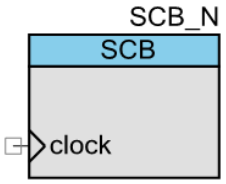
PSoC 4 SCB Component

1.0

Features



- Supports five modes (I^2C, SPI, UART, EZI2C and EZSPI)
- Industry-standard NXP® I^2C bus interface with Slave, Master, Multi-Master and Multi-Master-Slave operation
- Standard I^2C data rates of 100/400/1000 kbps supported
- Standard SPI Master and Slave function with Motorola, Texas Instruments and National Semiconductors (MicroWire) mode
- SPI bit rate is up to 16 Mbit
- Standard UART transmitter and receiver interface with SmartCard reader (ISO7816) and IrDA protocol
- Standard UART baud rates from 110 to 921600 bps
- The EZI2C and EZSPI function mode with up to 32-byte buffer and up to 48 Mbps bitrate
- External clock input support for I^2C and SPI function
- Run-time customization

General Description

The PSoC 4 SCB component is multifunction hardware block that implements I2C, SPI, UART, EZI2C and EZSPI functionality.

When to Use an SCB Component

Overall the SCB can be used in a mode where it is configured using the customizer to be in one of the modes of operation with detailed configuration in that mode: I2C, SPI, UART, EZI2C or EZSPI. 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 configured mode will be the typical use case for any customer that is using Creator as their development environment. It is the simplest method to configure the SCB into the mode of operation that is desired. The unconfigured method will 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 Creator hardware design is created.

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 available in Unconfigured mode. For other modes the option exists to select clock source provided from the clock terminal.

interrupt – Output*

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

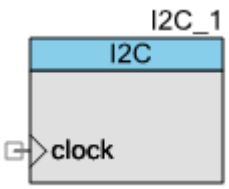
Not present:

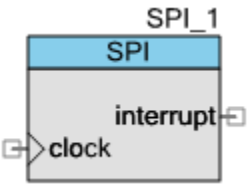
- I2C mode: required for internal operation
- SPI mode: internal buffer (buffer size > 8)
- UART mode: internal buffer (buffer size > 8)
- EZI2C and EZSPI modes: interrupt option is none
- Unconfigured mode: since the interrupt may be needed for I2C operation

The interface specific pins are buried inside component because these pins use dedicated connections and are not routable as general purpose signals.

Schematic Macro Information

By default, the PSoC Creator Component Catalog contains schematic macro implementations for the SCB component. Schematic macros are available for I2C, SPI, UART, EZI2C and EZSPI.

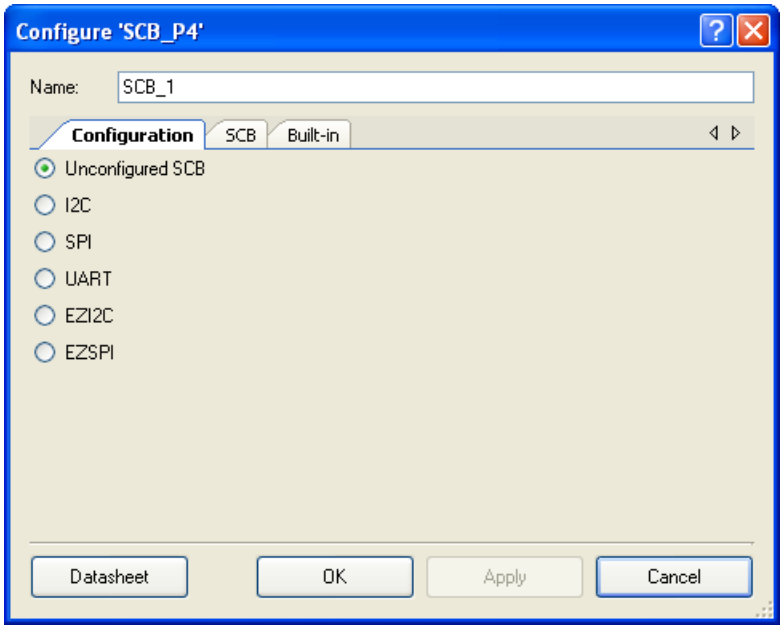




Component Parameters

Drag an SCB component onto your design and double click it to open the **Configure** dialog. Depending on the mode of operation chosen on the Configuration tab, one or two additional tabs will be shown for that specific mode of operation.

Configuration Tab



Unconfigured SCB

This is the default mode. The SCB component does not provide any functions when configured in this mode. The SCB configuration is provided by customer in run time. The SCB Tab becomes available when this mode is selected.

I2C

Component is configured to utilize I2C interface. The I2C Tab becomes available when this mode is selected.

SPI

Component is configured to utilize SPI interface. The SPI Basic Tab and SPI Advanced Tab become available when this mode is selected.

UART

Component is configured to utilize UART interface. The UART Basic Tab and UART Advanced Tab become available when this mode is selected.

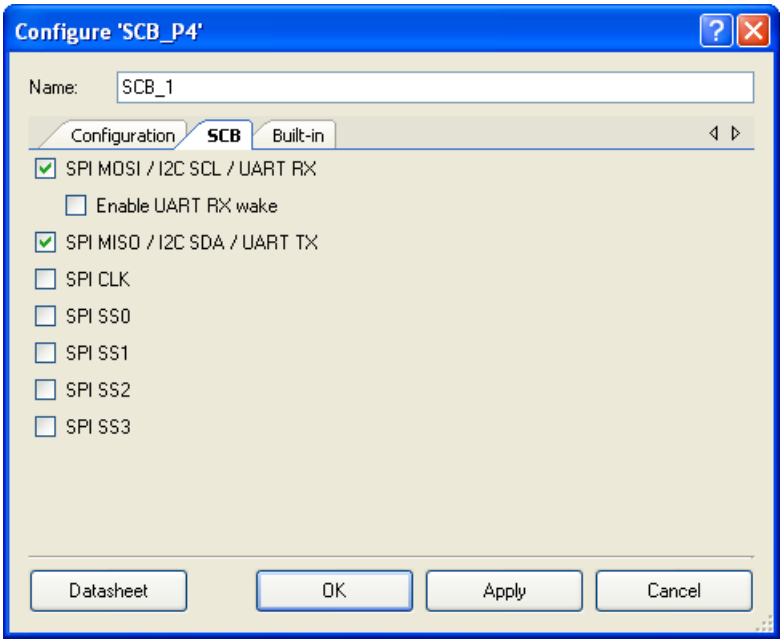
EZI2C

Component is configured to utilize EZI2C interface. The EZI2C Tab becomes available when this mode is selected.

EZSPI

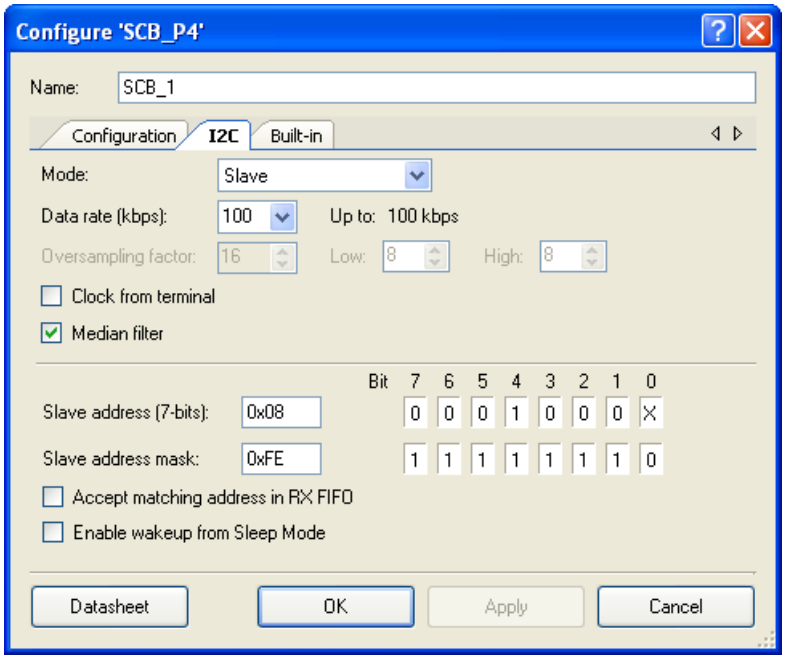
Component is configured to utilize EZSPI interface. The EZSPI Tab becomes available when this mode is selected.

SCB Tab



Allows to choice input and output component pins.

I2C Tab



Mode

This option determines what modes are supported, Slave, Master, Multi-Master, or Multi-Master-Slave.

Mode	Description
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; 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.

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 component clock.

Oversampling Factor

This parameter defines oversampling factor of I^2C SCL clock: the number of component clocks with in one I^2C SCL clock. Oversampling factor consists from oversampling factor low and high: when oversampling factor value is even it divides for two equal parts, when it is odd the oversampling low is greater by 1. An oversampling factor maximum value is 32 and minimum depends on median filter option when it is enabled 6 and 5 when disabled.

Oversampling Factor Low

This parameter defines oversampling factor low of I^2C SCL clock: the number of component clocks with in one low period I^2C SCL clock. An oversampling factor low maximum value is 16 and minimum depends on median filter option when it is enabled 4 and 3 when disabled.

Oversampling Factor High

This parameter defines oversampling factor high of I^2C SCL clock: the number of component clocks with in one high period I^2C SCL clock. An oversampling factor high maximum value is 16 and minimum depends on median filter option when it is enabled 4 and 3 when disabled.

Enable Median Filter

Applies 3 taps digital 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

This is the I^2C address that will be recognized by the slave. If slave operation is not selected, this parameter is ignored. A slave address between 0 and 127 (0x08 and 0x7F) may be selected; the default is **8**. This address is the 7-bit right-justified slave address and does not include the R/W bit.

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.

If a 10-bit slave address is required, you must use software address decoding and provide decode support for the second byte of the 10-bit address in the ISR.

Slave Address Mask

This parameter is used to mask bit of slave address while address match procedure. Slave address mask bit equal to zero value makes bit in slave address as doesn’t care. The bit 0 of slave address mask is always zero because it corresponds R/W bit.

For example: Slave address is 0x36 and Slave Address Mask is 0xEE (bit 0 and bit4 are doesn’t care). The matched slave addresses are: 0x36 and 0x26.

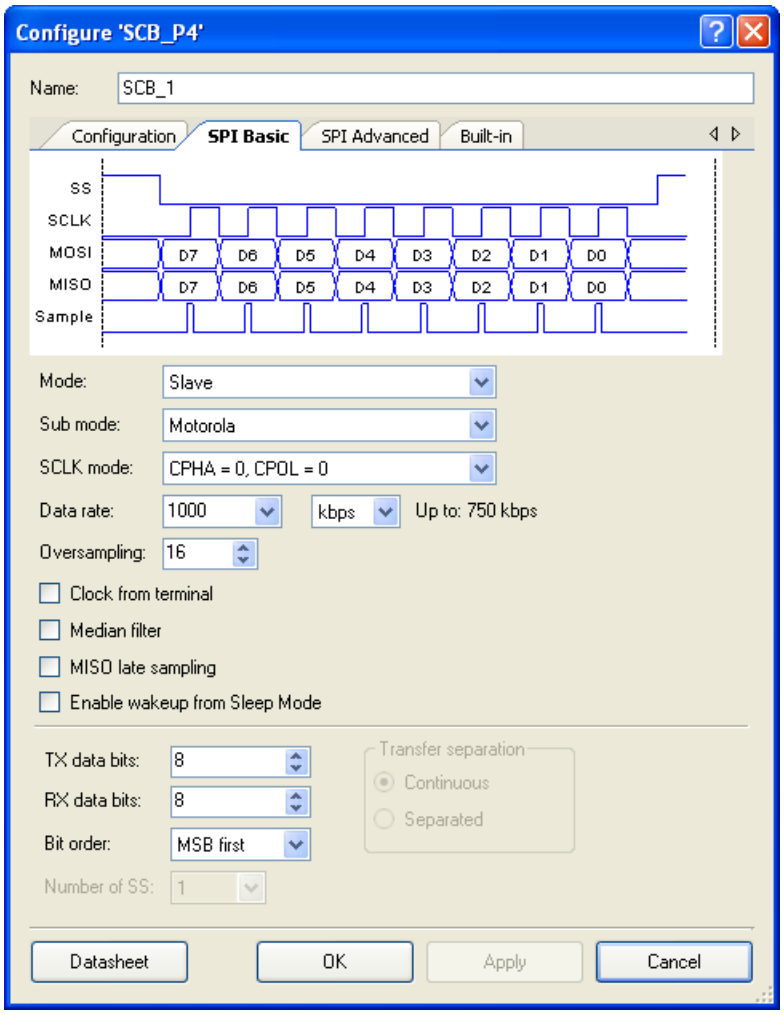
The value may be entered as decimal or hexadecimal; for hexadecimal numbers type ‘0x’ before the address. The binary input format is provided as well.

Enable wakeup from Sleep Mode

This option allows the system to be awakened from sleep when an address match occurs. The I^2C clock is stretched after matched address was received. The I^2C slave executes clock stretching and waits until device wakeup and ACKs the address. This option is only valid if **Mode** is Slave or Muti-Master-Slave.

You must enable the possibility for the I^2C to wake up the device on slave address match while switching to the sleep mode. You can do this by calling the SCB_Sleep() API; also refer to the Wakeup on Hardware Address Match section and to the “Power Management APIs” section of the *System Reference Guide*.

SPI Basic Tab



Mode

This option determines what mode is supported Slave or Master.

Sub mode

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

Sub mode	Description
Motorola	The original SPI protocol is defined by Motorola. It is a full duplex protocol: transmission and reception occur at the same time (default).
TI (Start Coincides)	The Texas Instruments' SPI protocol redefines the use of the Slave Select signal. It uses the signal to indicate the start of a data transfer, rather than a low active slave select signal. Start Coincides means that this pulse coincides with the transmission of the first data bit.
TI (Start Precedes)	The Texas Instruments' SPI protocol redefines the use of the Slave Select signal. It uses the signal to indicate the start of a data transfer, rather than a low active slave select signal. Start Precedes means that this pulse occurs one cycle before the transmission of the first data bit.
National Semiconductor	The National Semiconductor's SPI protocol is a half-duplex protocol (transmission happens before reception).

SCLK mode

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

Clock from terminal

Determines whether the clock is an internally generated clock or an internal clock but from a terminal on the component.

Bit rate

This parameter is used to set the SPI Master bit rate value up to 16000 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 16000 in multiples of 2000 kbps.

Oversampling factor

This parameter determines the number of internal clocks for each SPI clock period. The default value is a **4**. Any integer from 4 to 16 is a valid setting. If the **Enable median filter** is checked the minimum **Oversampling factor** value is 6. If the **Enable median filter** is checked and the **Oversampling factor** is set less than 6, it is automatically changed to a value of 6. It is not changed back if the **Enable median filter** is then unchecked.

Median filter

This option allows to use a 3-input median filter. The default value is a **Disabled**.

Enable late MISO sample

This option allows to change the SCLK edge on which MISO is captured. The default value is a **Disabled**.

Sleep wakeup

This option allows the system to be awakened from sleep on slave select.nable or disable waking from deep sleep on slave select. This option is only valid if **Mode** is Slave.

TX data bits

The number of **TX Data Bits** 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

The number of **RX data bits** 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.

Notes:

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

Bits 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.

SS number

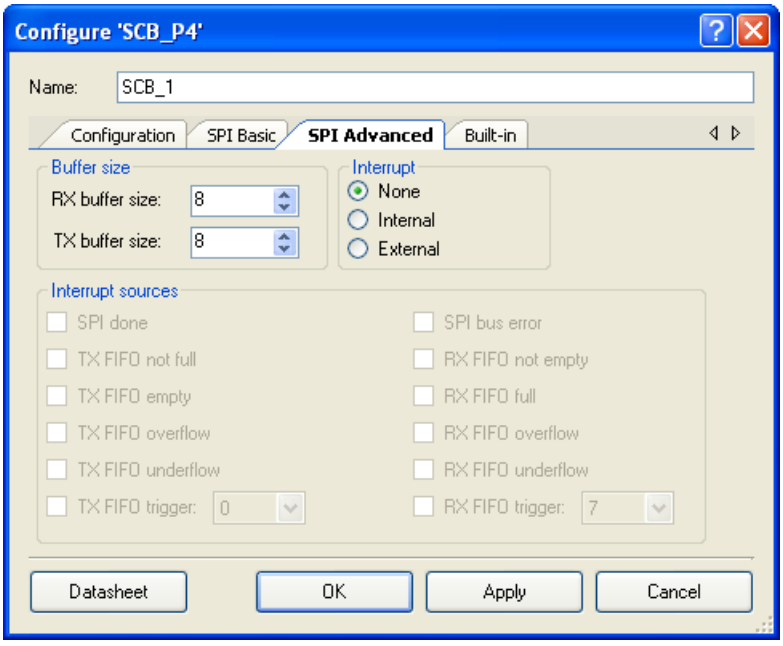
This parameter determines the number of SPI slave select lines. The default number of lines is a 1 line. Any integer from 1 to 4 is a valid setting. This option is only valid in **Master** mode.

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 **Error! Reference source not found.** for more information.

Transfer separation	Description
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

SPI Advanced Tab



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 byte/words of the RX FIFO is implemented in the hardware. Automatically set to 8 if a number less than 8 is entered. All other values up to 2^{32} use the 8-byte/word RX FIFO and a software buffer controlled by the supplied API. In this case the real buffer size will be limited only available memory.

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 byte/words of the TX FIFO is implemented in the hardware. Automatically set to 8 if a number less than 8 is entered. All other

values up to 2^{32} use the 8-byte/word TX FIFO and a software buffer controlled by the supplied API. In this case the real buffer size will be limited only available memory.

Interrupt

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

The **None** option removes interrupt component from design.

The **Internal** options allow you to use the predefined TX and RX ISRs of the component, or supply your own custom function which can be registered with the internal interrupt routine and called before the internal interrupt processing completes.

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

If the RX or TX buffer size is greater than 8, the component automatically reserves interrupt sources required for proper internal buffer operations. The internal ISR is used to handle data transferring to/from the hardware TX/RX FIFO.

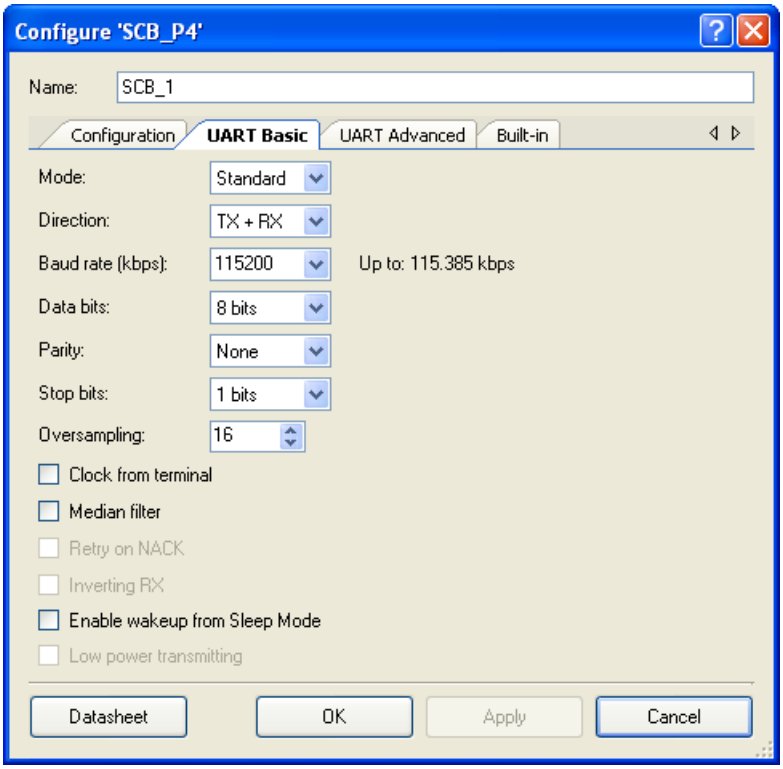
Notes:

- For buffer sizes greater than 8 bytes/words, the component automatically enables the internal interrupt. In addition, the global interrupt enable must be explicitly enabled for proper buffer handling.
- When RX buffer size is greater than 8 bytes/words, the ‘RX FIFO NOT EMPTY’ interrupt is reserved by the component. Do not clear or disabled because it causes incorrect internal buffer operation.
- When TX buffer size is greater than 8 bytes/words, the ‘TX FIFO NOT FULL’ interrupt is reserved by the component. It is enabled when TX data is available and disabled when there is no data to transmit. Do not clear or disabled because it causes incorrect internal buffer operation.

Interrupt sources

The **Interrupt sources** selection allow you to configure sources of interrupt event. Interrupt generation is a masked OR of all of the enabled TX and RX interrupt sources. The bits chosen with these parameters define the mask implemented with the initial component configuration.

UART Basic Tab



Mode

This option determines the operating mode of the UART interface : 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), RS232 Receiver (**RX Only**) or Transmitter (**TX Only**).

Clock from terminal

Determines whether the clock is an internally generated clock or an internal clock but from a terminal on the component.

Baud Rate

This parameter defines the baud-rate or bit-width configuration of the hardware for clock generation up to 921600. 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 using Mark/Space parity. Mark/Space parity should be selected if you are using 9 data bits mode.

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 determines the oversampling rate for the interface. The default value is a **16**. Any integer from 8 to 16 is a valid setting.

Enable median filter

This option allows to use a 3-input median filter. The default value is a **Disabled**.

Retry on NACK

Enables retry on NACK functionality of the SmartCard mode. This option is only valid if **Mode** is SmartCard.

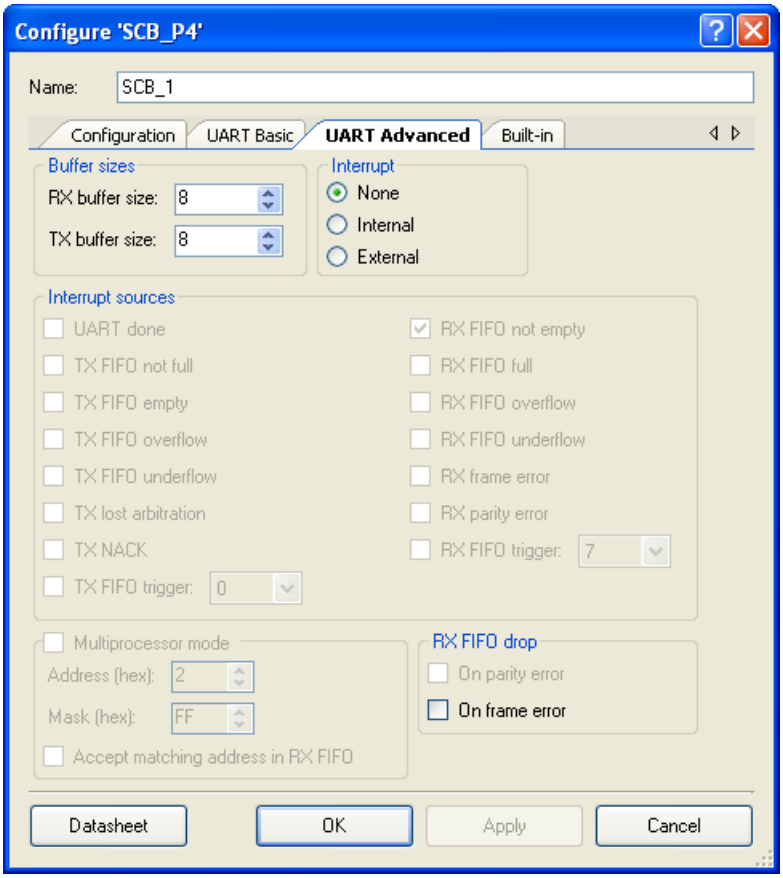
Inverting RX

Enables the inversion of the received signal for IrDA operation. This option is only valid if **Mode** is IrDA.

Sleep wakeup

This option allows the system to be awakened from sleep on on start bit This option is only valid in **TX only** mode.

UART Advanced Tab



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 byte/words of the RX FIFO is implemented in the hardware. Automatically set to 8 if a number less than 8 is entered. All other values up to 2^{32} use the 8-byte/word RX FIFO and a software buffer controlled by the supplied API. In this case the real buffer size will be limited only available memory.

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 byte/words of the TX FIFO is implemented in the hardware. Automatically set to 8 if a number less than 8 is entered. All other values up to 2^{32} use the 8-byte/word TX FIFO and a software buffer controlled by the supplied API. In this case the real buffer size will be limited only available memory.

Interrupt

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

The **None** option removes interrupt component from design.

The **Internal** options allow you to use the predefined TX and RX ISRs of the component, or supply your own custom function which can be registered with the internal interrupt routine and called before the internal interrupt processing completes.

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

If the RX or TX buffer size is greater than 8, the component automatically reserves interrupt sources required for proper internal buffer operations. The internal ISR is used to handle data transferring to/from the hardware TX/RX FIFO.

Notes:

- For buffer sizes greater than 8 bytes/words, the component automatically enables the internal interrupt. In addition, the global interrupt enable must be explicitly enabled for proper buffer handling.
- When RX buffer size is greater than 8 bytes/words, the ‘RX FIFO NOT EMPTY’ interrupt is reserved by the component. Do not clear or disabled because it causes incorrect internal buffer operation.
- When TX buffer size is greater than 8 bytes/words, the ‘TX FIFO NOT FULL’ interrupt is reserved by the component. It is enabled when TX data is available and disabled when there is no data to transmit. Do not clear or disabled because it causes incorrect internal buffer operation.

Interrupt sources

The **Interrupt sources** selection allow you to configure sources of interrupt event. Interrupt generation is a masked OR of all of the enabled TX and RX interrupt sources. The bits chosen with these parameters define the mask implemented with the initial component configuration.

Multiprocessor Mode

Enables the multiprocessor mode where the first bit of 9 bits indicates an address. The default value is a **Disabled**.

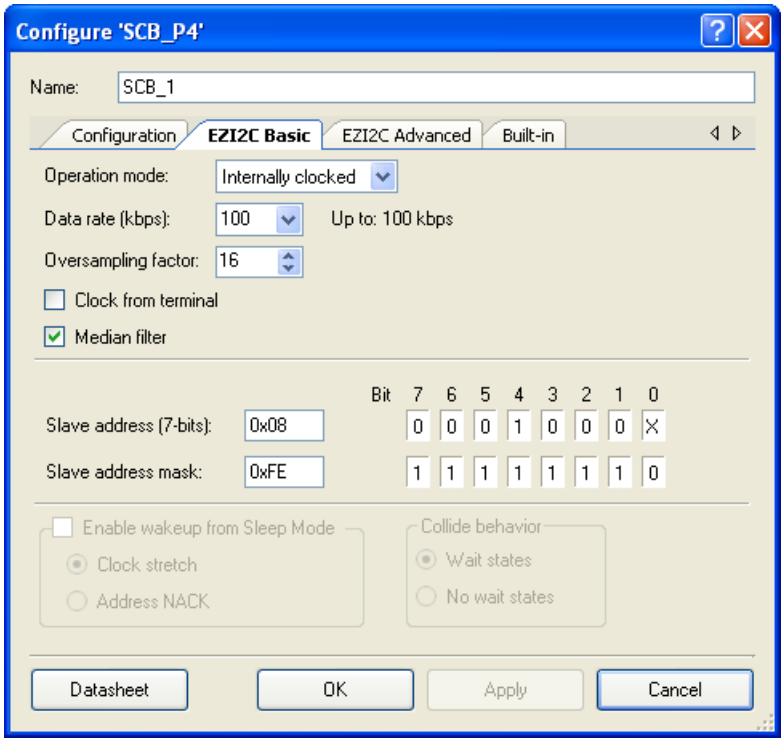
Multiprocessor Address

Address to match when multiprocessor mode is enabled. The default value is **0x02**.

Multiprocessor Address Mask

Mask of bits that are used when matching to the multiprocessor address. The default value is **0xFF**.

EZI2C Tab



Clock mode

Determines the operating mode between Internally and Externally clocked. The default value is **Internally clocked**.

Data Rate

This parameter is used to set the I^2C data rate value up to 1000 kbps; 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.

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 component clock.

Oversampling Factor

This parameter defines oversampling factor of I^2C SCL clock: the number of component clocks with in one I^2C SCL clock. An oversampling factor maximum value is 32 and minimum depends on median filter option when it is enabled 6 and 5 when disabled.

Median filter

This option allows to use a 3-input median filter. The default value is a **Disabled**.

Collide behavior

Determines how to handle a collision between the SCB and the CPU when accessing registers. The default value is a **Wait states**.

Slave Address

Address of the slave that the EZI2C will respond to. The default value is a **0x08**.

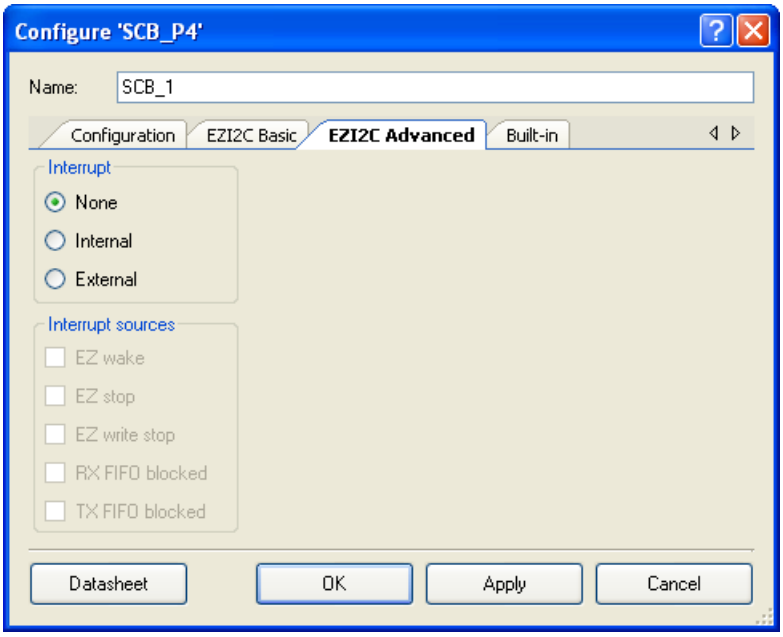
Slave Address Mask

Mask of bits that are used when matching to the slave address. The default value is a **0xFE**.

Enable wakeup from Sleep Mode

This option allows the system to be awakened from sleep when an address match occurs. The two options of I^2C behavior after address match occurred exist: Address ACK and Address NACK. **Clock stretch** - the I^2C slave executes clock stretching and waits until device wakeup and ACKs the address. **Address NACK** - the I^2C slave NACKs the address immediately, master has to poll the slave again after device wakeup time passed.

EZI2C Advanced Tab



Interrupt

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

The **None** option removes interrupt component from design.

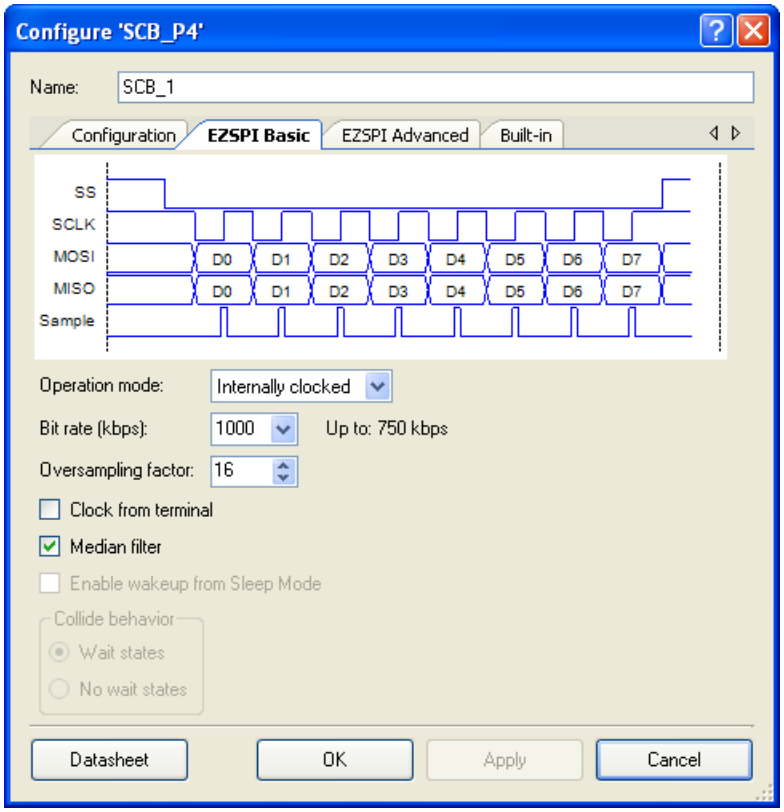
The **Internal** option allows usage internal interrupt which is configured by component. The component interrupt handler contains the component code required for operation in this case. The interface to customer interrupt handler is provided to call customer function on every interrupt trigger.

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

Interrupt sources

The **Interrupt sources** selection allow you to configure sources of interrupt event. Interrupt generation is a masked OR of all of the enabled I2C, TX and RX interrupt sources. The bits chosen with these parameters define the mask implemented with the initial component configuration.

EZSPI Tab



Clock source

Determines the operating mode between Internal and External clock and also determines the source of the Internal clock. The default value is **Internal generated clock**.

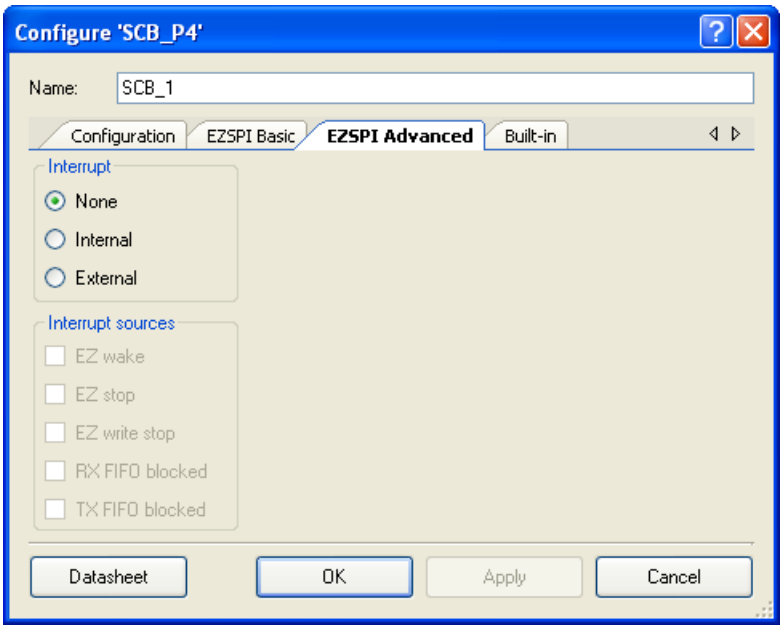
Median filter

This option allows to use a 3-input median filter. The default value is a **Disabled**.

Collide behavior

Determines how to handle a collision between the SCB and the CPU when accessing registers. The default value is a **Wait states**.

EZSPI Advanced Tab



Interrupt

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

The **None** option removes interrupt component from design.

The **Internal** option allows usage internal interrupt which is configured by component. The component interrupt handler contains the component code required for operation in this case. The interface to customer interrupt handler is provided to call customer function on every interrupt trigger.

The **External** option removes internal interrupt and provides interrupt terminal. The interrupt component can be connected to it if customer interrupt handler desired.

Interrupt sources

The **Interrupt sources** selection allow you to configure sources of interrupt event. Interrupt generation is a masked OR of all of the enabled I2C, TX and RX interrupt sources. The bits chosen with these parameters define the mask implemented with the initial component configuration.

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 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 for that mode.

Placement

All placement information is provided to the API through the *cyfitter.h* file.

Resources

Mode		Resource Type	API Memory (Bytes)	
		SCB Fixed Blocks	Flash	RAM
Unconfigured		1		
EZI2C		1		
EZSPI		1		
I2C	Slave	1		
	Master	1		
	Multi-Master	1		
	Multi-Master-Slave	1		
SPI	Slave	1		
	Master	1		
UART	Standard	1		
	SmartCard	1		
	IrDA	1		

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component during runtime. The following table lists and describes the interface to each function. The subsequent sections cover 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”.

Common Functions

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 DeepSleep.
SCB_Wakeup()	Prepares component to exit DeepSleep.
SCB_SaveConfig()	Do nothing.
SCB_RestoreConfig()	Do nothing.

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: Initialize the SCB component according to defined parameters in the customizer.

Parameters: None

Return Value: None

Side Effects: All registers will be set to values according to the customizer Configure dialog.

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.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Start(void)

Description: Starts the SCB. Invokes the SCB_Init() and SCB_Enable() functions. The SCB_initVar is set to 1 to notify that component was initialized.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Stop(void)

Description: Disable the SCB component.

Parameters: None

Return Value: None

Side Effects: None

void SCB_Sleep(void)

Description: Prepares component to enter DeepSleep. The “EnableWakeup” selection has an influence on this function implementation. For configurations with slave functionality and wake on address match selection, the clocking mode will be switched to external clocking mode.

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

Parameters: None

Return Value: None

Side Effects: None

void SCB_Wakeup(void)

Description: Prepares component to exit DeepSleep. The “EnableWakeup” selection has influence to on this function implementation.

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_SaveConfig(void)

Description: Do nothing.

Parameters: None

Return Value: None

Side Effects: None

void SCB_RestoreConfig(void)

Description: Do nothing.

Parameters: None

Return Value: None

Side Effects: None

I2C Specific Functions

Multi-Master-Slave incorporates Slave and Multi-Master functions.

Function	Description
SCB_I2CInit()	Configures the SCB for I2C operation.

Function	Description
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 255 (0x00 to 0xFF).
SCB_I2CSlaveInitReadBuf()	Sets up the slave receive data buffer (master <- slave).
SCB_I2CSlaveInitWriteBuf()	Sets up the slave write buffer (master -> slave).
SCB_I2CSlaveGetReadBufSize()	Returns the number of bytes read by the master since the buffer was reset.
SCB_I2CSlaveGetWriteBufSize()	Returns the number of bytes written by the master since the buffer was reset.
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()	Sends only a start to the specific address.
SCB_I2CMasterSendRestart()	Sends only a restart to the specified 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 byte count of data read since the SCB_I2CMasterClearReadBuf() function was called.
SCB_I2CMasterGetWriteBufSize()	Returns the byte count of the data written since the SCB_I2CMasterClearWriteBuf() function was called.
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_state	Current state of I^2C state machine.
SCB_mstrStatus	Current status of I^2C master.
SCB_mstrControl	Controls master end of transaction with or without generating a stop.
SCB_mstrRdBufPtr	Pointer to master read buffer.
SCB_mstrRdBufSize	Size of master read buffer.
SCB_mstrRdBufIndex	Current index within master read buffer.
SCB_mstrWrBufPtr	Pointer to master write buffer.
SCB_mstrWrBufSize	Size of master write buffer.
SCB_mstrWrBufIndex	Current index within master write buffer.
SCB_slStatus	Current status of I^2C slave.
SCB_slRdBufPtr	Pointer to slave read buffer.
SCB_slRdBufSize	Size of slave read buffer.
SCB_slRdBufIndex	Current index within slave read buffer.
SCB_slWrBufPtr	Pointer to slave write buffer.
SCB_slWrBufSize	Size of slave write buffer.
SCB_slWrBufIndex	Current index within slave write buffer.

void SCB_ I2CInit(SCB_I2C_INIT_STRUCT *config)

Description: Configures the SCB for I2C operation. This function is **intended specifically** to be used when the component is set to “Unconfigured SCB” in the customizer. To initialize the SCB based on the I2C configuration set in the customizer use the SCB_Init() function. After initializing the SCB, the component can be enabled using the SCB_Enable() function.

This function uses a structure to provide 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
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
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.
oversampleHigh	Oversampling factor for the high phase of the I2C clock. Ignored for Slave mode operation.
enableMedianFilter	0 – disable 1 – enable
slaveAddr	7 bit slave address. Ignored for non-slave modes.
slaveAddrMask	8 bit slave address mask. Bit 0 must have a value of 0. Ignored for non-slave modes.
acceptAddr	0 – disable 1 – enable When enabled the matching address is received into the Rx FIFO.
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^2C slave.

Slave Status Constants	Description
SCB_I2C_SSTAT_RD_CMPT	Slave read transfer complete. Set when master indicates it is done reading by sending a NAK
SCB_ I2C_SSTAT_RD_BUSY	Slave read transfer in progress. Set when master addresses slave with a read, cleared when RD_CMPT is set.
SCB_ I2C_SSTAT_RD_ERR_OVFL	Master attempted to read more bytes than are in buffer.
SCB_ I2C_SSTAT_WR_CMPT	Slave write transfer complete. Set at reception of a Stop condition, or when WR_ERR_OVFL is set
SCB_ I2C_SSTAT_WR_BUSY	Slave write transfer in progress. Set when the master addresses the slave with a write, cleared at reception of a Stop condition or when WR_ERR_OVFL is set
SCB_ I2C_SSTAT_WR_ERR_OVFL	Master attempted to write past end of buffer.

Side Effects: None

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: None

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: None

void SCB_I2CSlaveSetAddress(uint32 address)

Description: Sets the I^2C slave address

Parameters: uint32 address: I^2C slave address. This value may be any address between 8 and 120 (0x08 to 0x78). This address is the 7-bit right-justified slave address and does not include the R/W bit.

Return Value: None

Side Effects: None

void SCB_I2CSlaveSetAddressMask(uint32 addressMask)

Description: Sets the I^2C slave address

Parameters: uint32 addressMask: I^2C slave address mask. This value may be any between 0 and 254 (0x00 to 0xFE).

Return Value: None

Side Effects: None

void SCB_I2CSlaveInitReadBuf(uint8 * rdBuf, uint32 bufSize)

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

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

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

Return Value: None

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

void SCB_I2CSlaveInitWriteBuf(uint8 * wrBuf, uint32 bufSize)

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

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

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

Return Value: None

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

uint32 SCB_I2CSlaveGetReadBufSize(void)

Description: Returns the number of bytes read by the I^2C master since an SCB_I2CSlaveInitReadBuf() or SCB_I2CSlaveClearReadBuf() function was executed. The maximum return value is the size of the read buffer.

Parameters: None

Return Value: uint32: Bytes read by master.

Side Effects: None

uint32 SCB_I2CSlaveGetWriteBufSize(void)

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

Parameters: None

Return Value: uint32: Bytes written by master.

Side Effects: None

void SCB_I2CSlaveClearReadBuf(void)

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

Parameters: None

Return Value: None

Side Effects: None

void SCB_I2CSlaveClearWriteBuf(void)

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

Parameters: None

Return Value: None

Side Effects: None

uint32 SCB_I2CMasterStatus(void)

Description: Returns the master's communication status.

Parameters: None

Return Value: uint32: Current status of I^2C master. I^2C master status constants may be ORed together.

Master status constants	Description
SCB_ I2C_MSTAT_RD_CMPLT	Read transfer complete. The error condition bits must be checked to ensure that read transfer was successful.
SCB_ I2C_MSTAT_WR_CMPLT	Write transfer complete. The error condition bits must be checked to ensure that write transfer was successful.
SCB_ I2C_MSTAT_XFER_INP	Transfer in progress
SCB_ I2C_MSTAT_XFER_HALT	Transfer has been halted. The I^2C bus is waiting for restart or stop condition generation.
SCB_ I2C_MSTAT_ERR_SHORT_XFER	**Error condition**: Write transfer completed before all bytes were transferred.
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_START_ABORT	**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
SCB_ I2C_MSTAT_ERR_BUS_ERROR	**Error condition**: bus error was occurred with master while start generation or misplaced start condition on the bus
SCB_ I2C_MSTAT_ERR_XFER	**Error condition**: This is the ORed value of error conditions provided above. If all error condition bits are cleared, but this is set, the transfer was aborted due to Slave operation.

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 in byte-by-byte mode. Enables I^2C interrupt.

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.

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

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 in byte by byte mode. Enables I^2C interrupt.

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 R_nW)

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

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

uint32 R_nW: Set to zero, send write command; set to nonzero, send read command.

Return Value: uint32: Error Status.

Mode Constants	Description
SCB_I2C_MSTR_NO_ERROR	Function complete without error.
SCB_I2C_MSTR_BUS_BUSY	Bus is busy occurred, start condition generation not started.
SCB_I2C_MSTR_NOT_READY	Master is not valid master on the bus. The Slave operation may be in progress.
SCB_I2C_MSTR_ERR_LB_NAK	**Error condition**: Last byte was NAKed.
SCB_I2C_MSTR_ERR_ARB_LOST	**Error condition**: Master lost arbitration while start is generated. (This status is only valid if Multi-Master enabled.)
SCB_I2C_MSTR_ABORT_XFER	**Error condition**: The start condition generation was aborted due to start of Slave operation. (This status is only valid in Multi-Master-Slave mode.)

Side Effects: This function is blocking and doesn't exit until the start condition is generated.

uint32 SCB_MasterSendRestart(uint32 slaveAddress, uint32 R_nW)

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

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

uint32 R_nW: Set to zero, send write command; set to nonzero, send read command.

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

Side Effects: This function is blocking and doesn't exit until the repeat-start condition is generated.

uint32 SCB_I2CMasterSendStop(void)

Description: Generates I^2C stop condition on bus. This function does nothing if start or restart conditions failed before this function was called.

Parameters: None

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

Side Effects: This function is blocking and doesn't exit until:

Master: waits until a stop condition is generated.

Multi-Master, Multi-Master-Slave: waits until a stop condition is generated or arbitrage is lost on ACK/NAK bit.

uint32 SCB_I2CMasterWriteByte(uint32 theByte)

Description: Sends one byte to a slave. 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.

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

Return Value: uint32: Error Status.

Mode Constants	Description
SCB_I2C_MSTR_NO_ERROR	Function complete without error.
SCB_I2C_MSTR_NOT_READY	Master is not valid master on the bus. The Slave operation may be in progress.
SCB_I2C_MSTR_ERR_LB_NAK	Last byte was NAKed.
SCB_I2C_MSTR_ERR_ARB_LOST	**Error condition**: Master lost arbitration while start is generated. (This status is only valid if Multi-Master is enabled.)

Side Effects: This function is blocking and doesn't exit until byte is transmitted.

uint32 SCB_I2CMasterReadByte(uint32 acknNak)

Description: Reads one byte from a slave and ACKs or NAKs the transfer. A valid start or restart condition must be generated before calling this function. This function does nothing and returns a zero value if start or restart conditions failed before this function was called.

Parameters: uint32 acknNak: If zero, sends a NAK; if nonzero sends an ACK.

Return Value: uint32: Byte read from the slave

Side Effects: This function is blocking and doesn't exit until byte is received.

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: None

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 the byte count transferred so far.

Side Effects: None

void SCB_I2CMasterClearReadBufSize(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_I2CMasterClearWriteBufSize(void)

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

Parameters: None

Return Value: None

Side Effects: None

SPI Specific Functions

Function	Description
SCB_SpiInit()	Configures the SCB for SPI operation.
SCB_SpiSetActiveSlaveSelect()	Selects the active slave select line.

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 component is set to “Unconfigured SCB” in the customizer. To initialize the SCB based on the SPI configuration set in the customizer use the SCB_Init() function. After initializing the SCB, the component can be enabled using the SCB_Enable() function.

This function uses a structure to provide 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
mode	Mode of operation for SPI. The following defines are available choices: SCB_SPI_SLAVE SCB_SPI_MASTER
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
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
oversample	Oversampling factor for the SPI clock. Ignored for Slave mode operation.
enableMedianFilter	0 – disable 1 – enable
enableLateSampling	0 – disable 1 – enable Ignored for slave mode.
enableWake	0 – disable 1 – enable Ignored for master mode.
dataBitsRx	Number of data bits for RX direction. Different dataBitsRx and dataBitsTx are only alowed for National submode.
dataBitsTx	Number of data bits for TX direction. Different dataBitsRx and dataBitsTx are only alowed for National submode.
bitOrder	Determines the bit ordering. The following defines are available choices: SCB_BITS_ORDER_LSB_FIRST SCB_BITS_ORDER_MSB_FIRST

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
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 SW buffer.
rxBuffer	Buffer space provided for a RX SW buffer: • The buffer of rxBufferSize in words must be provided. If the dataBitsRx is greater than 8 then (2* rxBufferSize) bytes must be provided. • The NULL pointer must be provided if hardware buffering implies.
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 SW buffer.
txBuffer	Buffer space provided for a RX SW buffer: • The buffer of rxBufferSize in words must be provided. • If the dataBitsRx is greater than 8 then (2* rxBufferSize) bytes must be provided. • The NULL pointer must be provided if hardware buffering implies.
enableInterrupt	0 – disable 1 – enable
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
rxTriggerLevel	FIFO level for an RX trigger interrupt. This value is written regardless of whether the RX trigger interrupt is enabled.

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
txTriggerLevel	FIFO level for a TX trigger interrupt. This value is written regardless of whether the TX trigger interrupt is enabled.

Return Value: None

Side Effects: None

SCB_SpiSetActiveSlaveSelect(uint32 activeSS)

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 activeSS: 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

UART Specific Functions

Function	Description
SCB_UartInit()	Configures the SCB for SPI 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.
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.

Function	Description
SCB_UartSetRxAddressMask()	Sets the hardware address mask for the UART in Multiprocessor mode.

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 component is set to “Unconfigured SCB” in the customizer. To initialize the SCB based on the UART configuration set in the customizer use the SCB_Init() function. After initializing the SCB, the component can be enabled using the SCB_Enable() function.

This function uses a structure to provide 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
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
direction	Direction of operation for the UART. The following defines are available choices: SCB_UART_TX_RX SCB_UART_RX SCB_UART_TX
dataBits	Number of data bits
parity	Determines the parity. The following defines are available choices: SCB_UART_PARITY_EVEN SCB_UART_PARITY_ODD SCB_UART_PARITY_NONE
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
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
enableIrdaLowPower	IrDA low power RX mode is enabled. 0 – disable 1 – enable The TX functionality doesn’t work when enabled.

enableMedianFilter	0 – disable 1 – enable
enableRetryNack	0 – disable 1 – enable Ignored for modes other than SmartCard.
enableInvertedRx	0 – disable 1 – enable Ignored for modes other than IrDA.
dropOnParityErr	Drop data from RX FIFO and lost it if parity error detected. 0 – disable 1 – enable
dropOnFrameErr	Drop data from RX FIFO and lost it if frame error detected. 0 – disable 1 – enable
enableWake	0 – disable 1 – enable Ignored for modes other than standard UART. The RX functionality has to be enabled.
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 SW buffer.
rxBuffer	Buffer space provided for a RX SW buffer: • The buffer of rxBufferSize in words must be provided. If the dataBitsRx is greater than 8 then (2* rxBufferSize) bytes must be provided. • The NULL pointer must be provided if hardware buffering implies.
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 SW buffer.
txBuffer	Buffer space provided for a RX SW buffer: • The buffer of rxBufferSize in words must be provided. • If the dataBitsRx is greater than 8 then (2* rxBufferSize) bytes must be provided. • The NULL pointer must be provided if hardware buffering implies.
enableMultiproc	Enables multiprocessor mode. 0 – disable 1 – enable
multiprocAddr	8 bit address to match in Multiprocessor mode. Ignored for other modes.
multiprocAddrMask	8 bit mask of address bits that are compared for a Multiprocessor address match. Ignored for other modes.

enableInterrupt	0 – disable 1 – enable
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
rxTriggerLevel	FIFO level for an RX trigger interrupt. This value is written regardless of whether the RX trigger interrupt is enabled.
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
txTriggerLevel	FIFO level for a TX trigger interrupt. This value is written regardless of whether the TX trigger interrupt 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

Parameters: uint32 txDataByte: the data to be transmitted

Return Value: None

Side Effects: None

void SCB_UartPutString(char8 const 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: char8 const 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 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: 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: None

uint32 SCB_UartGetByte(void)

Description: Retrieves next data element from the receive buffer, returns received character 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: None

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.
'0' – address bit doesn't care while comparison,
'1' – address bit is significant while comparison

Return Value: None

Side Effects: None

Common SPI/UART Functions

Function	Description
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.

Function	Description
SCB_SpiUartGetRxBufferSize()	Returns the number of received data elements in the receive buffer.
SCB_SpiUartClearRxBuffer()	Clear the receive buffer and RX FIFO.

void SCB_SpiUartWriteTxData(uint32 txDataByte)

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 txDataByte: the data to be transmitted.

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.

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 disabled: returns the number of used entries in TX FIFO.
- TX software buffer 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 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: Next data element from the receive buffer.

Side Effects: None

uint32 SCB_SpiUartGetRxBufferSize(void)

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

- RX software buffer disabled: returns the number of used entries in RX FIFO.
- RX software buffer enabled: returns the number of elements which were placed in receive buffer.

Parameters: None

Return Value: uint32: Number of received data elements

Side Effects: None

void SCB_SpiUartClearRxBuffer(void)

Description: Clear the receive buffer and RX FIFO.

Parameters: None

Return Value: None

Side Effects: None

EZI2C Specific Functions

Function	Description
SCB_EZI2CInit()	Configures the SCB for EZI2C operation.

void SCB_EZI2CInit(SCB_EZI2C_INIT_STRUCT *config)

Description: Configures the SCB for EZI2C operation. This function is intended specifically to be used when the component is set to “Unconfigured SCB” in the customizer. To initialize the SCB based on the UART configuration set in the customizer use the SCB_Init() function. After initializing the SCB, the component can be enabled using the SCB_Enable() function.

This function uses a structure to provide 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
clockSource	Determines whether an internal clock or the external I2C clock is used for operation. The following defines are the available choices: SCB_EZ_INTERNALLY_CLOCKED SCB_EZ_EXTERNALLY_CLOCKED
enableMedianFilter	0 – disable 1 – enable
enableWaitStates	Determines whether wait states are inserted on collisions or if the read/write from the CPU fails. 0 – disable 1 – enable
slaveAddr	7 bit slave address. Ignored for non-slave modes.
slaveAddrMask	8 bit slave address mask. Bit 0 must have a value of 0. Ignored for non-slave modes.
enableWake	0 – disable 1 – enable
wakeAction	Determines what action is taken when a wake address matches. Ignored when wake from Deep Sleep is disabled. The following defines are the available choices: SCB_EZI2C_WAKE_CLOCK_STRETCH SCB_EZI2C_WAKE_ADDRESS_NACK
ezInterruptMask	Mask of interrupt sources to enable for EZ mode. 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. Internally clocked operation mode: • SCB_INTR_SLAVE_I2C_STOP • SCB_INTR_SLAVE_I2C_WRITE_STOP Externally clocked operation mode: • SCB_INTR_I2C_EC_WAKE_UP • SCB_INTR_I2C_EC_EZ_STOP • SCB_INTR_I2C_EC_EZ_WRITE_STOP

rxInterruptMask	Mask of interrupt sources to enable for RX mode. This mask is written regardless of the setting of the enableInterrupt field. SCB_INTR_RX_BLOCKED
txInterruptMask	Mask of interrupt sources to enable for TX mode. This mask is written regardless of the setting of the enableInterrupt field. SCB_INTR_TX_BLOCKED

Return Value: None

Side Effects: None

EZSPI Specific Functions

Function	Description
SCB_EZSPIInit()	Configures the SCB for EZSPI operation.

void SCB_EZSPIInit(SCB_EZSPI_INIT_STRUCT *config)

Description: Configures the SCB for EZSPI operation. This function is intended specifically to be used when the component is set to “Unconfigured SCB” in the customizer. To initialize the SCB based on the UART configuration set in the customizer use the SCB_Init() function. After initializing the SCB, the component can be enabled using the SCB_Enable() function.

This function uses a structure to provide 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
clockSource	Determines whether an internal clock or the external I2C clock is used for operation. The following defines are the available choices: SCB_EZ_INTERNALLY_CLOCKED SCB_EZ_EXTERNALLY_CLOCKED
enableMedianFilter	0 – disable 1 – enable
enableWaitStates	Determines whether wait states are inserted on collisions or if the read/write from the CPU fails. 0 – disable 1 – enable
ezInterruptMask	Mask of interrupt sources to enable for EZ mode. 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. Internally clocked operation mode: • SCB_INTR_SLAVE_I2C_STOP • SCB_INTR_SLAVE_I2C_WRITE_STOP Externally clocked operation mode: • SCB_INTR_I2C_EC_WAKE_UP • SCB_INTR_I2C_EC_EZ_STOP • SCB_INTR_I2C_EC_EZ_WRITE_STOP
rxInterruptMask	Mask of interrupt sources to enable for RX mode. This mask is written regardless of the setting of the enableInterrupt field. SCB_INTR_RX_BLOCKED
txInterruptMask	Mask of interrupt sources to enable for TX mode. This mask is written regardless of the setting of the enableInterrupt field. SCB_INTR_TX_BLOCKED

Return Value: None

Side Effects: None

EZ Data Functions

Function	Description
SCB_EzWriteByte()	Writes one byte of data to the component hardware buffer at offset addr.
SCB_EzReadByte()	Reads one byte of data from the component hardware buffer at offset addr.
SCB_EzGetAddress()	Returns the address which was most recently accessed (written or read) by external master.
SCB_EzIsBusBusy()	Checks the state of the EZ interface and determines whether a transaction is active.

uint32 SCB_EzWriteByte(uint32 addr, uint32 dataByte)

Description: Writes one byte of data to the component hardware buffer at offset addr. The status of write operation is returned: zero value if write operation was successful and non-zero otherwise. The clocking mode has influence on the function implementation:

- Internally clocked: the write operations are always successful and zero value returned.
- Externally clocked: in case of conflict between internal and external logic the write operation is ignored and the non-zero value is returned by the function.

Parameters: uint32 addr: address within the buffer. The address range is not checked.

uint32 data: the byte required to be written.

Return Value: None

Side Effects: None

uint32 SCB_EzReadByte(uint32 addr)

Description: Reads one byte of data from the component hardware buffer at offset addr. The error value (0xFFFF FFFF) is returned when read operation was failed, the valid value is byte (0x0000 00XX). The clocking mode has influence on the function implementation:

- Internally clocked: the read operations are always successful and zero value returned.
- Externally clocked: in case of conflict between internal and external logic the value (0xFFFF FFFF) is returned by the function.

Parameters: uint32 addr: address within the buffer. The address range is not checked.

Return Value: None

Side Effects: None

uint32 SCB_EzGetAddress(void)

Description: Returns the address which was most recently accessed (written or read) by external master.

The clocking mode has influence on the return value: hardware returns a non-reliable value **when externally clocked** mode is selected. The value of 0xFFFF FFFF will be returned in this case.

Parameters: None

Return Value: uint32: address which was most recently accessed (written or read) by external master

Side Effects: None

uint32 SCB_EzIsBusBusy(void)

Description: Checks the state of the EZ interface and determines whether a transaction is active.

Parameters: None

Return Value: uint32: 0 – bus is free and 1 – bus is busy.

Side Effects: None

Interrupt Functions

Which specific interrupt API to use is dependent on the mode of operation.

Function	Description
SCB_EnableInt()	When using an Internal interrupt, this enables the interrupt in the NVIC
SCB_DisableInt()	When using an Internal interrupt, this disables the interrupt in the NVIC.
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
SCB_ClearMasterInterruptSource()	Clears the bit-mask of pending Master interrupt sources
SCB_SetMasterInterrupt()	Generates interrupt event from bit-mask of Master interrupt sources

Function	Description
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
SCB_SetI2CExtClkInterruptMode()	Configures which bits of the externally clocked I2C interrupt request register will trigger an interrupt event
SCB_GetI2CExtClkInterruptMode()	Returns the externally clocked I2C interrupt mask
SCB_GetI2CExtClkInterruptSourceMasked()	Returns the externally clocked I2C interrupt request register masked by interrupt mask
SCB_GetI2CExtClkInterruptSource()	Returns the bit-mask of pending externally clocked I2C interrupt sources
SCB_ClearI2CExtClkInterruptSource()	Clears the bit-mask of pending externally clocked I2C interrupt sources
SCB_SetI2CExtClkInterrupt()	Generates interrupt event from bit-mask of externally clocked I2C interrupt sources
SCB_SetSpiExtClkInterruptMode()	Configures which bits of the externally clocked SPI interrupt request register will trigger an interrupt event
SCB_GetSpiExtClkInterruptMode()	Returns the externally clocked SPI interrupt mask
SCB_GetSpiExtClkInterruptSourceMasked()	Returns the externally clocked SPI interrupt request register masked by interrupt mask
SCB_GetSpiExtClkInterruptSource()	Returns the bit-mask of pending externally clocked SPI interrupt sources
SCB_ClearSpiExtClkInterruptSource()	Clears the bit-mask of pending externally clocked SPI interrupt sources
SCB_SetSpiExtClkInterrupt()	Generates interrupt event from bit-mask of externally clocked SPI interrupt sources

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
SCB_INTR_CAUSE_I2C_EC	Interrupt from externally clocked I2C
SCB_INTR_CAUSE_SPI_EC	Interrupt from externally clocked SPI

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.

Interrupt causes constants	Description
SCB_INTR_TX_TRIGGER	Fewer entries in the transmitter FIFO 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_BLOCKED	CPU write to EZ memory failed due to an externally clock transaction
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 doesn't 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: None

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.

Interrupt causes constants	Description
SCB_INTR_RX_TRIGGER	More entries in the receiver FIFO 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_BLOCKED	CPU read of EZ memory failed due to an externally clock transaction
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: None

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.

Interrupt causes constants	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 negative acknowledgement
SCB_INTR_MASTER_I2C_ACK	I2C master acknowledgement
SCB_INTR_MASTER_I2C_STOP	I2C master generated STOP
SCB_INTR_MASTER_I2C_BUS_ERROR	I2C master bus error (unexpected detection of 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	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	Slave negative acknowledgement received.
INTR_SLAVE_I2C_ACK	Slave acknowledgement received.

INTR_SLAVE_I2C_WRITE_STOP	Stop or Repeated Start event for write transfer intended for this slave (address matching is performed). The event is detected only on write transfers that have EZ data written to the memory structure (a write transfer that only communicates an EZ address, will not result in this event being detected).
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 Start received.
INTR_SLAVE_I2C_ADDR_MATCH	I2C matching address received.
INTR_SLAVE_I2C_GENERAL	Slave general call address received.
INTR_SLAVE_I2C_BUS_ERROR	Slave bus error (unexpected detection of START or STOP condition).
INTR_SLAVE_SPI_EZ_WRITE_STOP	SPI slave deselected after EZ SPI write transfer occurred.
INTR_SLAVE_SPI_EZ_STOP	SPI slave deselected after any EZ SPI transfer occurred.
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

void SCB_SetI2CExtClkInterruptMode(uint32 interruptMask)

Description: Configures which bits of the externally clocked I2C interrupt request register will trigger an interrupt event

Parameters: uint32 interruptMask: Bit-mask of the externally clocked I2C interrupt sources to be enabled.

I2C Ext clock interrupt sources	Description
SCB_INTR_I2C_EC_WAKE	Wake up request. Active on incoming slave request (with address match).
SCB_INTR_I2C_EC_EZ_STOP	Stop event is detected, only active when transaction intended for this slave (address matching is performed).

SCB_INTR_I2C_EC_EZ_WRITE_STOP	Stop detection after a write transfer occurred. Only generated when a write transfer to the EZ memory occurred between the Stop and the preceding Start (note that multiple Repeated Starts may have happened in between the Start and Stop). If a write transfer only modified the EZ address, and not the EZ memory, this event is NOT generated.

Return Value: None

Side Effects: None

uint32 SCB_GetI2CExtClkInterruptMode(void)

Description: Returns the externally clocked I2C interrupt mask

Parameters: None

Return Value: uint32: Mask of enabled externally clocked I2C interrupt sources (refer to SCB_SetI2CExtClkInterruptMode() function for return values)

Side Effects: None

uint32 SCB_GetI2CExtClkInterruptSourceMasked(void)

Description: Returns the externally clocked I2C interrupt request register masked by interrupt mask

Parameters: None

Return Value: uint32: Status only of enabled externally clocked I2C interrupt sources (refer to SCB_SetI2CExtClkInterruptMode() function for return values)

Side Effects: None

uint32 SCB_GetI2CExtClkInterruptSource(void)

Description: Returns the bit-mask of pending externally clocked I2C interrupt sources

Parameters: None

Return Value: uint32: Status of the externally clocked I2C interrupt sources (refer to SCB_SetI2CExtClkInterruptMode() function for return values)

Side Effects: None

void SCB_ClearI2CExtClkInterruptSource(uint32 interruptMask)

Description: Clears the bit-mask of pending externally clocked I2C interrupt sources

Parameters: uint32 interruptMask: Bit-mask of pending externally clocked I2C interrupt sources to clear (refer to SCB_SetI2CExtClkInterruptMode() function for return values)

Return Value: None

Side Effects: None

void SCB_SetI2CExtClkInterrupt(uint32 interruptMask)

Description: Generates interrupt event from bit-mask of externally clocked I2C interrupt sources

Parameters: uint32 interruptMask: Bit-mask of the externally clocked I2C interrupt sources to generate an interrupt event (refer to SCB_SetI2CExtClkInterruptMode() function for return values)

Return Value: None

Side Effects: None

void SCB_SetSpiExtClkInterruptMode(uint32 interruptMask)

Description: Configures which bits of the externally clocked SPI interrupt request register will trigger an interrupt event

Parameters: uint32 interruptMask: Bit-mask of the externally clocked SPI interrupt sources to be enabled.

SPI Ext clock interrupt sources	Description
SCB_INTR_SPI_EC_EZ_WAKE	Wake up request. Active on incoming slave request.
SCB_INTR_SPI_EC_EZ_STOP	SPI slave deselected after any SPI transfer occurred (Stop detect)
SCB_INTR_SPI_EC_EZ_WRITE_STOP	SPI slave deselected after write EZ SPI transfer occurred (Stop detect). If a write transfer only transferred the EZ address, this event will not be generated. This event is an indication that EZ memory location may have changed contents.

Return Value: None

Side Effects: None

uint32 SCB_GetSpiExtClkInterruptMode(void)

Description: Returns the externally clocked SPI interrupt mask

Parameters: None

Return Value: uint32: Mask of enabled externally clocked SPI interrupt sources (refer to SCB_SetSpiExtClkInterruptMode() function for return values)

Side Effects: None

uint32 SCB_GetSpiExtClkInterruptSourceMasked(void)

Description:	Returns the externally clocked SPI interrupt request register masked by interrupt mask
Parameters:	None
Return Value:	uint32: Status only of enabled externally clocked SPI interrupt sources (refer to SCB_SetSpiExtClkInterruptMode() function for return values)
Side Effects:	None

uint32 SCB_GetSpiExtClkInterruptSource(void)

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

void SCB_ClearSpiExtClkInterruptSource(uint32 interruptMask)

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

void SCB_SetSpiExtClkInterrupt(uint32 interruptMask)

Description:	Generates interrupt event from bit-mask of externally clocked SPI interrupt sources
Parameters:	uint32 interruptMask: Bit-mask of the externally clocked SPI interrupt sources to generate an interrupt event (refer to SCB_SetSpiExtClkInterruptMode() function for return values)
Return Value:	None
Side Effects:	None

Bootloader communication Functions

The SCB component only supports I2C bootloader communication. Use the following SCB configuration 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.

For more information about the Bootloader, refer to the “Bootloader System” section of the *System Reference Guide*.

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

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^2C component and disables its interrupt.

Parameters: None

Return Value: None

Side Effects: None

void SCB_CyBtldrCommReset(void)

Description: Sets read and write I^2C 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: 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

Sample Firmware Source Code

PSoC Creator provides many example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples,

open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the “Find Example Project” topic in the PSoC Creator Help for more information.

Interrupt Service Routine

The SCB block has a single interrupt output that is directly connected to the interrupt controller. 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 since only a single interrupt is available for each SCB.

Functional Description

I2C

This component supports I^2C slave, master, multi-master, and multi-master-slave configurations. The following sections provide an overview of how to use the slave, master, and multi-master components.

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

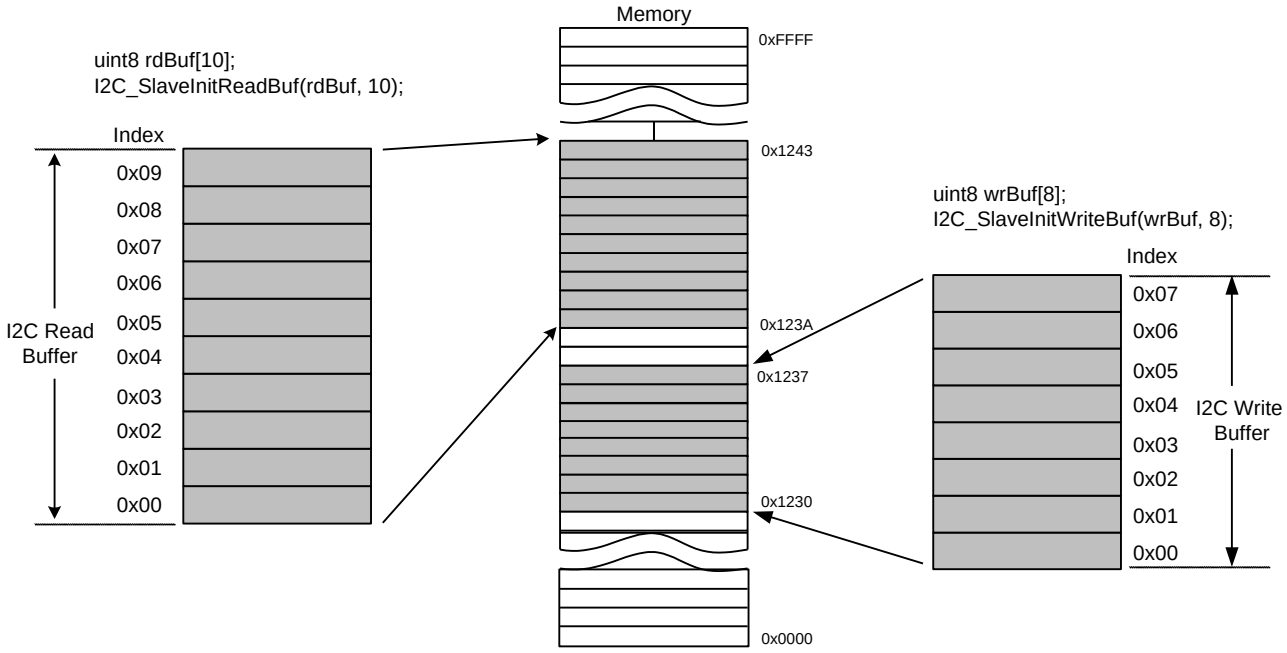
Slave Operation

The slave interface consists of two buffers in memory, one for data written to the slave by a master and a second buffer for data read by a master from the slave. Remember that reads and writes are from the perspective of the I^2C master. The I^2C 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 I2C_SlaveInitReadBuf(uint8 * rdBuf, uint32 bufSize)
void I2C_SlaveInitWriteBuf(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 I2C_SlaveInitReadBuf() or I2C_SlaveInitWriteBuf() 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^{2}C master, the index is incremented until the offset is one less than the byteCount. At any time the number of bytes transferred may be queried by calling either I2C_SlaveGetReadBufSize() or I2C_SlaveGetWriteBufSize() for the read and write buffers, respectively. Reading or writing more bytes than are in the buffers causes an overflow error. The error is set in the slave status byte and may be read with the I2C_SlaveStatus() API.

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

```
void I2C_SlaveClearReadBuf(void)
void I2C_SlaveClearWriteBuf(void)
```

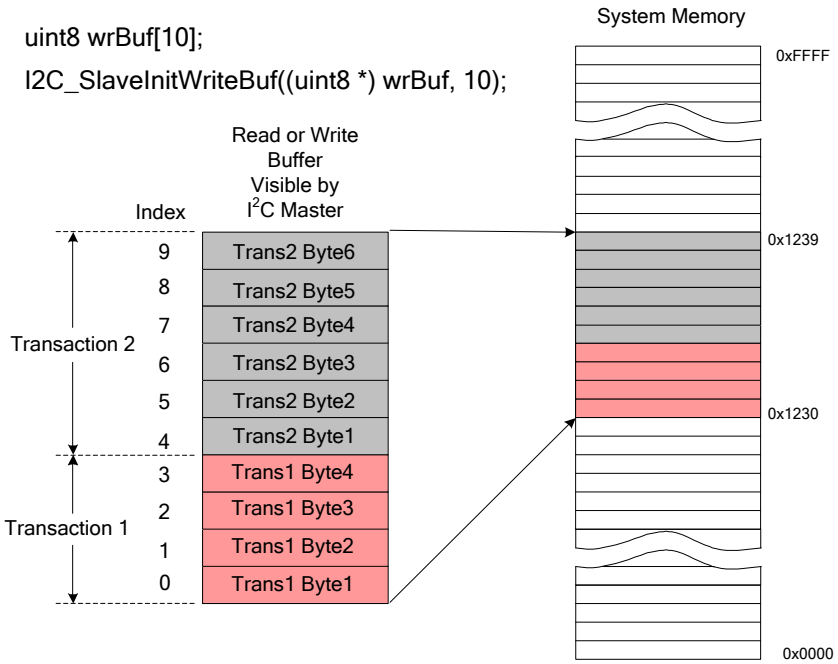
This resets the index back to zero. The next byte read or written to by the I^{2}C master is the first byte in the array. Before these clear buffer commands are used, the data in the arrays should be read or updated.

Multiple reads or writes by the I^{2}C 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^{2}C 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 I2C_SlaveClearWriteBuf() 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	Value	Description
I2C_SSTAT_RD_CMPT	0x01	Slave read transfer complete
I2C_SSTAT_RD_BUSY	0x02	Slave read transfer in progress (busy)
I2C_SSTAT_RD_OVFL	0x04	Master attempted to read more bytes than are in buffer
I2C_SSTAT_WR_CMPT	0x10	Slave write transfer complete
I2C_SSTAT_WR_BUSY	0x20	Slave Write transfer in progress (busy)
I2C_SSTAT_WR_OVFL	0x40	Master attempted to write past end of buffer

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 I2C_Start */
I2C_SlaveInitWriteBuf((uint8 *) wrBuf, 10);

/* Start I2C Slave operation */
I2C_Start();

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

for(;;)  /* loop forever  */
{
   /* Wait for I2C master to complete a write */
   if(0u != (I2C_SlaveStatus() & I2C_SSTAT_WR_CMPT))
   {
      byteCnt = I2C_SlaveGetWriteBufSize();
      I2C_SlaveClearWriteStatus();
      for(i=0; i < byteCnt; i++)
      {
         userArray[i] = wrBuf[i]; /* Transfer data */
      }
      I2C_SlaveClearWriteBuf();
   }
}
```

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 error 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 I2C_MasterWriteBuf(uint32 slaveAddress, uint8 * xferData, 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 I2C_MasterReadBuf(uint32 slaveAddress, uint8 * xferData, uint32 cnt, uint32
mode);
```

Both of these functions return status. See the status table for the I2C_MasterStatus() function return value. Since the read and write transfers complete in the background during the I^2C interrupt code, the I2C_MasterStatus() function can be used to determine when the transfer is complete. A code snippet that shows a typical write to a slave follows.

```
I2C_MasterClearStatus(); /* Clear any previous status */
I2C_MasterWriteBuf(4, (uint8 *) wrData, 10, I2C_MODE_COMPLETE_XFER);
for(;;)
{
   if(0u != (I2C_MasterStatus() & I2C_MSTAT_WR_CMPLT))
   {

      /* Transfer complete. Check Master status to make sure that transfer
         completed without errors. */

      break;
   }
}
```

The I^2C master can also be operated manually. In this mode, each part of the write transaction is performed with individual commands.

```
status = I2C_MasterSendStart(4, I2C_WRITE_XFER_MODE);
if(status == I2C_MSTR_NO_ERROR)   /* Check if transfer completed without errors */
{
   /* Send array of 5 bytes */
   for(i=0; i<5; i++)
   {
      status = I2C_MasterWriteByte(userArray[i]);
      if(status != I2C_MSTR_NO_ERROR)
      {
         break;
      }
   }
}
I2C_MasterSendStop();      /* 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 = I2C_MasterSendStart(4, I2C_READ_XFER_MODE);
```

```
if(status == I2C_MSTR_NO_ERROR)   /* Check if transfer completed without errors */
{
   /* Read array of 5 bytes */
   for(i=0; i<5; i++)
   {
      if(i < 4)
      {
         userArray[i] = I2C_MasterReadByte(I2C_ACK_DATA);
      }
      else
      {
         userArray[i] = I2C_MasterReadByte(I2C_NAK_DATA);
      }
   }
}
I2C_MasterSendStop();      /* 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 hardware reverts to Slave mode and the received byte generates a slave address interrupt.

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

Arbitrage on address byte limitations with hardware address match enabled: When a master loses arbitration during an address byte, the slave address interrupt is only generated if slave is addressed. In other cases, the lost arbitrage status is lost by interrupt-based functions. The software address detect eliminates this possibility, but excludes the Wakeup on Hardware Address Match feature.

The manual function I2C_MasterSendStart() provides correct status information in the case described above.

Start of Multi-Master-Slave Transfer

When using Multi-Master-Slave, the Slave can be addressed at any time. The Multi-Master must take time to prepare to generate a start condition when the bus is free. During this time, the Slave could be addressed and, in this case, the Multi-Master transaction is lost and Slave operation proceeds. Be careful not to break the Slave operation; the I^2C interrupt must be disabled before generating a start condition to prevent the transaction from passing the address stage. This action allows you to abort a Multi-Master transaction and start Slave operation correctly. The following cases are possible when disabling the I^2C interrupt:

- The bus is busy (Slave operation is in progress or other traffic is on the bus) before start generation. The Multi-Master does not try to generate a start condition. Slave operation proceeds when the I^2C interrupt is enabled. The I2C_MasterWriteBuf(),

I2C_MasterReadBuf(), or I2C_MasterSendStart() call returns the status **I2C_MSTR_BUS_BUSY**.

- The bus is free before start generation. The Multi-Master generates a start condition on the bus and proceeds with operation when I^2C interrupt is enabled. The I2C_MasterWriteBuf(), I2C_MasterReadBuf(), or I2C_MasterSendStart() call returns the status **I2C_MSTR_NO_ERROR**.
- The bus is free before start generation. The Multi-Master tries to generate a start but another Multi-Master addresses the Slave before this and the bus becomes busy. The start condition generation is queued. The Slave operation stops at the address stage because of a disabled I^2C interrupt. When I^2C interrupt is enabled, the Multi-Master transaction is aborted from queue and Slave operation proceeds. The I2C_MasterWriteBuf() or I2C_MasterReadBuf() call does not notice this and returns **I2C_MSTR_NO_ERROR**. The I2C_MasterStatus() returns **I2C_MSTAT_WR_CMPLT** or **I2C_MSTAT_RD_CMPLT** with **I2C_MSTAT_ERR_XFER** (all other error condition bits are cleared) after the Multi-Master transaction is aborted. The I2C_MasterSendStart() call returns the error status **I2C_MSTR_ABORT_XFER**.

Interrupt Function Operation

- I2C_MasterWriteBuf();
- I2C_MasterReadBuf();

```
I2C_MasterClearStatus();    /* Clear any previous status */
I2C_DisableInt();           /* Disable interrupt */

status = I2C_MasterWriteBuf(4, (uint8 *) wrData, 10, I2C_MODE_COMPLETE_XFER);
/* Try to generate, start. The disabled I2C interrupt halt the transaction on
address stage in case of Slave addressed or Master generates start condition */

I2C_EnableInt();            /* Enable interrupt and proceed Master or Slave
transaction */

for(;;)
{
   if(0u != (I2C_MasterStatus() & I2C_MSTAT_WR_CMPLT))
   {
     /* Transfer complete. Check Master status to make sure that transfer
        completed without errors. */
      break;
   }
}

if (0u != (I2C_MasterStatus() & I2C_MSTAT_ERR_XFER))
{
      /* Error occurred while transfer, clean up Master status and
         retry the transfer */
}
```

Manual Function Operation

Manual Multi-Master operation assumes that I²C interrupt is disabled, but it is best to take the following precaution:

```
I2C_DisableInt();        /* Disable interrupt */
status = I2C_MasterSendStart(4, I2C_WRITE_XFER_MODE);;    /* Try to generate start
condition */
if (status == I2C_MSTR_NO_ERROR)   /* Check if start generation completed without
errors */
{
    /* Proceed the write operation */
   /* Send array of 5 bytes */
   for(i=0; i<5; i++)
   {
      status = I2C_MasterWriteByte(userArray[i]);
      if(status != I2C_MSTR_NO_ERROR)
      {
         break;
      }
   }
    I2C_MasterSendStop();      /* Send Stop */
}
I2C_EnableInt();         /* Enable interrupt, if it was enabled before */
```

Wakeup on Hardware Address Match

The wakeup from sleep on I²C address match event is possible if the following demands are met:

- The I²C slave should be enabled. Slave or Multi-Master-Slave mode is selected.
- Enable wakeup from Sleep Mode is checked.

How it Works

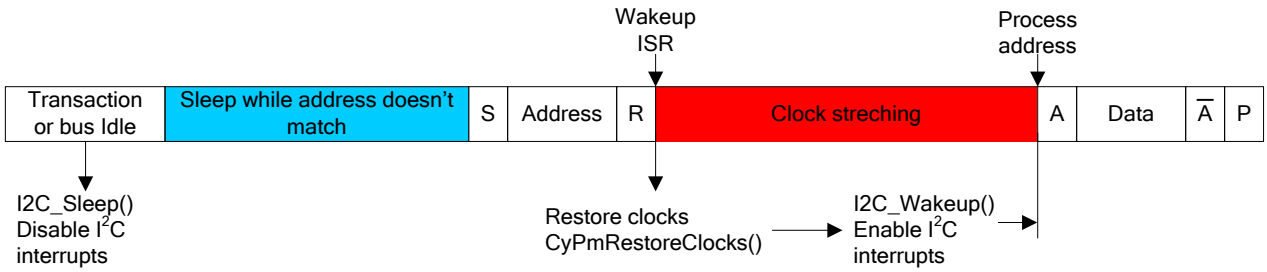
The I²C block responds to transactions on the I²C bus during sleep mode. The I2C wakes the system if the incoming address matches with the slave address. Once the address matches, wakeup interrupt is asserted to wake up the system and SCL is pulled low. The ACK is sent out after the system wakes up and the CPU determines the next action in the transaction.

Wakeup and Clock Stretching

The I²C slave stretches the clock while exiting sleep mode. All clocks in the system must be restored before continuing the I²C transactions. The I²C interrupt is disabled before going to sleep and only enabled after the I2C_Wakeup() function is called. The time between wakeup and end of calling I2C_Wakeup(), SCL line is pulled low.

Sample code:

```
…
TBD
…
```



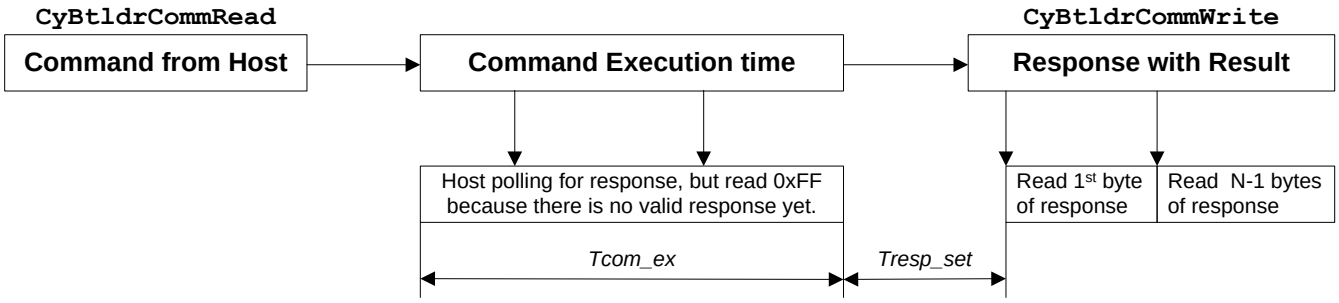
I²C interface Bootloader Communication

The bootloader protocol is implemented as command (write transaction) and response (read transaction).

The time between the Host issuing the command and the bootloader sending back the response is the command execution time. The I²C communication component for the bootloader is designed in this way: when the Host asks for a response, and bootloader still executes command, 0xFF is returned.

Startup: The I²C bootloader communication component expects to receive the command and does not have a valid response yet. All read transactions from the Host return 0xFF. All write transactions are treated as command.

Bootloader process: The Host is issued the command with single write transaction and starts polling for a response. The I²C communication component answers with 0xFF until a valid response is passed by the bootloader. After receiving 0x01, the Host must perform another read to get the remaining N – 1 bytes of the response. Once both reads are complete, the results should be combined to form the full response packet.

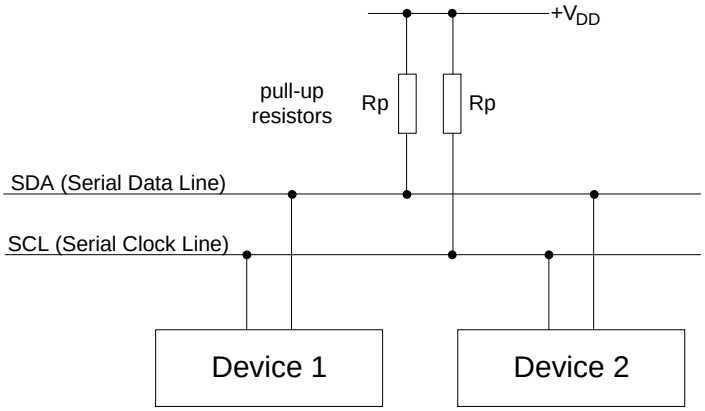


Polling must be executed by reading one byte; reading more bytes could corrupt the response. For example, 0xFF 0x01 0x03 (two bytes of response were read, instead of one). The next read of the full response returns two invalid bytes, because these bytes were already read (0x01 and 0x03).

External Electrical Connections

As Figure 3 shows, the I²C 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, see *The I²C-Bus Specification* on the NXP web site at www.nxp.com.

Figure 3. Connection of Devices to the I²C Bus



Note Purchase of I²C components from Cypress or one of its sublicensed Associated Companies, conveys a license under the Philips I²C Patent Rights to use these components in an I²C system, provided that the system conforms to the I²C Standard Specification as defined by Philips. As of October 1, 2006, Philips Semiconductors has a new trade name - NXP Semiconductors.

DC and AC Electrical Characteristics

The following values indicate expected performance and are based on initial characterization data.

I²C DC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
	Block current consumption	TBD	--	--	x	µA
		TBD	--	--	x	µA
		TBD	--	--	x	µA
		Wake from sleep mode	--	--	x	µA

I²C AC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
F_{clk}	Clock Frequency	Fixed Function Implementation			48	MHz

Component Changes

This section lists the major changes in the component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
1.0		The first release of the SCB component