInit()	Configures the SCB component for operation in EZ I^2C 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.
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.

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.

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. 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_Start(void)

Description:	Invokes SCB_Init() and SCB_Enable(). After this function call the component is enabled and ready for 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_Stop(void)

Description:	Disables the SCB component and its interrupt.
Parameters:	None
Return Value:	None
Side Effects:	None

void SCB_Sleep(void)

Description:	Prepares component to enter Deep Sleep. The “Enable wakeup from Sleep Mode” selection has an influence on this function implementation. 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 exit Deep Sleep. The “Enable wakeup from Sleep Mode” selection influences this function implementation. **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 I^{2}C 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 I^{2}C 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	0 – disable 1 – enable
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.

Return Value: None

Side Effects: None

uint32 SCB_EzI2CGetActivity(void)

Description: Returns EZ I^2C 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 I^2C slave completes.

This function disables EZ I^2C slave interrupt while execution to operate correctly. This may have significant impact to correctness of EZ I^2C 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 I^2C slave interrupt latency. Refer to section Clock Stretching Disable for more details.

Parameters: None

Return Value: uint32: Current status of EZ I^2C 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. When SCB_EZI2C_STATUS_ERR is set the slave buffer may contain invalid byte. It is recommended to discard buffer content in this case.

Side Effects: None

void SCB_EzI2CSetAddress1(uint32 address)

Description:	Sets the primary EZ I^{2}C slave address.
Parameters:	uint32 address: primary I^{2}C 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 whether it violates the I^{2}C 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^{2}C 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^{2}C 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^{2}C 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^{2}C 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^{2}C slave address.
Parameters:	uint32 address: secondary I^{2}C 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 whether 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^{2}C 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^{2}C 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 I2C 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 I2C 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.

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

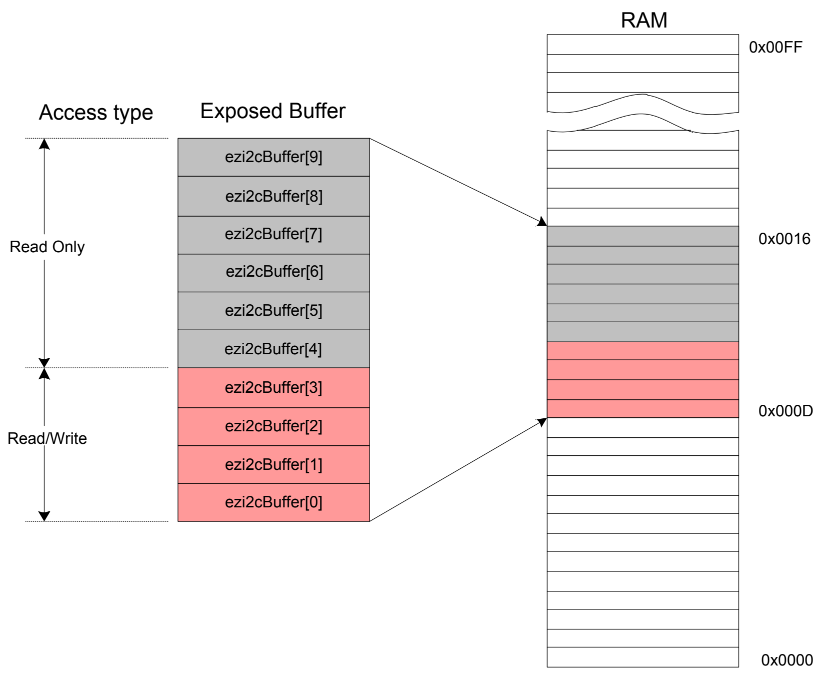
To an I^2C master, the interface looks very similar to a common I^2C EEPROM. The EZ I^2C buffer can be configured as a variable, array, or structure but it is preferable to use an array. The buffer acts as a shared memory interface between your program and an I^2C master through the I^2C bus. The component permits read and write I^2C master access to the specified buffer memory and prevents any access outside the buffer or write access into a read only region.

For example: if 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 22.

Figure 22. EZ I^2C buffer exposed to an I^2C master



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

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

Handling endianness

The EZ I^2C 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 I^2C bus. It is the I^2C 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 I^2C 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 I^2C buffer may be required. This status byte would be modified by the I^2C master but the compiler is not aware that an interrupt routine may modify this buffer. That 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:

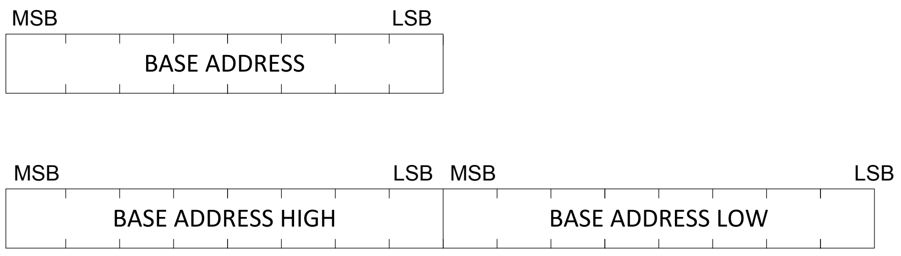
```
#define BUFFER_SIZE       (0x0Au)
#define STATUS_BYTE_POS   (0u)

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 I^{2}C slave component supports basic read and write operations for the read/write region and read operations for the read-only region. The two I^{2}C address interfaces contain separate data buffers that are addressed with separate base addresses. The base address is an index within the EZ I^{2}C 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 16bit value Figure 23.

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

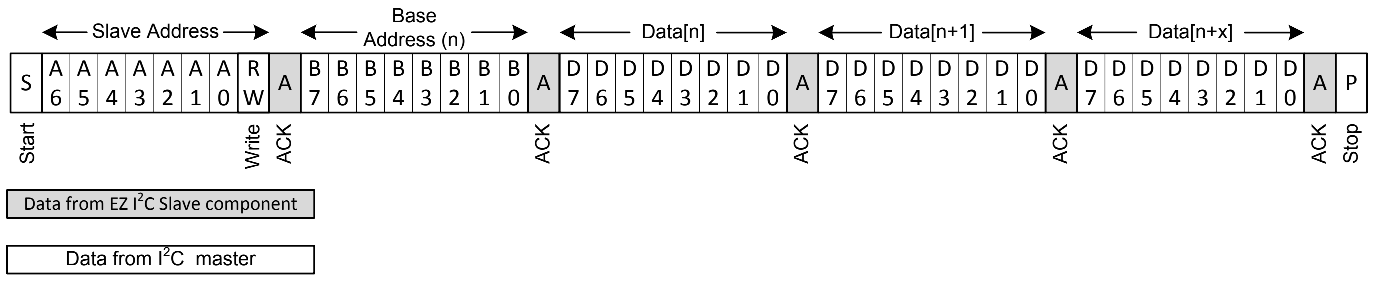
Figure 23. The 8-bit and 16-bit Sub-Address Size



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:

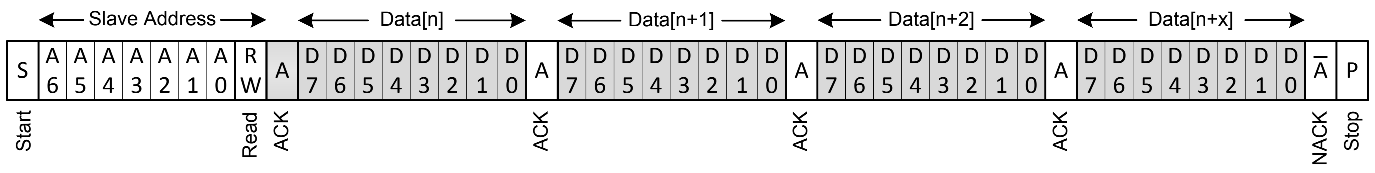
- Enabled: the byte is NAKed by the slave and the master has to stop the current transaction. The NAKed byte is discarded by the slave.
- Disabled: all written bytes are ACKed by the slave, but these bytes are discarded.

Figure 24. I^{2}C Master writes X bytes to the EZ I^{2}C Slave buffer


Slave Address
Base Address (n)
Data[n]
Data[n+1]
Data[n+x]
S A6 A5 A4 A3 A2 A1 A0 RW A B7 B6 B5 B4 B3 B2 B1 B0 A D7 D6 D5 D4 D3 D2 D1 D0 A D7 D6 D5 D4 D3 D2 D1 D0 A D7 D6 D5 D4 D3 D2 D1 D0 A P
Start
Write
ACK
ACK
ACK
ACK
ACK
Stop
Data from EZ I2C Slave component
Data from I2C master

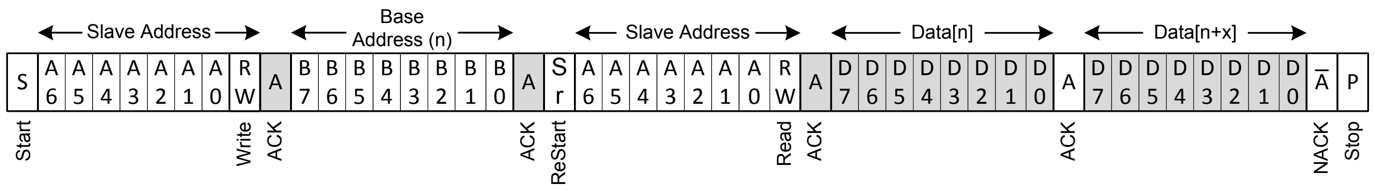
A read operation **always starts** from the base address set by the most recent write operation. The buffer index is incremented for each read byte. Two sequential read operations start from the same base address no matter how many bytes were read. The length of a read operation is **not limited** by 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 25. I^{2}C Master reads X bytes from the EZ I^{2}C Slave buffer


Slave Address
Data[n]
Data[n+1]
Data[n+2]
Data[n+x]
S A6 A5 A4 A3 A2 A1 A0 RW A D7 D6 D5 D4 D3 D2 D1 D0 A D7 D6 D5 D4 D3 D2 D1 D0 A D7 D6 D5 D4 D3 D2 D1 D0 A D7 D6 D5 D4 D3 D2 D1 D0 Ā P
Start
Read
ACK
ACK
ACK
ACK
NACK
Stop

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 I^{2}C 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 26. I^{2}C Master sets the base address and reads X bytes from the EZ I^{2}C Slave buffer


Slave Address
Base Address (n)
Slave Address
Data[n]
Data[n+x]
S A6 A5 A4 A3 A2 A1 A0 RW A B7 B6 B5 B4 B3 B2 B1 B0 A Sr A6 A5 A4 A3 A2 A1 A0 RW A D7 D6 D5 D4 D3 D2 D1 D0 A D7 D6 D5 D4 D3 D2 D1 D0 Ā P
Start
Write
ACK
ACK
ReStart
Read
ACK
ACK
NACK
Stop

Detailed descriptions of the I^{2}C bus and its implementation are available in the complete I^{2}C 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 set up 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^{2}C spec. For that reason the EZ I^{2}C 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^{2}C slave operation for any slave interrupt latency. The drawback is that the master has to support clock stretching as well.

Clock Stretching Disable

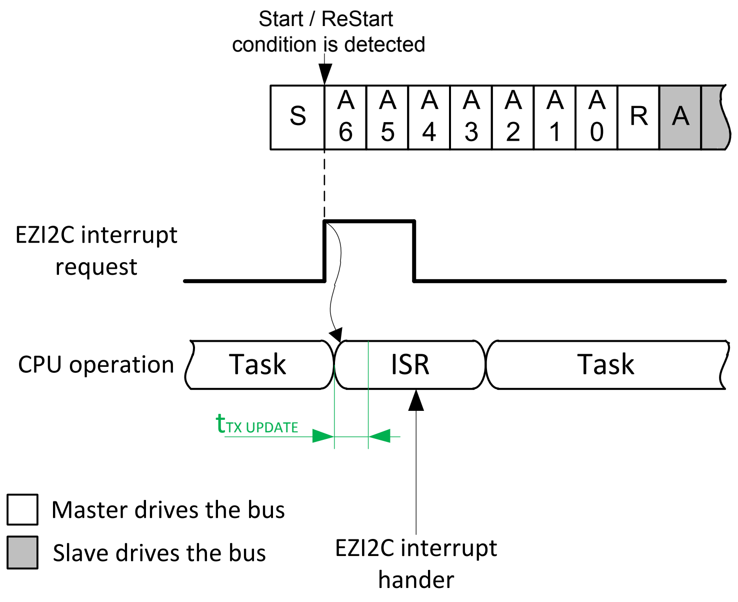
Disabling clock stretching configures the EZ I^{2}C slave to operate with an optimized interrupt service routine. This allows the EZ I^{2}C slave to operate without clock stretching. Despite the optimization, the slave interrupt still must be serviced frequently enough. The maximum time that the slave interrupt service can be delayed is defined as the maximum EZ I^{2}C slave interrupt latency. A design that does not satisfy the required maximum EZ I^{2}C 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^{2}C 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 generation. The slave hardware detects this condition and generates an interrupt request, which starts slave operation (Figure 27). The ReStart condition generation has the same effect on the slave as Start condition, except previous transaction completion flags have to be set. But service of the ReStart condition has greater priority.

Figure 27. EZ I^{2}C 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 27). The TX FIFO update consists of the TX FIFO clear and writes the byte into the TX FIFO from the slave buffer. 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 the 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 28).

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.

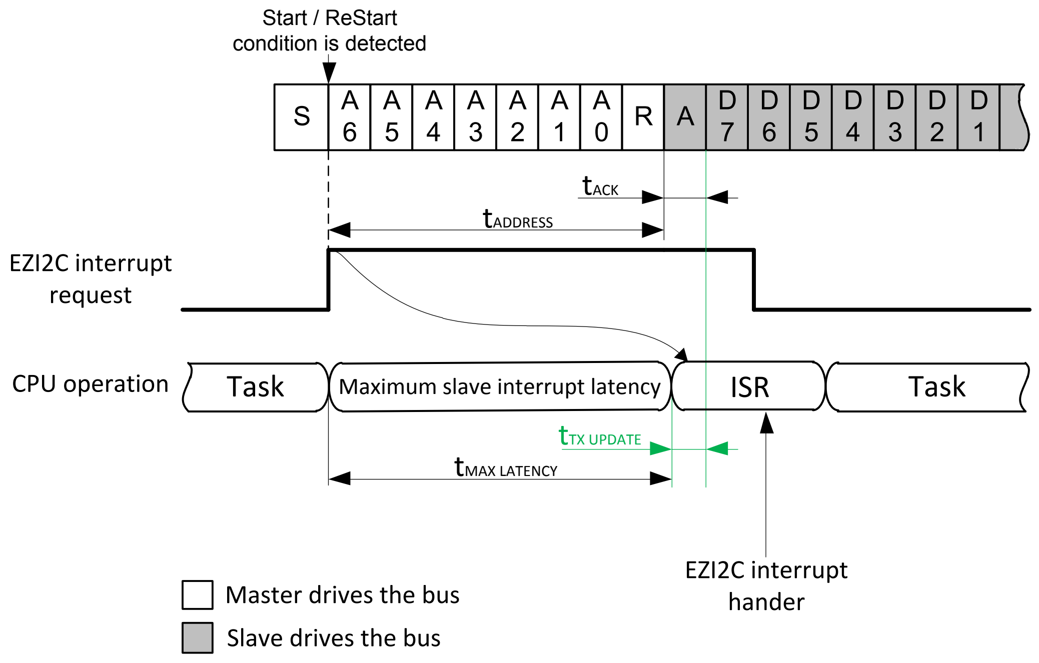
Therefore, the maximum interrupt latency must be less than the master address byte transmit time ($t_{ADDRESS}$) plus slave ACK bit transmit time (t_{ACK}). But taking to account TX FIFO update constraint, the maximum interrupt latency must be less than:

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

For example I²C data rate of 100 kpbs, the maximum interrupt latency must be less than:

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

Figure 28. 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 I²C slave interrupt.
2. Use optimization options of compiler as it reduces number of instructions to execute in the EZ I²C slave interrupt.
3. Set EZ I²C 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 I²C maximum interrupt latency.
4. Calculate duration of the each critical section in the design and compare with EZ I²C maximum interrupt latency to make sure that design meets criteria.

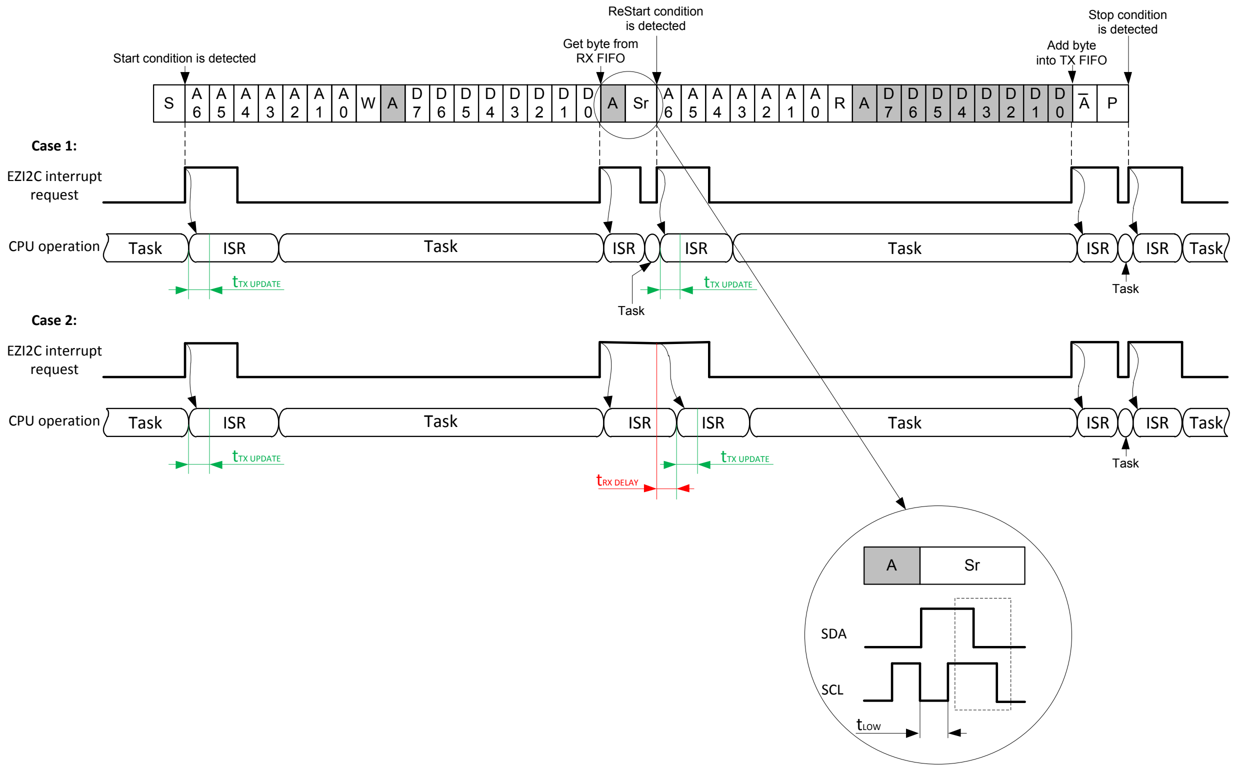
Transactions chained with ReStart

The master write base address and read data transaction are chained together with ReStart (Figure 29). 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 29, case 1). The service of the RX FIFO has greater priority than the ReStart condition service because the base address might be 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 29, case 2), the maximum interrupt latency is reduced by the amount which adds the RX FIFO service after the ReStart condition is generated:

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

The master ReStart timings have to be examined because the I^{2}C spec provides minimum values. Some masters before ReStart generation extend t_{LOW} due to preparation 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 29. Master set base address and read data


Start condition is detected
Get byte from RX FIFO
ReStart condition is detected
Add byte into TX FIFO
Stop condition is detected
S A6 A5 A4 A3 A2 A1 A0 W A D7 D6 D5 D4 D3 D2 D1 D0 A Sr A6 A5 A4 A3 A2 A1 A0 R A D7 D6 D5 D4 D3 D2 D1 D0 Ā P
Case 1:
EZI2C interrupt request
CPU operation
Task
ISR
$t_{TX\ UPDATE}$
Case 2:
$t_{RX\ DELAY}$
A
Sr
SDA
SCL
t_{LOW}

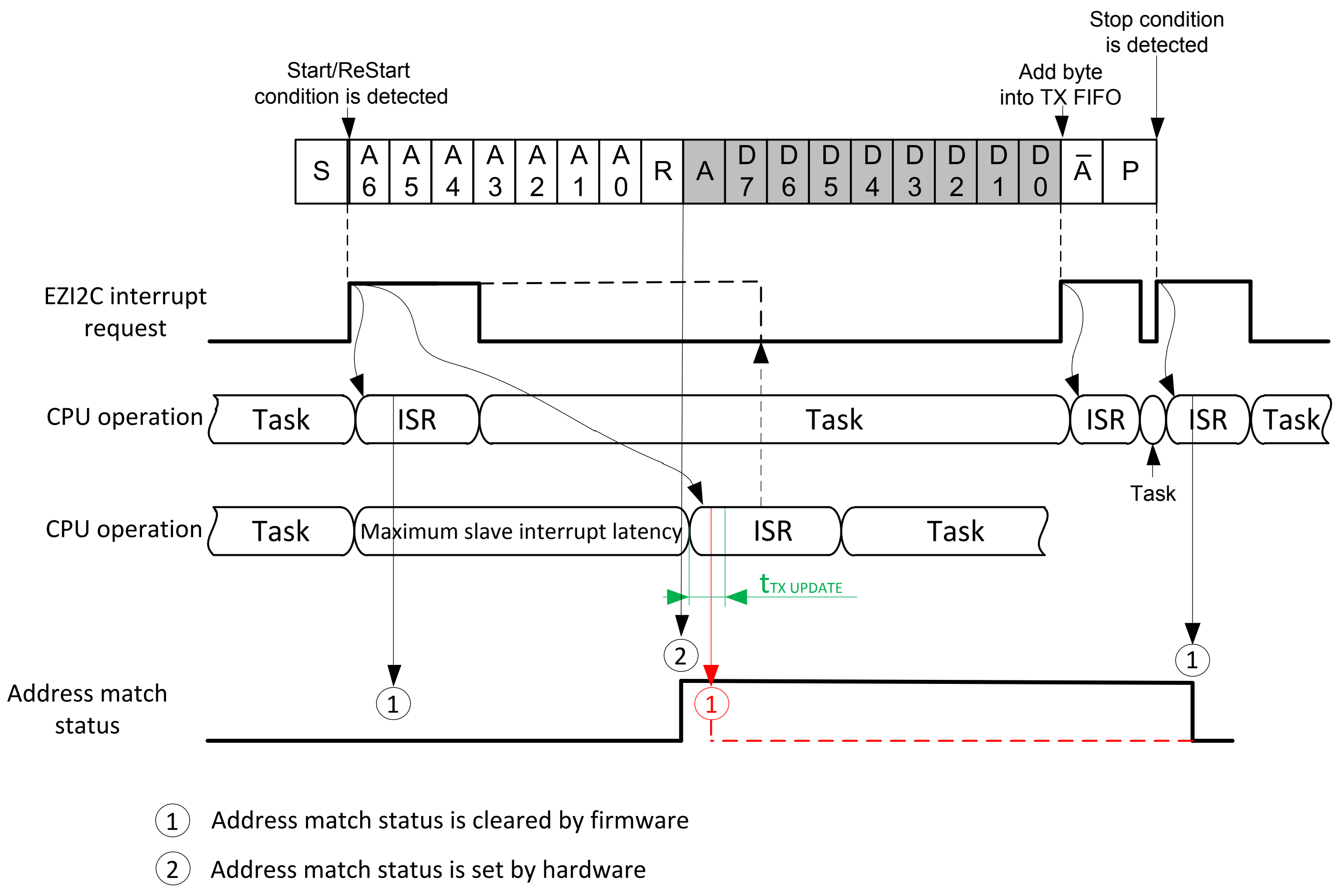
Slave busy management

The SCB_EzI2CGetActivity() and SCB_Sleep() (only for PSoC 4000 devices) APIs use address match status to track slave busy status. This event triggers 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 30, black cycle 2).

To be used as slave busy status, the address match is cleared by firmware on the Start / ReStart or Stop condition service (Figure 30, black cycle 1). If Start / ReStart interrupt service is delayed for maximum interrupt latency, the address match status is cleared too early (Figure 30, red cycle 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 30. Slave busy management



External Electrical Connections

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

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 NACK.

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 NACK all others 0x20 and 0x34.

Low power modes

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

The Sleep mode is identical to Active from peripheral point of view. It is not required any configuration changes in component or code to be called before enter/exit this mode. Any commination intended to the slave causes interrupt to occur and leads to wake up.

The Deep Sleep mode requires that slave has to be properly configured to be wakeup source. The "Enable wakeup from Sleep mode" must be checked in the slave configuration dialog. The SCB_Sleep() and SCB_Wakeup() has to be called before/after enter/exit Deep Sleep.

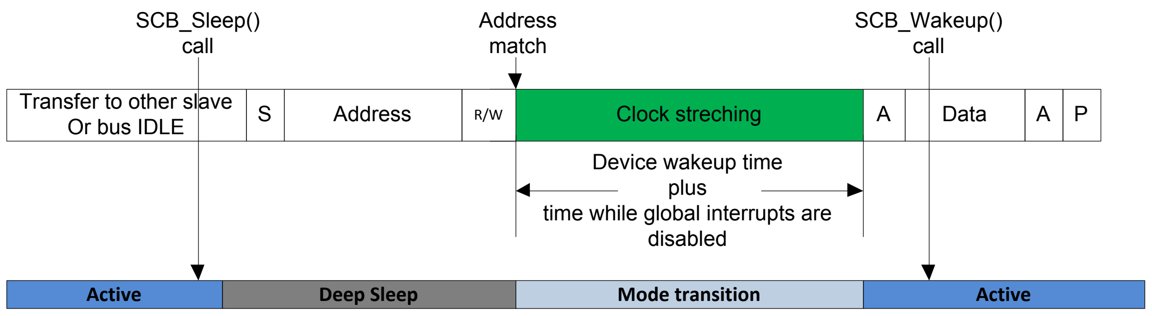
The wakeup event is slave address match. The externally clocked logic performs address matching and when it was occurred the wakeup interrupt triggers. But the slave behavior after address match depends on clock stretching option selection.

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

Before enter Deep Sleep the on-going transaction intended to the slave has to be completed therefore following code is suggested:

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

/* Check if a transaction is in process */
status = (SCB_EzI2CGetActivity() & SCB_EZI2C_STATUS_BUSY);

if(0u == status) /* Slave is not addressed */
{
    SCB_Sleep(); /* Prepare for Deep Sleep: enables wakeup interrupt */

    CySysPmDeepSleep();

    CyGlobalIntEnable; /* Enable all interrupts to unlock I2C bus state */

    SCB_Wakeup(); /* Restore for Active mode: disables wakeup interrupt */
}
else
{
    /* Transaction in progress: do not go to Deep Sleep */
    CyGlobalIntEnable;
}
```

Figure 31. Master transfer wakes up device on slave address match (Clock stretching enable)



Note The values for the primary and secondary addresses effect 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 NACKed in this case.

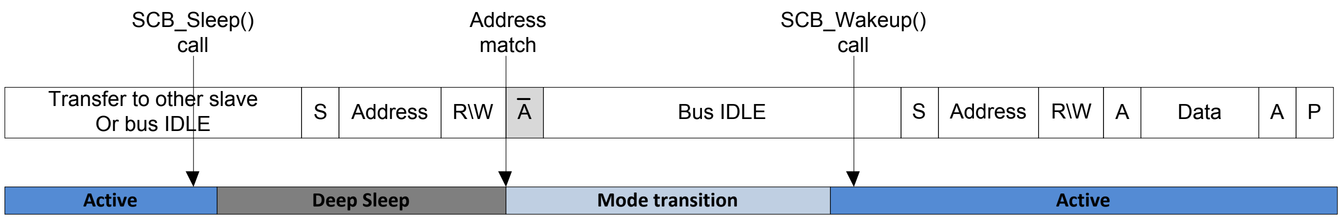
Clock stretching disable: the slave NACKs matched address and following matched addresses while device wakeup time.

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 will be free to configure it to be wakeup source. 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 has to be restored by SCB_Wakeup(). The agreement between slave and master has to be concluded to not access 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 32. Master transfer wakes up device on slave address match (Clock stretching disable)



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 what the source of the current triggered interrupt.
SCB_SetCustomInterruptHandler()	Registers a function to be called by the internal interrupt handler.
SCB_SetTxInterruptMode()	Configures which bits of TX interrupt request register will trigger an interrupt event.
SCB_GetTxInterruptMode()	Returns TX interrupt mask
SCB_GetTxInterruptSourceMasked()	Returns TX interrupt request register masked by interrupt mask
SCB_GetTxInterruptSource()	Returns the bit-mask of pending TX interrupt sources
SCB_ClearTxInterruptSource()	Clears the bit-mask of pending TX interrupt sources
SCB_SetTxInterrupt()	Generates interrupt event from bit-mask of TX interrupt sources
SCB_SetRxInterruptMode()	Configures which bits of RX interrupt request register will trigger an interrupt event
SCB_GetRxInterruptMode()	Returns RX interrupt mask
SCB_GetRxInterruptSourceMasked()	Returns RX interrupt request register masked by interrupt mask
SCB_GetRxInterruptSource()	Returns the bit-mask of pending RX interrupt sources
SCB_ClearRxInterruptSource()	Clears the bit-mask of pending RX interrupt sources
SCB_SetRxInterrupt()	Generates interrupt event from bit-mask of RX interrupt sources
SCB_SetMasterInterruptMode()	Configures which bits of Master interrupt request register will trigger an interrupt event
SCB_GetMasterInterruptMode()	Returns Master interrupt mask
SCB_GetMasterInterruptSourceMasked()	Returns Master interrupt request register masked by interrupt mask
SCB_GetMasterInterruptSource()	Returns the bit-mask of pending Master interrupt sources

Function	Description
SCB_ClearMasterInterruptSource()	Clears the bit-mask of pending Master interrupt sources
SCB_SetMasterInterrupt()	Generates interrupt event from bit-mask of Master interrupt sources
SCB_SetSlaveInterruptMode()	Configures which bits of Slave interrupt request register will trigger an interrupt event
SCB_GetSlaveInterruptMode()	Returns Slave interrupt mask
SCB_GetSlaveInterruptSourceMasked()	Returns Slave interrupt request register masked by interrupt mask
SCB_GetSlaveInterruptSource()	Returns the bit-mask of pending Slave interrupt sources
SCB_ClearSlaveInterruptSource()	Clears the bit-mask of pending Slave interrupt sources
SCB_SetSlaveInterrupt()	Generates interrupt event from bit-mask of Slave interrupt sources

Interrupt Function Appliance

Function	I2C	SPI	UART	EZI2C
SCB_EnableInt()	+	+	+	+
SCB_DisableInt()	+	+	+	+
SCB_GetInterruptCause()	+	+	+	+
SCB_SetCustomInterruptHandler()	+	+	+	+
SCB_SetTxInterruptMode()	+	+	+	+
SCB_GetTxInterruptMode()	+	+	+	+
SCB_GetTxInterruptSourceMasked()	+	+	+	+
SCB_GetTxInterruptSource()	+	+	+	+
SCB_ClearTxInterruptSource()	+	+	+	+
SCB_SetTxInterrupt()	+	+	+	+
SCB_SetRxInterruptMode()	+	+	+	+
SCB_GetRxInterruptMode()	+	+	+	+
SCB_GetRxInterruptSourceMasked()	+	+	+	+
SCB_GetRxInterruptSource()	+	+	+	+
SCB_ClearRxInterruptSource()	+	+	+	+
SCB_SetRxInterrupt()	+	+	+	+
SCB_SetMasterInterruptMode()	+	+	–	–
SCB_GetMasterInterruptMode()	+	+	–	–
SCB_GetMasterInterruptSourceMasked()	+	+	–	–

Function	I2C	SPI	UART	EZI2C
SCB_GetMasterInterruptSource()	+	+	–	–
SCB_ClearMasterInterruptSource()	+	+	–	–
SCB_SetMasterInterrupt()	+	+	–	–
SCB_SetSlaveInterruptMode()	+	+	–	+
SCB_GetSlaveInterruptMode()	+	+	–	+
SCB_GetSlaveInterruptSourceMasked()	+	+	–	+
SCB_GetSlaveInterruptSource()	+	+	–	+
SCB_ClearSlaveInterruptSource()	+	+	–	+
SCB_SetSlaveInterrupt()	+	+	–	+

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 what 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 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, then the internal interrupt handler performs any operations such as software buffer management functions before the interrupt returns. 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

void SCB_SetTxInterruptMode(uint32 interruptMask)

Description: Configures which bits of TX interrupt request register will trigger an interrupt event.

Parameters: uint32 interruptMask: bit-mask of TX interrupt sources to be enabled.

TX interrupt sources	Description
SCB_INTR_TX_TRIGGER	Transmitter FIFO has fewer entries than the value specified by trigger 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 and the TX FIFO is empty.
SCB_INTR_TX_UART_ARB_LOST	Value on the TX line of the UART does not match the value on the RX line.

Return Value: None

Side Effects: None

uint32 SCB_GetTxInterruptMode(void)

Description: Returns TX interrupt mask.

Parameters: None

Return Value: uint32: Mask of enabled TX interrupt sources (refer to SCB_SetTxInterruptMode() function for return values).

Side Effects: None

uint32 SCB_GetTxInterruptSourceMasked(void)

Description: Returns TX interrupt request register masked by interrupt mask.

Parameters: None

Return Value: uint32: Status only of enabled TX interrupt sources (refer to SCB_SetTxInterruptMode() function for return values).

Side Effects: None

uint32 SCB_GetTxInterruptSource(void)

Description: Returns the bit-mask of pending TX interrupt sources.

Parameters: None

Return Value: uint32: Status of TX interrupt sources (refer to SCB_SetTxInterruptMode() function for return values).

Side Effects: None

void SCB_ClearTxInterruptSource(uint32 interruptMask)

Description: Clears the bit-mask of pending TX interrupt sources.

Parameters: uint32 interruptMask: Bit-mask of pending TX interrupt sources to clear (refer to SCB_SetTxInterruptMode() function for return values).

Return Value: None

Side Effects: The side effects are listed in the table below for each affected interrupt source.

TX interrupt sources	Description
SCB_INTR_TX_TRIGGER	Interrupt source is not cleared when trigger event is valid.
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 clear it. This behavior only observed for PSoC 4100/PSoC 4200 devices.

void SCB_SetTxInterrupt(uint32 interruptMask)

Description: Generates interrupt event from bit-mask of TX interrupt sources.

Parameters: uint32 interruptMask: Bit-mask of TX interrupt sources to generate an interrupt event (refer to SCB_SetTxInterruptMode() function for return values).

Return Value: None

Side Effects: None

void SCB_SetRxInterruptMode(uint32 interruptMask)

Description: Configures which bits of RX interrupt request register will trigger an interrupt event.

Parameters: uint32 interruptMask: Bit-mask of RX interrupt sources to be enabled.

RX interrupt sources	Description
SCB_INTR_RX_TRIGGER	Receiver FIFO has more entries than the value specified by trigger 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.

Return Value: None

Side Effects: None

uint32 SCB_GetRxInterruptMode(void)

Description: Returns RX interrupt mask.

Parameters: None

Return Value: uint32: Mask of enabled RX interrupt sources (refer to SCB_SetRxInterruptMode() function for return values).

Side Effects: None

uint32 SCB_GetRxInterruptSourceMasked(void)

Description: Returns RX interrupt request register masked by interrupt mask.

Parameters: None

Return Value: uint32: Status only of enabled RX interrupt sources (refer to SCB_SetRxInterruptMode() function for return values).

Side Effects: None

uint32 SCB_GetRxInterruptSource(void)

Description: Returns the bit-mask of pending RX interrupt sources.

Parameters: None

Return Value: uint32: Status of RX interrupt sources (refer to SCB_SetRxInterruptMode() function for return values).

Side Effects: None

void SCB_ClearRxInterruptSource(uint32 interruptMask)

Description: Clears the bit-mask of pending RX interrupt sources.

Parameters: uint32 interruptMask: Bit-mask of pending RX interrupt sources to clear (refer to SCB_SetRxInterruptMode() function for return values)

Return Value: None

Side Effects: The side effects are listed in the table below for each affected interrupt source.

RX interrupt sources	Description
SCB_INTR_RX_TRIGGER	Interrupt source is not cleared when trigger event is still valid.
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: Generates interrupt event from bit-mask of RX interrupt sources.

Parameters: uint32 interruptMask: Bit-mask of RX interrupt sources to generate an interrupt event (refer to SCB_SetRxInterruptMode() function for return values).

Return Value: None

Side Effects: None

void SCB_SetMasterInterruptMode(uint32 interruptMask)

Description: Configures which bits of Master interrupt request register will trigger an interrupt event.

Parameters: uint32 interruptMask: Bit-mask of RX interrupt sources to be enabled.

Master interrupt sources	Description
SCB_INTR_MASTER_SPI_DONE	SPI Master transfer is complete and the TX FIFO is empty.
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).

Return Value: None

Side Effects: None

uint32 SCB_GetMasterInterruptMode(void)

Description: Returns Master interrupt mask

Parameters: None

Return Value: uint32: Mask of enabled Master interrupt sources (refer to SCB_SetMasterInterruptMode() function for return values).

Side Effects: None

uint32 SCB_GetMasterInterruptSourceMasked(void)

Description: Returns Master interrupt request register masked by interrupt mask.

Parameters: None

Return Value: uint32: Status only of enabled Master interrupt sources (refer to SCB_SetMasterInterruptMode() function for return values).

Side Effects: None

uint32 SCB_GetMasterInterruptSource(void)

Description:	Returns the bit-mask of pending Master interrupt sources.
Parameters:	None
Return Value:	uint32: Status of Master interrupt sources (refer to SCB_SetMasterInterruptMode() function for return values).
Side Effects:	None

void SCB_ClearMasterInterruptSource(uint32 interruptMask)

Description:	Clears the bit-mask of pending Master interrupt sources.
Parameters:	uint32 interruptMask: Bit-mask of pending Master interrupt sources to clear (refer to SCB_SetMasterInterruptMode() function for return values).
Return Value:	None
Side Effects:	None

void SCB_SetMasterInterrupt(uint32 interruptMask)

Description:	Generates interrupt event from bit-mask of Master interrupt sources.
Parameters:	uint32 interruptMask: Bit-mask of Master interrupt sources to generate an interrupt event (refer to SCB_SetMasterInterruptMode() function for return values).
Return Value:	None
Side Effects:	None

void SCB_SetSlaveInterruptMode(uint32 interruptMask)

Description: Configures which bits of Slave interrupt request register will trigger an interrupt event.

Parameters: uint32 interruptMask: Bit-mask of Slave interrupt sources to be enabled.

Slave interrupt sources	Description
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.
INTR_SLAVE_I2C_NACK	I2C slave received negative acknowledgement (NAK).
INTR_SLAVE_I2C_ACK	I2C slave received acknowledgement (ACK).
INTR_SLAVE_I2C_WRITE_STOP	Stop or Repeated Start event for write transfer intended for this slave (address matching is performed).
INTR_SLAVE_I2C_STOP	Stop or Repeated Start event for (read or write) transfer intended for this slave (address matching is performed).
INTR_SLAVE_I2C_START	I2C slave received Start condition.
INTR_SLAVE_I2C_ADDR_MATCH	I2C slave received matching address.
INTR_SLAVE_I2C_GENERAL	I2C Slave received general call address.
INTR_SLAVE_I2C_BUS_ERROR	I2C slave bus error (detection of unexpected START or STOP condition).
INTR_SLAVE_SPI_BUS_ERROR	SPI slave deselected at an expected time in the SPI transfer.

Return Value: None

Side Effects: None

uint32 SCB_GetSlaveInterruptMode(void)

Description: Returns Slave interrupt mask.

Parameters: None

Return Value: uint32: Mask of enabled Slave interrupt sources (refer to SCB_SetSlaveInterruptMode() function for return values).

Side Effects: None

uint32 SCB_GetSlaveInterruptSourceMasked(void)

Description:	Slave interrupt request register masked by interrupt mask.
Parameters:	None
Return Value:	uint32: Status only of enabled Slave interrupt sources (refer to SCB_SetSlaveInterruptMode() function for return values).
Side Effects:	None

uint32 SCB_GetSlaveInterruptSource(void)

Description:	Returns the bit-mask of pending Slave interrupt sources
Parameters:	None
Return Value:	uint32: Status of Slave interrupt sources (refer to SCB_SetSlaveInterruptMode() function for return values)
Side Effects:	None

void SCB_ClearSlaveInterruptSource(uint32 interruptMask)

Description:	Clears the bit-mask of pending Slave interrupt sources.
Parameters:	uint32 interruptMask: Bit-mask of pending Slave interrupt sources to clear (refer to SCB_SetSlaveInterruptMode() function for return values).
Return Value:	None
Side Effects:	None

void SCB_SetSlaveInterrupt(uint32 interruptMask)

Description:	Generates interrupt event from bit-mask of Slave interrupt sources.
Parameters:	uint32 interruptMask: Bit-mask of Slave interrupt sources to generate an interrupt event (refer to SCB_SetSlaveInterruptMode() function for return values).
Return Value:	None
Side Effects:	None

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[8]	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.

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.

8 Required / Advisory

Interrupt Service Routine

The SCB supports interrupts on the various events, depends on mode which is working on. 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 there 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 interrupt sources have specific behavior: TX FIFO empty, TX FIFO not full, and TX FIFO trigger. These interrupt sources trigger current status of the TX FIFO and keep it until clear operation. When the SCB component is disabled, TX FIFO becomes empty and TX interrupt sources trigger its state. Clearing these interrupt sources do not make any sense if TX FIFO is empty because they are restored back. The restore operation takes one clock cycle; therefore, the interrupt source is cleared during this time. While filling TX FIFO, it is better to monitor the number of entries in it rather than try to clear TX FIFO not full or trigger interrupt source after each byte put in TX FIFO. The restore time causes false clearing of TX FIFO not full or trigger interrupt source. To start TX FIFO interrupt sources processing, the suggested flow is: fill TX FIFO with data, cleared triggered “old” interrupt source and enable it.

To clear interrupt source, write ‘1’ to corresponding bit position.

Note The TX FIFO trigger interrupt source behavior depends on value of trigger level.

RX FIFO interrupt sources

The following interrupt sources have specific behavior: RX FIFO not empty, RX FIFO full and RX FIFO trigger. These interrupt sources trigger current status of the RX FIFO and keep it until clear operation. When SCB component is disabled RX FIFO becomes empty and triggered interrupt source can be cleared. Clearing these interrupt sources do not make any sense when RX FIFO is full because they are restored back. The restore operation takes one clock cycle; therefore, the interrupt source is cleared while this time. While getting data from RX FIFO it is better to monitor number of entries in it rather than try to clear RX FIFO not empty or trigger interrupt source after each read byte. The restore time causes false clearing of RX FIFO not empty or trigger interrupt source. To start RX FIFO interrupt sources processing the suggested flow is: cleared triggered “old” interrupt source and enable it. In most cases the clear operation is not required.

To clear interrupt source, write ‘1’ to corresponding bit position.

Note The RX FIFO trigger interrupt source behavior depends on value of trigger level.

Placement

The SCB is placed as Fixed Function block and all placement information is provided to the API through the *cyfitter.h* file.

Registers

See the chip *Technical Reference Manual (TRM)* for more information about registers.

Resources

SCB is implemented as a fixed-function block.

Mode		Resource Type		
		SCB Fixed Blocks		Interrupts
Unconfigured SCB			1	1
I^{2}C	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
	SmartCard	Hardware buffers	1	–
		Software buffers	1	1
	IrDA	Hardware buffers	1	–
		Software buffers	1	1
EZ I^{2}C	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	
			Flash	RAM	Flash	RAM
Unconfigured SCB			7006	154	10036	171
I^2C	Slave		1458	42	1530	42
	Master		2504	46	2748	46
	Multi-Master		2504	46	2748	46
	Multi-Master-Slave		3672	79	3904	79
SPI	Slave	Hardware buffers [9]	N/A	N/A	438	9
		Software buffers [10]	N/A	N/A	998	50
	Master	Hardware buffers [9]	N/A	N/A	442	9
		Software buffers [10]	N/A	N/A	1002	50
UART	Standard	Hardware buffers [9]	N/A	N/A	640	9
		Software buffers [10]	N/A	N/A	1152	50
	Standard (Multiprocessor mode)	Hardware buffers [9]	N/A	N/A	652	9
		Software buffers [10]	N/A	N/A	1224	70
	SmartCard	Hardware buffers [9]	N/A	N/A	644	9
		Software buffers [10]	N/A	N/A	1200	50
	IrDA	Hardware buffers [9]	N/A	N/A	648	9
		Software buffers [10]	N/A	N/A	1204	50
EZ I^2C	Clock stretching enabled	One address	1240	26	1284	26
		Two addresses	1636	50	1664	50
	Clock stretching disabled [11]	One address	1216	23	1044	24

9 Hardware buffers – RX and TX buffers size equal 8 bytes, only hardware FIFO is used. Interrupt mode is None.

10 Software buffers – RX and TX buffers size equal 10 bytes, the internal RAM 10 bytes buffers are used as well as hardware FIFO. Internal interrupt is automatically enabled in this case.

11 The "Enable wakeup from Sleep Mode" is enabled for PSoC 4000 devices.

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

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_{I2C4}	I²C enabled in Deep Sleep mode	–	–	2.5	µA	

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}	I²C 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	
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 × FCPU	ns	
T_{DSO_ext}	MISO valid after Sclock driving edge in Ext. Clock mode	–	–	48	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
1.20.a	Fixed EZ I^2C with clock stretching operation when SCB Unconfigured mode is selected.	The compiler generated warnings during compilation.
	Fixed the EZ I^2C mode with clock stretching buffer update when the master writes a multiple of 8 bytes and the base address is also a multiple of 8 bytes.	The EZ I^2C slave completed the transfer too early and the last 8 bytes remained in the RX FIFO. The buffer was not updated properly.
	Fixed the EZ I^2C current address behavior for a buffer size equal to 256 bytes.	When a buffer read or write overflow occurred the address was not handled correctly.
	Improved the interrupt handling timing for EZ I^2C mode without clock stretching.	Provide more margin to allow for operation without clock stretching.
	Fixed the I^2C slave error status reporting.	The reporting of the error status for read and write errors was swapped.
	Added support for PSoC 4000 devices.	
1.10	EZ I^2C mode added.	
	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	