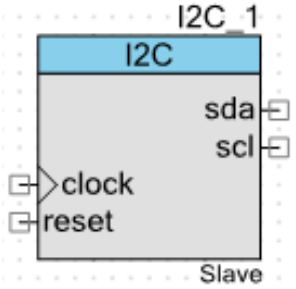
I²C Master/Multi-Master/Slave
2.10

Features



- Industry standard Philips® I²C bus interface
- Supports Slave, Master, Multi-Master and Multi-Master-Slave operation
- Only two pins (SDA and SCL) required to interface to I²C bus
- Standard data rates of 100/400/1000 kbps supported
- High level APIs require minimal user programming

General Description

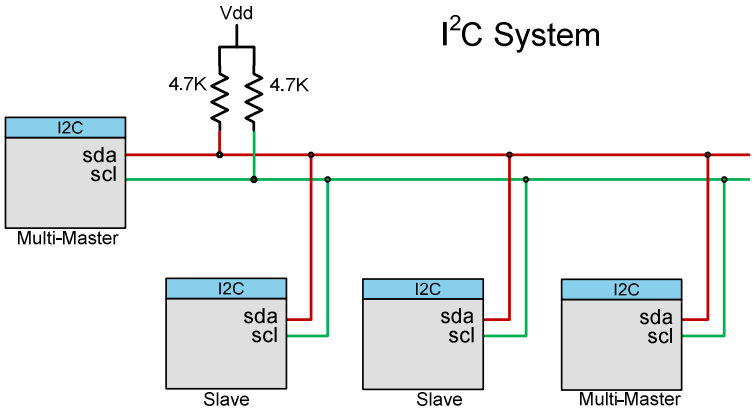
The I²C component supports I²C Slave, Master, and Multi-Master configurations. The I²C bus is an industry standard, two-wire hardware interface developed by Philips. The master initiates all communication on the I²C bus and supplies the clock for all slave devices.

The I²C component supports standard clock speeds up to 1000 kbps. The I²C component is compatible with other third party slave and master devices.

Note This version of the component datasheet covers both the fixed hardware I²C block and the UDB version.

When to Use an I²C Component

The I²C component is an ideal solution when networking multiple devices on a single board or small system. The system can be designed with a single master and multiple slaves, multiple masters, or a combination of masters and slaves.



Input/Output Connections

This section describes the various input and output connections for the I²C 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.

sda – In/Out

Serial data (SDA) is the I²C data signal. It is a bidirectional data signal used to transmit or receive all bus data. The pin connected to **sda** should be configured as Open-Drain-Drives-Low.

scl – In/Out

Serial clock (SCL) is the master generated I²C clock. Although the slave never generates the clock signal, it may hold the clock low stalling the bus until it is ready to send data or ACK/NAK the latest data or address. The pin connected to **scl** should be configured as Open-Drain-Drives-Low.

clock – Input *

The clock input is available when the **Implementation** parameter is set to **UDB**. The UDB version needs a clock to provide 16 times oversampling.

Bus	Clock
50 kbps	800 kHz
100 kbps	1.6 MHz
400 kbps	6.4 MHz
1000 kbps	16 MHz

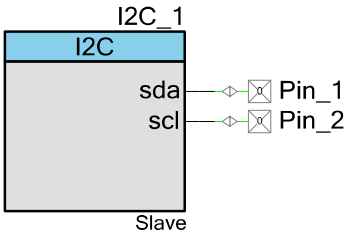
reset – Input *

The reset input is available when the ‘Implementation’ parameter is set to UDB. If the reset pin is held to logic high, the I²C block will be held in reset, and communication over I²C stop. This is a hardware reset only. Software must be independently reset by using the I2C_Stop() and I2C_Start() APIs.

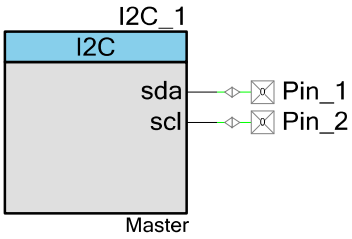
Schematic Macro Information

By default, the PSoC Creator Component Catalog contains four Schematic Macro implementations for the I²C component. These macros contain already connected and configured pins and provide a clock source, as needed. The Schematic Macros use I²C Slave and Master components, configured for fixed function and UDB hardware, as shown below.

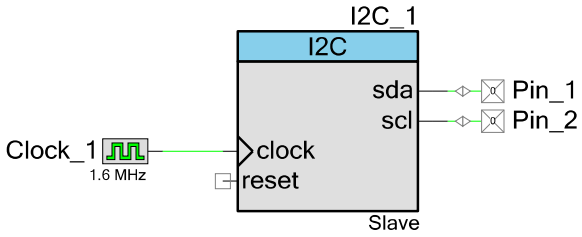
Fixed Function I²C Slave with Pins



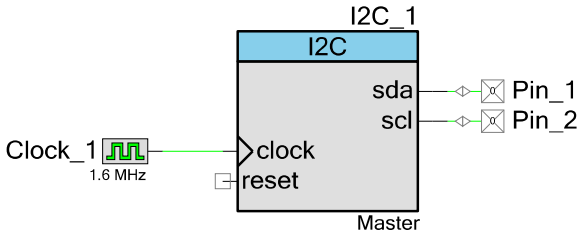
Fixed Function I²C Master Pins



UDB I²C Slave with Clock and Pins

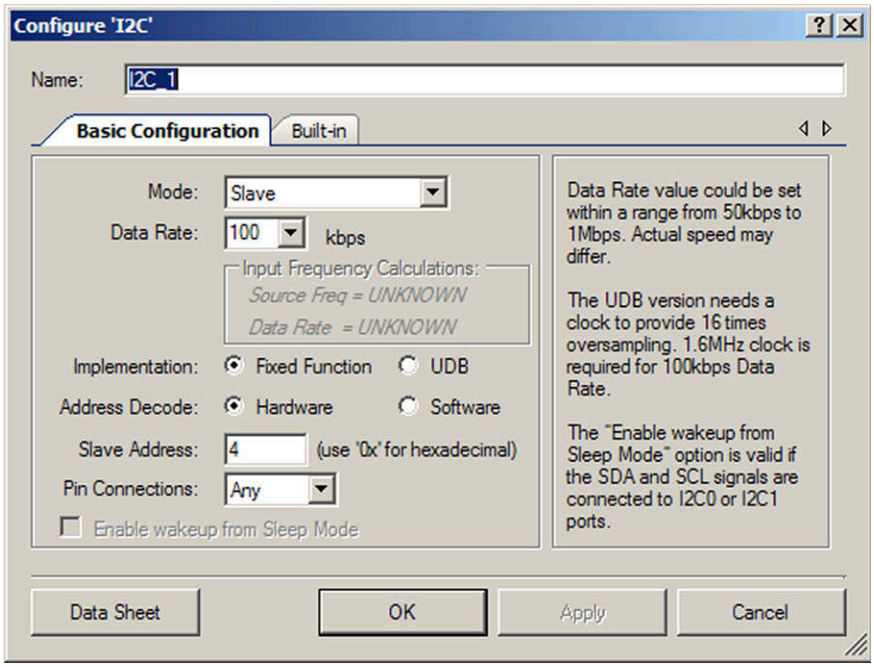


UDB I²C Master with Clock and Pins



Parameters and Setup

Drag an I²C component onto your design and double-click it to open the Configure dialog.



The I²C component provides the following parameters.

Mode

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

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 I²C data rate value up to 1000 kbps; the actual speed may differ based on available clock speed and divider range. The standard speeds are 50, 100 (default), 400, and 1000 kbps. If **Implementation** is set to **UDB**, the **Data Rate** parameter is ignored; the 16x input clock determines the data rate.

Implementation

This option determines how the I²C hardware is implemented on the device.

Implementation	Description
FixedFunction	Use the fixed function block on the device (default).
UDB	Implement the I²C in the UDB array.

Address Decode

This parameter gives you the option to choose between software and hardware address decoding. For most applications where the provided APIs are sufficient and only one slave address is required, "Hardware" address decoding is preferred. In applications where you prefer to modify the source code to provide detection of multiple slave addresses, "Software" address detection is required. Hardware is the default. If hardware address decode is enabled, the block will automatically NAK addresses that are not its own without CPU intervention. It will automatically interrupt the CPU on correct address reception, and hold the SCL line low until CPU intervention.

Slave Address

This is the I²C 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 (0x00 and 0x7F) may be selected; the default is 4. 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. If a 10-bit slave address is required, the designer must use software address decoding and provide decode support for the second byte of the 10-bit address in the ISR.

Pin Connections

This parameter determines which type of pins to use for SDA and SCL signal connections. There are three possible values: **Any**, **I2C0**, and **I2C1**. The default is **Any**.

Any means general purpose I/O (GPIO or SIO). If **Enable wakeup from Sleep Mode** is not required, **Any** should be used for SDA and SCL. If **Enable wakeup from Sleep Mode** is required, **I2C0** or **I2C1** must be used; using either **I2C0** or **I2C1** will allow you to configure the device for wakeup on I²C address match.

The I²C component does not check the correct pins assignment.

Value	Pins
Any	Any GPIO or SIO pins through schematic routing
I2C0	SCL=SIO pin P12[0], SDA=SIO pin P12[1]
I2C1	SCL=SIO pin P12[4], SDA=SIO pin P12[5]

Enable wakeup from Sleep Mode

This option enables the system to be awakened from sleep when an address match occurs. This option is only valid if **Hardware Address Decode** is selected and the SDA and SCL signals are connected to SIO pins (I**2C0** or **I2C1**). The default is disabled.

The possibility of I²C to wake up device on slave address match should be enabled while switching to the sleep mode, this can be done by calling the I2C_Sleep() API; also refer to the “Wakeup on hardware address match” section and to the "Power Management APIs" section of the *System Reference Guide*.

Clock Selection

The UDB implementation requires an external clock source to provide 16 times oversampling. For example, a 1.6 MHz clock is required for 100 kHz data rate.

The Fixed Function block use BUS_CLK and calculated by customizer divider to archive the 16/32 oversampling rate (50kHz oversampling rate is 32, for all others is 16).

Note
Look at Errata Item 49. I²C Clocking to provide desired clock for I²C Fixed Function block on early silicon versions.

Resources

The following configuration settings were used to generate the resource usage information: (1) Address detection set to hardware; (2) Enable wakeup from sleep set to false. The functions I2C_SlavePutReadByte() and I2C_SlaveGetWriteByte() cannot gather the bus status information necessary to be implemented correctly. These two functions do not work and should not be used in designs

The fixed I²C block is used for the Fixed Function implementation.

Mode	Digital Blocks					API Memory (Bytes)		Pins
	Datapaths	Macro cells	Status Registers	Control Registers	Counter7	Flash	RAM	
Slave	N/A	N/A	N/A	N/A	N/A	857	21	2
Master	N/A	N/A	N/A	N/A	N/A	1714	20	2
Multi-Master	N/A	N/A	N/A	N/A	N/A	1884	20	2
Multi-Master-Slave	N/A	N/A	N/A	N/A	N/A	2476	33	2

For UDB implementation, see the following table.

Mode	Digital Blocks					API Memory (Bytes)		Pins
	Datapaths	Macro cells	Status Registers	Control Registers	Counter7	Flash	RAM	
Slave	1	25	1	1	1	861	16	2
Master	2	16	1	1	0	1786	16	2

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 "I2C_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 "I2C".

All API functions assume that data direction is from the perspective of the I²C master. A write event occurs when data is written from the master to the slave. A read event occurs when the master reads data from the slave.

Generic Functions

This section includes the functions that are generic to I²C slave or master operation.

Function	Description
void I2C_Start(void)	I²C component is initialized and enabled. The component interrupt must be enabled to start responding to I²C traffic (I²C Interrupt remains disabled).
void I2C_Stop(void)	Stops responding to I²C traffic (Disables interrupt).
void I2C_EnableInt(void)	Enables interrupt, which is required for most I²C operations.
void I2C_DisableInt(void)	Disables interrupt. The I2C_Stop() API does this automatically.
void I2C_Sleep(void)	Stops I²C operation and saves I²C non-retention configuration registers (Disables interrupt). Prepares wake on address match operation If Wakeup from Sleep Mode enabled. (Disables I²C Interrupt).
void I2C_Wakeup(void)	Restores I²C non-retention configuration registers and enables I²C operation (Enables I²C Interrupt).
void I2C_SaveConfig(void)	Saves I²C non-retention configuration registers (Disables I²C Interrupt).
void I2C_RestoreConfig(void)	Restores I²C non-retention configuration registers saved by I2C_SaveConfig() or I2C_Sleep() (Enables I²C Interrupt).
void I2C_Init(void)	Initializes I²C registers with initial values provided from customizer.
void I2C_Enable(void)	Activates I²C hardware and begins component operation.

Global Variables

Knowledge of these variables is not required for normal operations.

Variable	Description
I2C_initVar	Indicates whether the I²C has been initialized. The variable is initialized to 0 and set to 1 the first time I2C_Start() is called. This allows the component to restart without reinitialization after the first call to the I2C_Start() routine. If reinitialization of the component is required the variable should be set to 0 before the I2C_Start() routine is called. Alternately, the I2C can be reinitialized by calling the I2C_Init() and I2C_Enable() functions.
I2C_State	Current state of I²C state machine.
I2C_mstrStatus	Current status of I²C master.
I2C_mstrControl	Controls master end of transaction with or without the stop generation.
I2C_mstrRdBufPtr	Pointer to master read buffer.
I2C_mstrRdBufSize	Size of master read buffer.
I2C_mstrRdBufIndex	Current index within master read buffer.
I2C_mstrWrBufPtr	Pointer to master write buffer.
I2C_mstrWrBufSize	Size of master write buffer.
I2C_mstrWrBufIndex	Current index within master write buffer.
I2C_slStatus	Current status of I²C slave.
I2C_Address	Software address of I²C slave.
I2C_readBufPtr	Pointer to slave read buffer.
I2C_readBufSize	Size of slave read buffer.
I2C_readBufIndex	Current index within slave read buffer.
I2C_writeBufPtr	Pointer to slave write buffer.
I2C_writeBufSize	Size of slave write buffer.
I2C_writeBufIndex	Current index within slave write buffer.

void I2C_Start(void)

Description: This is the preferred method to begin component operation. I2C_Start() calls the I2C_Init() function, and then calls the I2C_Enable() function. I2C_Start() must be called before I²C bus operation.
This API does not enable the I²C interrupt; however, interrupts are required for most I²C operations.

Parameters: None

Return Value: None

Side Effects: If the I²C interrupt is disabled while the I²C is still running, it may cause the I²C bus to lock up.

void I2C_Stop(void)

Description: Disables I²C hardware and interrupt.

Parameters: None

Return Value: None

Side Effects: None

void I2C_EnableInt(void)

Description: Enables I²C interrupt. Interrupts are required for most operations. Should be called after I2C_Start() API.

Parameters: None

Return Value: None

Side Effects: None

void I2C_DisableInt(void)

Description: Disables I²C interrupts. Normally this function is not required since the Stop function disables the interrupt.

Parameters: None

Return Value: None

Side Effects: If the I²C interrupt is disabled while the I²C is still running, it may cause the I²C bus to lock up.

void I2C_Sleep(void)

Description: This is the preferred API to prepare the component for sleep. The I²C interrupt is disabled after function call.
Wakeup on address match enabled: This API will wait until all I²C transaction intended to this device will be completed. All subsequent I²C traffic intended to this device will be NAKed until the device is put to sleep. The address match event will wake up the chip.
Wakeup on address match disabled: This API checks current I²C component state, saves it and disables component calling I2C_Stop() if it is currently enabled. Then I2C_SaveConfig() is called to save the I²C non-retention configuration registers.
Call the I2C_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 I2C_Wakeup(void)

Description: This is the preferred API to restore the component to the state when I2C_Sleep() was last called. The I²C interrupt is enabled after function call.
Wakeup on address match enabled: This API enables I²C master functionality if it was enabled before sleep, and disables I²C backup regulator. The incoming transaction continues at the moment when I²C interrupt will be enabled
Wakeup on address match disabled: This API restores the I²C non-retention configuration registers calling I2C_RestoreConfig(). If the component was enabled before the I2C_Sleep() function was called, I2C_Wakeup() will re-enable it.

Parameters: None

Return Value: None

Side Effects: Calling the I2C_Wakeup() function without first calling the I2C_Sleep() or I2C_SaveConfig() function may produce unexpected behavior.

void I2C_Init(void)

Description: Initializes or restores the component according to the customizer Configure dialog settings. It is not necessary to call I2C_Init() because the I2C_Start() API calls this function, which is the preferred method to begin component operation.

Parameters: None

Return Value: None

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

void I2C_Enable(void)

Description: Activates the hardware and begins component operation. It is not necessary to call I2C_Enable() because the I2C_Start() API calls this function, which is the preferred method to begin component operation. If this API is called, I2C_Start(), or I2C_Init() must be called first.

Parameters: None

Return Value: None

Side Effects: None

void I2C_SaveConfig(void)

Description: This function saves the I²C component non-retention configuration registers and disables I²C interrupt
Wakeup on address match enabled: This API disables I²C master, if it was enabled before and enables I²C backup regulator. Waits while on–going transaction will be completed and I²C will be ready go to sleep. All subsequent I²C traffic will be NAKed until the device is put to sleep.
Wakeup on address match disabled: Refer to main description above.
Disabling of I²C interrupt does not depend on wakeup on address match enabled or disabled.

Parameters: None

Return Value: None

Side Effects: None

void I2C_RestoreConfig(void)

Description: This function restores the I²C component non-retention configuration registers, to the state they were before I2C_Sleep() or I2C_SaveConifg() were called. Enables I²C interrupt.
Wakeup on address match enabled: This API enables I²C master functionality if it was enabled before and disables I2C backup regulator.
Wakeup on address match disabled: Refer to main description above.
Enabling of I²C interrupt does not depend on wakeup on address match enabled or disabled.

Parameters: None

Return Value: None

Calling this function without first calling the I2C_Sleep() or I2C_SaveConfig() function may produce unexpected behavior.

Slave Functions

This section lists the functions that are used for I²C slave operation. These functions will be available if slave operation is enabled.

Function	Description
uint8 I2C_SlaveStatus(void)	Returns slave status flags.
uint8 I2C_SlaveClearReadStatus(void)	Returns read status flags and clears slave read status flags.
uint8 I2C_SlaveClearWriteStatus(void)	Returns the write status and clears the slave write status flags.
void I2C_SlaveSetAddress(uint8 address)	Sets slave address, a value between 0 and 127 (0x00-0x7F).
void I2C_SlaveInitReadBuf(uint8 * rdBuf, uint8 byteCount)	Sets up the slave receive data buffer. (master <- slave)

void I2C_SlaveInitWriteBuf(uint8 * wrBuf, uint8 byteCount)	Sets up the slave write buffer. (master -> slave)
uint8 I2C_SlaveGetReadBufSize(void)	Returns the number of bytes read by the master since the buffer was reset.
uint8 I2C_SlaveGetWriteBufSize(void)	Returns the number of bytes written by the master since the buffer was reset.
void I2C_SlaveClearReadBuf(void)	Resets the read buffer counter to zero.
void I2C_SlaveClearWriteBuf(void)	Resets the write buffer counter to zero.
void I2C_SlavePutReadByte (uint8 transmitDataByte)	For Master Read, sends 1 byte out Slave transmit buffer. **This function should not be used due to impossibility of hardware to support its correct operation.**
uint8 I2C_SlaveGetWriteByte (uint8 ackNak)	For a Master Write, ACKs or NAKs the previous byte and reads out the last byte transmitted. **This function should not be used due to impossibility of hardware to support its correct operation.**

uint8 I2C_SlaveStatus(void)

Description: Returns the slave's communication status.

Parameters: None

Return Value: (uint8) Current status of I²C slave.

Slave status constants	Description
I2C_SSTAT_RD_CMPT	Slave read transfer complete, set when master indicates it is done reading by sending a NAK
I2C_SSTAT_RD_BUSY	Slave read transfer in progress, set when master addresses slave with a read, cleared when RD_CMPT set.
I2C_SSTAT_RD_ERR_OVFL	Master attempted to read more bytes than are in buffer.
I2C_SSTAT_WR_CMPT	Slave write transfer complete, Set at reception of a Stop bit, or when WR_ERR_OVFL set
I2C_SSTAT_WR_BUSY	Slave Write transfer in progress. Set when master addresses slave with a write, cleared at reception of a Stop bit, or when WR_ERR_OVFL set
I2C_SSTAT_WR_ERR_OVFL	Master attempted to write past end of buffer.

Side Effects: None

uint8 I2C_SlaveClearReadStatus(void)

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

Parameters: None

Return Value: (uint8) Current read status of slave (See I2C_SlaveStatus function for constants).

Side Effects: None

uint8 I2C_SlaveClearWriteStatus(void)

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

Parameters: None

Return Value: (uint8) Current write status of slave (See I2C_SlaveStatus function for constants).

Side Effects: None

void I2C_SlaveSetAddress(uint8 address)

Description: Sets the I²C slave address

Parameters: (uint8) address: I²C slave address for the primary device. This value may be any address between 0 and 127 (0x00-0x7F). 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 I2C_SlaveInitReadBuf(uint8 * rdBuf, uint8 bufSize)

Description: Sets the buffer pointer and size of the read buffer. This function also resets the transfer count returned with the I2C_SlaveGetReadBufSize function.

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

(uint8) bufSize: – Size of the buffer exposed to the I²C 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 I2C_SlaveInitWriteBuf(uint8 * wrBuf, uint8 bufSize)

Description: Sets the buffer pointer and size of the write buffer. This function also resets the transfer count returned with the I2C_SlaveGetWriteBufSize function.

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

(uint8) bufSize – Size of the write buffer exposed to the I²C 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.

uint8 I2C_SlaveGetReadBufSize(void)

Description: Returns the number of bytes read by the I²C master since an I2C_SlaveInitReadBuf or I2C_SlaveClearReadBuf function was executed.
The maximum return value will be the size of the read buffer.

Parameters: None

Return Value: (uint8) Bytes read by master.

Side Effects: None

uint8 I2C_SlaveGetWriteBufSize(void)

Description: Returns the number of bytes written by the I²C master since an I2C_SlaveInitWriteBuf or I2C_SlaveClearWriteBuf function was executed.
The maximum return value will be the size of the write buffer.

Parameters: None

Return Value: uint8: Bytes written by master.

Side Effects: None

void I2C_SlaveClearReadBuf(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 I2C_SlaveClearWriteBuf(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

void I2C_SlavePutReadByte (uint8 transmitDataByte)

Description: This API can be used instead of I2C_SlaveInitReadBuf(), for those designs that require low level byte by byte control of the I2C read data. However, if low level byte by byte control of the I2C read data is not required the I2C_SlaveInitReadBuf() API should be used. This API places 1 byte of data directly into the I2C slave transmit buffer. This byte will be shifted out on the next I2C master read.

Note: I²C interrupts must be disabled for this API to work properly.

Note: Take care to ensure that the previous byte has shifted out of the transmit buffer before writing another byte via this API.

Note: This API will release the SCL line, if held low, and ACK the previous transfer

Parameters: (uint8) transmitDataByte: Byte containing the data to transmit.

Return Value: None

Side Effects: This function should not be used due to impossibility of hardware to support its correct operation.

uint8 I2C_SlaveGetWriteByte (uint8 ackNak)

Description: This API grabs the latest data written to the I2C transmit buffer by the master. It will also ACK/NAK this data based on the setting of the ackNak parameter. The bus is stalled between SlaveGetWriteByte() calls.

Note: I²C interrupts must be disabled for this API to work properly.

Note: Take care to ensure that new data has completely shifted into the transmit buffer before writing another byte via this API.

Parameters: (uint8) ackNak: 1 = byte was ACKed, 0 = bytes was NAKed for the previous byte received.

Return Value: (uint8) Last byte transmitted or last byte in buffer from Master.

Side Effects: This function should not be used due to impossibility of hardware to support its correct operation.

Master and Multi-Master Functions

These functions are only available if Master or Multi-Master mode is enabled.

Function	Description
uint8 I2C_MasterStatus(void)	Returns master status.
uint8 I2C_MasterClearStatus(void)	Returns the master status and clear the status flags.
uint8 I2C_MasterWriteBuf(uint8 slaveAddr, uint8 * wrData, uint8 cnt, uint8 mode)	Writes the referenced data buffer to a specified slave address.
uint8 I2C_MasterReadBuf(uint8 slaveAddr, uint8 * rdData, uint8 cnt, uint8 mode)	Reads data from the specified slave address and place the data in the referenced buffer.
uint8 I2C_MasterSendStart(uint8 slaveAddress, uint8 R_nW)	Sends just a start to the specific address.
uint8 I2C_MasterSendRestart(uint8 slaveAddress, uint8 R_nW)	Sends just a restart to the specified address.
uint8 I2C_MasterSendStop(void)	Generates a stop condition.
uint8 I2C_MasterSendStart(uint8 slaveAddress, uint8 R_nW)	Sends just a start to the specific address.
uint8 I2C_MasterSendRestart(uint8 slaveAddress, uint8 R_nW)	Sends just a restart to the specified address.
uint8 I2C_MasterSendStop(void)	Generates a stop condition.
uint8 I2C_MasterWriteBuf(uint8 slaveAddr, uint8 * wrData, uint8 cnt, uint8 mode)	Writes the reference data buffer to a specified slave address.
uint8 I2C_MasterReadBuf(uint8 slaveAddr, uint8 * rdData, uint8 cnt, uint8 mode)	Reads data from the specified slave address and place the data in the referenced buffer.
uint8 I2C_MasterWriteByte(uint8 theByte)	Writes a single byte. This is a manual command that should only be used with MasterSendStart or MasterSendRestart functions.
uint8 I2C_MasterReadByte(uint8 acknNack)	Reads a single byte. This is a manual command that should only be used with MasterSendStart or MasterSendRestart functions.
uint8 I2C_MasterGetReadBufSize(void)	Returns the byte count of data read since the MasterClearReadBuf function was called.
uint8 I2C_MasterGetWriteBufSize(void)	Returns the byte count of the data written since the MasterClearWriteBuf function was called.
void I2C_MasterClearReadBuf(void)	Resets the read buffer pointer back to the beginning of the buffer.
void I2C_MasterClearWriteBuf(void)	Resets the write buffer pointer back to the beginning of the buffer.

uint8 I2C_MasterStatus(void)

Description: Returns the master's communication status.

Parameters: None

Return Value: (uint8) Current status of I²C master.

Master status constants	Description
I2C_MSTAT_RD_CMPLT	Read transfer complete
I2C_MSTAT_WR_CMPLT	Write transfer complete
I2C_MSTAT_XFER_INP	Transfer in progress
I2C_MSTAT_XFER_HALT	Transfer has been halted
I2C_MSTAT_ERR_SHORT_XFER	Transfer completed before all bytes transferred.
I2C_MSTAT_ERR_ADDR_NAK	Slave did not acknowledge address
I2C_MSTAT_ERR_ARB_LOST	Master lost arbitration during communications with slave.
I2C_MSTAT_ERR_XFER	Error occurred during transfer

Side Effects: None

uint8 I2C_MasterClearStatus(void)

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

Parameters: None

Return Value: (uint8) Current status of master (See I2C_MasterStatus function for constants).

Side Effects: None

uint8 I2C_MasterWriteBuf(uint8 slaveAddress, uint8 * wrData, uint8 cnt, uint8 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.

Parameters: (uint8) slaveAddress: Right justified 7-bit Slave address. (Valid range 0 to 127)

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

(uint8) cnt: Number of bytes of buffer to send.

(uint8) 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
I2C_MODE_COMPLETE_XFER	Perform complete transfer from Start to Stop.
I2C_MODE_REPEAT_START	Send Repeat Start instead of Start.
I2C_MODE_NO_STOP	Execute transfer without a Stop

Return Value: (uint8) Error Status (See I2C_MasterSendStart() command for constants).

Side Effects: None

uint8 I2C_MasterReadBuf(uint8 slaveAddress, uint8 * rdData, uint8 cnt, uint8 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.

Parameters: (uint8) slaveAddress: Right justified 7-bit Slave address. (Valid range 0 to 127)

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

(uint8) cnt: Number of bytes of buffer to read.

(uint8) 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
I2C_MODE_COMPLETE_XFER	Perform complete transfer for Start to Stop.
I2C_MODE_REPEAT_START	Send Repeat Start instead of Start.
I2C_MODE_NO_STOP	Execute transfer without a Stop

Return Value: uint8: Error Status. (See I2C_MasterSendStart command for constants).

Side Effects: None

uint8 I2C_MasterSendStart(uint8 slaveAddress, uint8 R_nW)

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

Parameters: (uint8) slaveAddress: Right justified 7-bit Slave address. (Valid range 0 to 127)

(uint8) R_nW: Zero, send write command, non-zero send read command.

Return Value: (uint8) Error Status.

Mode Constants	Description
I2C_MSTR_NO_ERROR	Function complete without error.
I2C_MSTR_BUS_BUSY	Bus is busy occurred, Start condition generation not started.
I2C_MSTR_SLAVE_BUSY	Slave operation in progress.
I2C_MSTR_ERR_LB_NAK	Last Byte was NAKed.
I2C_MSTR_ERR_ARB_LOST	Master lost arbitration while Start generation.

Side Effects: This function is blocking and doesn't exit until Byte Complete bit is set in the I2C_CSR register.

uint8 I2C_MasterSendRestart(uint8 slaveAddress, uint8 R_nW)

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

Parameters: (uint8) slaveAddress: Right justified 7-bit Slave address (Valid range 0 to 127).

(uint8) R_nW: Zero, send write command, non-zero send read command.

Return Value: (uint8) Error Status (See I2C_MasterSendStart() function for constants).

Side Effects: This function is entered without a 'byte complete' flag set in the I2C_CSR register, it does not exit until it is set.

uint8 I2C_MasterSendStop(void)

Description: Generates I²C Stop condition on bus. This function does nothing if start or restart conditions failed before this function was called.

Parameters: None

Return Value: (uint8) Error Status (See I2C_MasterSendStart() command for constants).

Side Effects: **Master:** This function does not wait while stop condition is generated.
Multi-Master, Multi-Master-Slave: This function waits while Stop condition is generated or arbitrage is lost on ACK/NACK bit.

uint8 I2C_MasterWriteByte(uint8 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: (uint8) theByte: The data byte to send to the slave.

Return Value: (uint8) Error Status.

Mode Constants	Description
I2C_MSTR_NO_ERROR	Function complete without error.
I2C_MSTR_SLAVE_BUSY	Master is not valid master on the bus.
I2C_MSTR_ERR_LB_NAK	Last Byte was NAKed.

Side Effects: This function is blocking and doesn't exit until the Byte Complete bit is set in the I2C_CSR register.

uint8 I2C_MasterReadByte(uint8 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: (uint8) acknNack– Zero, sends a NAK, if non-zero send a ACK.

Return Value: (uint8) Byte read from the slave

Side Effects: This function is blocking and doesn't exit until the Byte Complete bit is set in the I2C_CSR register

uint8 I2C_MasterGetReadBufSize(void)

Description: Returns the number of bytes that has been transferred with an I2C_MasterReadBuf command.

Parameters: None

Return Value: (uint8) Byte count of transfer. If the transfer is not yet complete, it will return the byte count transferred so far.

Side Effects: None

uint8 I2C_MasterGetWriteBufSize(void)

Description: Returns the number of bytes that has been transferred with an I2C_MasterWriteBuf command.

Parameters: None

Return Value: (uint8) Byte count of transfer. If the transfer is not yet complete, it will return the byte count transferred so far.

Side Effects: None

void I2C_MasterClearReadBufSize(void)

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

Parameters: None

Return Value: None

Side Effects: None

void I2C_MasterClearWriteBufSize(void)

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

Parameters: None

Return Value: None

Side Effects: None

Multi-Master-Slave Functions

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

Bootloader Support

The I²C component could be used as a communication component for the Bootloader. The following configuration should be used to support communication protocol from an external system to the Bootloader:

- Mode: slave
- Implementation: fixed function
- Data rate: required to match with Host (boot device) data rate.
- Slave address: required to match with Host (boot device) selected slave address.
- Address Match: doesn't care (it is preferred to use hardware)
- Pins connection: doesn't care

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

For additional information about I²C communication component implementation, refer to "Bootloader protocol interaction with I²C communication component" section.

The I²C Component provides a set of API functions for the Bootloader usage.

Function	Description
I2C_CyBtldrCommStart	Starts the I²C component and enables it interrupt
I2C_CyBtldrCommStop	Disable the I²C component and disables its interrupt.
I2C_CyBtldrCommReset	Set read and write I²C buffers to the initial state and resets the slave status.
I2C_CyBtldrCommWrite	Allows the caller to write data to the boot loader host. The function will handle polling to allow a block of data to be completely sent to the host device.
I2C_CyBtldrCommRead	Allows the caller to read data from the boot loader host. The function will handle polling to allow a block of data to be completely received from the host device.

void I2C_CyBtldrCommStart(void)

Description: Starts the I²C component and enables it interrupt. Every incoming I²C write transaction will be treated as a command for the bootloader. Every incoming I²C read transaction will be answered with 0xFF till valid bootloader command will be issued.

Parameters: None

Return Value: None

Side Effects: None

void I2C_CyBtldrCommStop(void)

Description: Disable the I²C component and disables its interrupt.

Parameters: None

Return Value: None

Side Effects: None

void I2C_CyBtldrCommReset (void)

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

Parameters: None

Return Value: None

Side Effects: None

cystatus I2C_CyBtldrCommRead(uint8 * Data, uint16 Size, uint16 * Count, uint8 TimeOut)

Description: Allows the caller to read data from the boot loader host. The function will handle polling to allow a block of data to be completely received from the host device.

Parameters: (uint8) *data: A pointer to the block of data to send to the device.
(uint16) size: The 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 1 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 I2C_CyBtldrCommWrite(uint8 * Data, uint16 Size, uint16 * Count, uint8 TimeOut)

Description:	Allows the caller to write data to the boot loader host. The function will handle polling to allow a block of data to be completely sent to the host device.
Parameters:	(uint8) *data: A pointer to the block of data to send to the device. (uint16) size: The number of bytes to write. (uint16) *count: Pointer to variable to write the number of bytes actually written. (uint8) timeout: Number of units in 10mS to wait before returning because of a timeout.
Return Value:	(cystatus): Returns 1 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 numerous 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.

Functional Description

This component supports I²C slave, master, multi-master and multi-master-slave configurations. The following sections give an overview of how to use the slave, master, and/multi-master components.

This component requires that you enable global interrupts since the I²C hardware is interrupt driven. Even though this component requires interrupts, you do not need to add any code to the ISR (Interrupt Service Routine). The component services all interrupts (data transfers) independent of your code. The memory buffers allocated for this interface look like simple dual port memory between your application and the I²C Master/Slave.

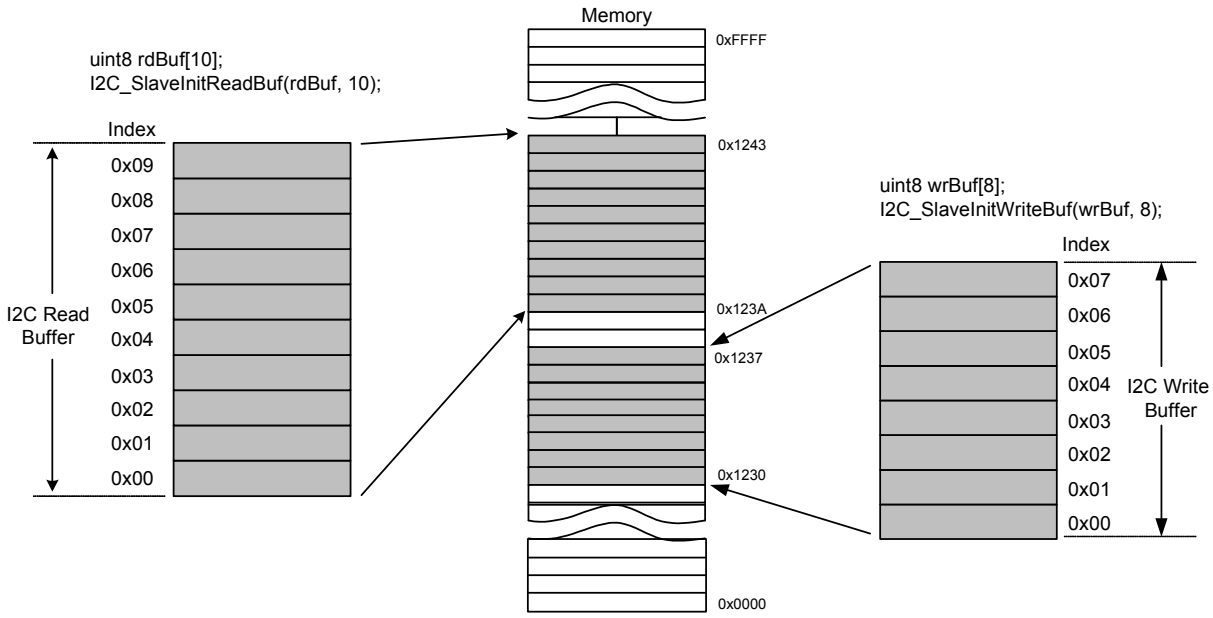
Slave Operation

The slave interface consists of two buffers in memory, one for data written to the slave by a master and a second buffer for data read by a master from the slave. Remember that reads and writes are from the perspective of the I²C Master. The I²C slave read and write buffers are set by the initialization commands below. These commands do not allocate memory, but instead copy the array pointer and size to the internal component variables. The arrays used for the buffers must be instantiated by the designer, since they are not automatically generated by the component. The same buffer may be used for both read and write buffers, but care must be taken to manage the data properly.

```
void I2C_SlaveInitReadBuf(uint8 * rdBuf, uint8 bufSize)
void I2C_SlaveInitWriteBuf(uint8 * wrBuf, uint8 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 they 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 I²C master the index is incremented until the offset is one less than the byteCount. At anytime 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 will cause an overflow error. The error will be 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)
```

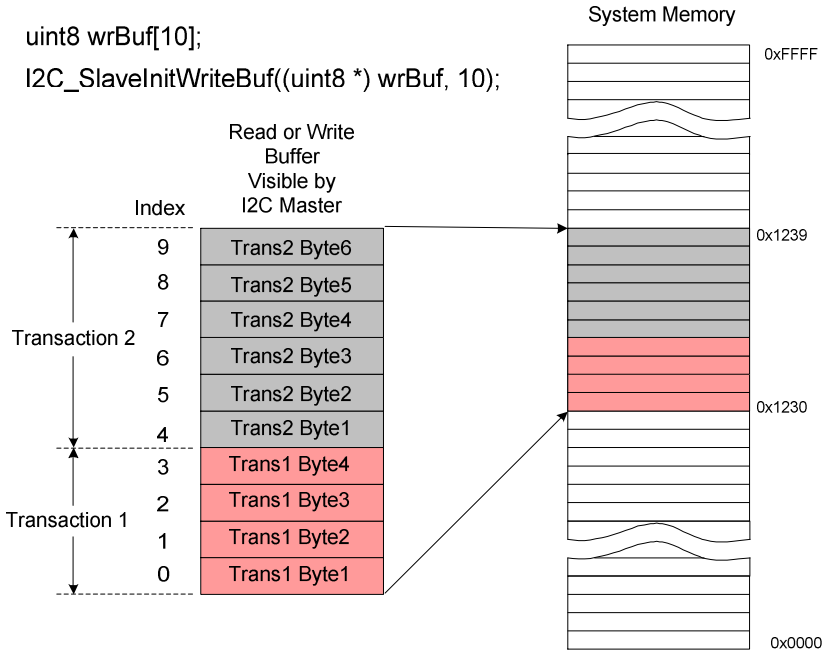
This will reset the index back to zero. The next byte read or written to by the I²C master will be the first byte in the array. Before these clear buffer commands are used, the data in the arrays should be read or updated.

Multiple reads or writes by the I²C master will continue to increment the array index until the clear buffer commands are used or the array index attempts to grow beyond the array size. The figure below shows an example where an I²C master has executed two write transactions. The first write was 4 bytes and the second write was 6 bytes. The 6th byte in the second transaction was

NAKed by the slave to signal that the end of the buffer has occurred. If the master tried to write a 7th 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 will reset the index back to zero and cause the second transaction to overwrite the data from the first transaction. Care should be taken to 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, buffer overflow, and transfer error. 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. See table below for 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. A read buffer example would look almost identical 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];
uint8 byteCnt;
I2C_SlaveInitWriteBuf((uint8 *) wrBuf, 10);
/* Wait for I2C master to complete a write */

for(;;)  /* loop forever  */
{
   /* Wait for I2C master to complete a write */
   if(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 except for two exceptions. When operating in Multi-Master mode, the bus should always be checked to see if it is busy. Another master may be already 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, it is possible that two masters start at the exact same time. If this happens, one of the two masters will lose arbitration. This condition must be checked for after each byte is transferred. The component will automatically check for this condition and respond with an error if arbitration is lost.

There are a couple options when operating the I²C 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 will be pre-filled 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 the automatic mode, use the following function.

```
uint8 I2C_MasterWriteBuf(uint8 SlaveAddr, uint8 * wrData, uint8 cnt, uint8 mode)
```

The SlaveAddr variable is a right justified 7-bit slave address of 0 to 127. The component API will automatically append the write flag to the msb of the address byte. The array of data to transfer is pointed to with the second parameter "wrData". The "cnt" is the amount 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. See function below.

```
uint8 I2C_MasterReadBuf(uint8 SlaveAddr, uint8 * rdData, uint8 cnt, uint8 mode);
```

Both of these functions return status. See the status table for the MasterStatus() function return value. Since the read and write transfers complete in the background during the I²C interrupt code, the MasterStatus() function can be used to determine when the transfer is complete. Below is a code snippet that shows a typical write to a slave.

```
I2C_MasterClearStatus(); /* Clear any previous status */
I2C_MasterWriteBuf(4, (uint8 *) wrData, 10, I2C_MODE_COMPLETE_XFER);
for(;;)
{
   if(I2C_MasterStatus() & I2C_MSTAT_WR_CMPLT)
   {
      /* Transfer complete */
      break;
   }
}
```

The I²C master can also be operated in a manual way. In this mode each part of the write transaction is performed with individual commands. See the example code below.

```
I2C_MasterClearStatus();
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.

```
I2C_MasterClearStatus();
```

```
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 will only be generated if slave is addressed. In other cases the lost arbitrage status will be lost by interrupt based functions. The software address detect eliminates this possibility, but excludes the Wakeup on Hardware Address Match feature.

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

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.
- The I²C Hardware address detection is selected.
- The SIO pair is connected to SCL and SDA and the proper pair is selected in the customizer: I2C0 – SCL P12[0], SDA – SDA P12[1] and I2C1 – SCL P12[4], SDA – SDA P12[5].

The following demands are controlled by the I2C component customizer, except correct pins assignment.

How it works

The I²C block will respond 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 will be sent out once the system wakes up and the CPU determines the next action in the transaction.

Wakeup and clock stretching

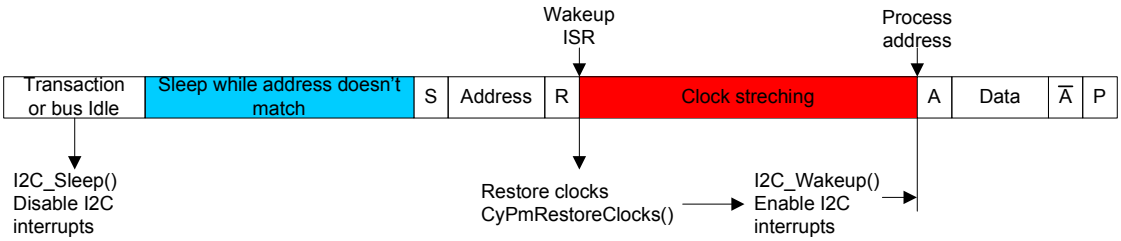
The I²C slave stretches the clock while exiting sleep mode. All clocks in the system need to be restored before continuing the I²C transactions. The I²C interrupt is disabled before going to sleep and only enabled after the I2C_Wakeup() function is called. The time between wakeup and end of calling I2C_Wakeup(),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

Sample code:

```
…
I2C_Sleep();            /* Go to Sleep and disable I2C interrupt */
CyPmSaveClocks();       /* Save clocks settings */

CyPmSleep(PM_SLEEP_TIME_NONE, PM_SLEEP_SRC_I2C);

CyPmRestoreClocks();    /* Restore clocks */
I2C_Wakeup();           /* Wakeup, enable I2C interrupt and ACK the address, till
end of this call the SCL is pulled low */
…
```



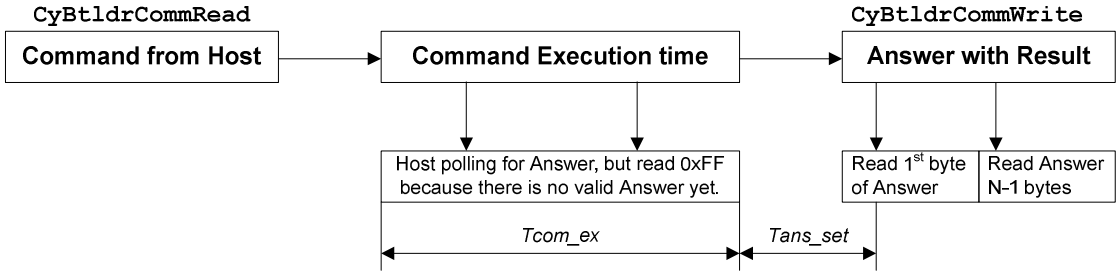
Bootloader protocol interaction with I2C communication component

The bootloader protocol is implemented as Command and Answer.

The time between when the Host issues the Command and the bootloader sends back the Answer is the command execution time. The I2C communication component for bootloader is designed in this way: when the Host asks for Answer and bootloader still executes command, the Answer is 0xFF.

Startup – The I²C bootloader communication component expects to receive the Command and does not have a valid Answer yet. All read transactions from the Host will be answered with 0xFF. All write transactions will be treated as Command.

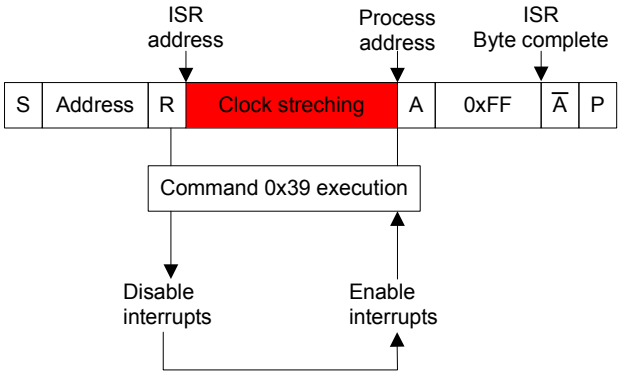
Bootloader process – The Host is issued the Command and starts polling for an Answer. The I2C communication component will answer with 0xFF until a valid Answer is passed by the bootloader. The 0x01 will be the first byte of Answer that the Host reads. The next read will be Answer size – 1.



The polling must be executed by reading one byte; reading more bytes could corrupt the Answer, for example: 0xFF 0x01 0x03 (the two bytes of Answer was read, instead one). The next read of Answer size – 1 will return one invalid byte.

How to avoid polling: The Command execution time (*Tcom_ex*) plus Answer setup time (*Tans_set*) should be measured according to the system settings (CPU speed, compiler, compiler optimization level). The Host needs to ask for Answer after this time. Command execution time changes across the Commands, so the greater time should be chosen.

Clock stretching while polling: The I²C communication component requires interrupts to be enabled while in operation. Command Program Row (0x39), which writes one row of flash data to the device, requires interrupts to be disabled. The clock stretching will occur if the address is accepted by the I2C communication component while interrupts are disabled.



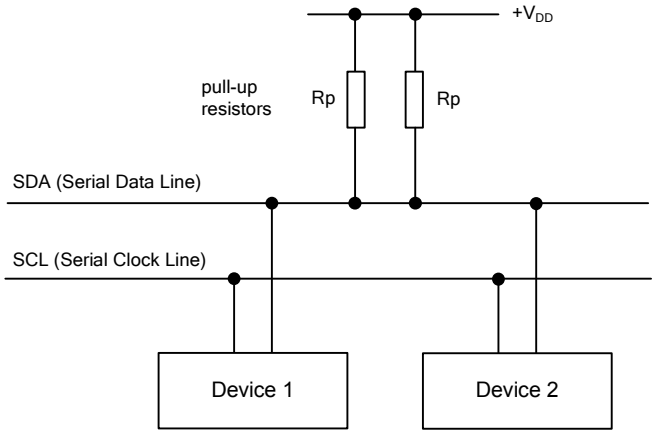
How to avoid clock stretching: To avoid clock stretching, the Command Program Row (0x39) execution time (*Tcom_ex*) should be measured according to the system settings (CPU speed, compiler, compiler optimization level). The Host needs to ask for Answer after this time.

External Electrical Connections

As the block diagram illustrates, the I²C bus requires external pull up resistors. The pull up resistors (RP) 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 VOLmax = 0.4V

for the output stage. This limits the minimum pull up resistor value for a 5V system to about 1.5 kΩ. The maximum value for RP depends upon the bus capacitance and clock speed. For a 5V 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 Philips web site at www.philips.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.

Interrupt Service Routine

The interrupt service routine is used by the component code itself and should not be modified.

Block Diagram and Configuration

Not applicable

Registers

The functions provided support the common runtime functions required for most applications. The following register references provide brief descriptions for the advanced user. The I2C_Data register may be used to write data directly to the bus without using the API. This may be useful for either CPU or DMA use.

The registers available to each of the configurations of the I²C component are grouped according to the implementation as fixed function or UDB.

Fixed Function Master / Slave Registers

Please refer to the chip Technical Reference Manual (TRM) for more information on these registers. All bits which are added in PSoC3 ES3 chip are indicated with an asterisk (*) in the definitions listed below.

I2C_XFCG

The extended configuration register is available in the fixed function hardware block to configure the hardware address mode and clock source.

Bits	7	6	5	4	3	2	1	0
Value	csr_clk_en	i2c_on*	ready_to_sleep*	force_nack*	RSVD			hw_addr_en

- csr_clk_en: Used to enable gating for the fixed function block core logic
- i2c_on*: Used to select I2C block as wake-up source.
- ready_to_sleep*: Used to notify that block is ready to sleep.
- force_nack*: Used to force NACK the transaction.
- hw_addr_en: Used to enable hardware address comparison.

I2C_ADDR

The slave address register is available in the fixed function hardware block to configure the slave device address for hardware comparison mode if enabled in the XCFG register above.

Bits	7	6	5	4	3	2	1	0
Value	RSVD	slave_address						

- slave_address: Used to define the 7-bit slave address for hardware address comparison mode

I2C_CFG

The configuration register is available in the fixed function hardware block to configure the basic functionality.

Bits	7	6	5	4	3	2	1	0
Value	sio_select	pselect	bus_error_ie	stop_ie	clock_rate[1:0]		en_mstr	en_slave

- sio_select: Used to select between SIO1 and SIO2 lines for SCL and SDA, pselect must be set for this bit to have an affect
- pselect: Used to select between SIO direct connections or DSI routed GPIO/SIO pins for SCL and SDA lines
- bus_error_ie: Used to enable interrupt generation for bus_error
- stop_ie: Used to enable interrupt generation on stop bit detection
- clock_rate: Used to select between 16 bit or 32 bit oversample. The PSoC3 ES3 only uses bit2.
- en_mstr: Used to enable master mode
- en_slave: Used to enable slave mode

I2C_CSR

The control and status register is available in the fixed function hardware block for runtime control and status feedback.

Bits	7	6	5	4	3	2	1	0
Value	bus_error	lost_arb*	stop_status	ack	address	transmit	lrb	byte_complete

- bus_error: Bus error detection status bit. This must by cleared by writing a ‘0’ to this bit position.
- lost_arb*: Lost arbitration detection status bit.
- stop_status: Stop detection status bit. This must by cleared by writing a ‘0’ to this position.
- ack: Acknowledge control bit. This bit must be set to ‘1’ to ACK the last byte received or ‘0’ to NACK the last byte received.
- address: Set if the byte just received was an address byte.
- transmit: Used by firmware to define the direction of a byte transfer.
- lrb: Last Received Bit status. This bit indicates the state of the 9th bit (ACK/NACK) response from the receiver for the last byte transmitted.
- byte_complete: Transmit or receive status since last read of this register. In Transmit mode this bit indicates 8-bits of data plus ACK/NAK have been transmitted since last

read. In Receive mode this bit indicates 8-bits of data have been received since last read of this register.

I2C_DATA

The data register is available in the fixed function hardware block for runtime transmit and receipt of data.

Bits	7	6	5	4	3	2	1	0
Value	data							

- data: In Transmit mode this register is written with the data to transmit. In Receive mode this register is read upon status receipt of byte_complete.

I2C_MCSR

The Master control and status register is available in the fixed function hardware block for runtime control and status feedback of Master mode operations.

Bits	7	6	5	4	3	2	1	0
Value	RSVD			stop_gen*	bus_busy	master_mode	restart_gen	start_gen

- stop_gen*: If set, a stop will be generated in master transmitter mode at the end of a byte transfer
- bus_busy: Indicates bus status, 0 means a stop condition was detected, 1 indicates a start condition was detected.
- master_mode: When hardware device is operating as master.
- restart_gen: Control registers to create a restart condition on the bus. This bit is cleared by hardware after the restart has been implemented (may be read as status after setting to poll for completion of the condition).
- start_gen: Control registers to create a start condition on the bus. This bit is cleared by hardware after the start has been implemented (may be read as status after setting to poll for completion of the condition).

UDB Master

The UDB register definitions are derived from the Verilog implementation of I²C. Please refer to the specific mode implementation Verilog for more information on these registers definitions.

I2C_CFG

The control register is available in the UDB implementation for runtime control of the hardware

Bits	7	6	5	4	3	2	1	0
Value	RSVD	stop_gen	restart_gen	nack	RSVD	transmit	en_master	RSVD

- stop_gen: Set to 1 to generate a stop condition on the bus. This bit must be cleared by firmware before initiating the next transaction.
- restart_gen: Set to 1 to generate a restart condition on the bus. This bit must be cleared by firmware after a restart condition is generated.
- nack: Set to 1 to NAK the next read byte. Clear to ACK next read byte. This bit must be cleared by firmware between bytes.
- transmit: Set to 1 to set the current mode to transmit or clear to 0 to receive a byte of data. This bit must be cleared by firmware before starting the next transmit or receive transaction.
- en_master: Set to 1 to enable the master functionality.

I2C_CSR

The status register is available in the UDB implementation for runtime status feedback from the hardware. The status data is registered at the input clock edge of the counter for all bits configured with mode=1, these bits are sticky and are cleared on a read of the status register. All other bits are configured as mode=0 read directly from the inputs to the status register; they are not sticky and therefore not cleared on read. All bits configured as mode=1 are indicated with an asterisk (*) in the definitions listed below.

Bits	7	6	5	4	3	2	1	0
Value		lost_arb*	stop_status*	bus_busy	RSVD	RSVD	lrb	byte_complete

- lost_arb*: If set Indicates arbitration was lost (Multi-Master and Multi-Master-Slave modes).
- stop_status*: If set indicates a stop condition was detected on the bus.
- bus_busy: Indicates the bus is busy if set. Data is currently being transmitted or received.
- lbr: Last Received Bit. Indicates the state of the last received bit which is the ACK/NACK received for the last byte transmitted. Cleared = ACK and set = NACK.

- byte_complete: Transmit or receive status since last read of this register. In Transmit mode this bit indicates 8 bits of data plus ACK/NAK have been transmitted since last read. In Receive mode this bit indicates 8 bits of data have been received since last read of this register.

I2C_INT_MASK

The Interrupt mask register is available in the UDB implementation to configure which status bits are enabled as interrupt sources. Any of the status register bits may be enabled as in interrupt source with a 1-to-1 bit correlation to the status registers bit-field definitions in I2C_CSR above.

I2C_ADDRESS

The slave address register is available in the UDB implementation to configure the slave device address for hardware comparison mode.

Bits	7	6	5	4	3	2	1	0
Value	RSVD	slave_address						

- slave_address: Used to define the 7-bit slave address for hardware address comparison mode

I2C_DATA

The data register is available in the UDB implementation block for runtime transmit and receipt of data.

Bits	7	6	5	4	3	2	1	0
Value	data							

- data: In Transmit mode this register is written with the data to transmit. In Receive mode this register is read upon status receipt of byte_complete.

I2C_GO

The Go register forces data in the data register to be transmitted when master transmits. The Go register forces to receive data in data register when master receives. Any write to this register forces action described above, no matter which value will be written.

UDB Slave

The UDB register definitions are derived from the Verilog implementation of I²C. Please refer to the specific mode implementation Verilog for more information on these registers definitions.

I2C_CFG

The control register is available in the UDB implementation for runtime control of the hardware

Bits	7	6	5	4	3	2	1	0
Value	RSVD	RSVD	RSVD	nack	any_address	transmit	RSVD	en_slave

- nack: If set used to NACK the last byte received. This bit must be cleared by firmware between bytes.
- any_address: If set used to enable the device to respond any device addresses it receives rather than just the single address provided in I2C_ADDRESS.
- transmit: Used to set the mode to transmit or receive data. This bit must be cleared by firmware between bytes. Set = transmit. Cleared = receive.
- en_slave: Set to 1 to enable the slave functionality.

I2C_CSR

The status register is available in the UDB implementation for runtime status feedback from the hardware. The status data is registered at the input clock edge of the counter for all bits configured with Mode=1, these bits are sticky and are cleared on a read of the status register. All other bits are configured as mode=0 and read directly from the inputs to the status register, they are not sticky and therefore not cleared on read. All bits configured as Mode=1 are indicated with an asterisk (*) in the definitions listed below.

Bits	7	6	5	4	3	2	1	0
Value	RSVD	RSVD	stop*	RSVD	address	RSVD	lrb	byte_complete

- stop*: If set Indicates a stop condition was detected on the bus.
- address: Address detection. If set Indicates that an address byte was received.
- lbr: Last Received Bit. Indicates the state of the last received bit which is the ACK/NAK received for the last byte transmitted. Cleared = ACK and set = NAK.
- byte_complete: Transmit or receive status since last read of this register. In Transmit mode this bit indicates 8-bits of data plus ACK/NAK have been transmitted since last read. In Receive mode this bit indicates 8-bits of data have been received since last read of this register.

I2C_INT_MASK

The Interrupt mask register is available in the UDB implementation to configure which status bits are enabled as interrupt sources. Any of the status register bits may be enabled as in interrupt source with a 1-to-1 bit correlation to the status register bit-field definitions in I2C_CSR above.

I2C_ADDRESS

The slave address register is available in the UDB implementation to configure the slave device address for hardware comparison mode.

Bits	7	6	5	4	3	2	1	0
Value	RSVD	slave_address						

- slave_address: Used to define the 7-bit slave address for hardware address comparison mode

I2C_DATA

The data register is available in the UDB implementation block for runtime transmit and receipt of data.

Bits	7	6	5	4	3	2	1	0
Value	data							

- data: In Transmit mode this register is written with the data to transmit. In Receive mode this register is read upon status receipt of byte_complete.

I2C_GO

The Go register forces data in the data register to be transmitted when master transmits. The Go register forces to receive data in data register when master receives. Any write to this register forces action described above, no matter which value will be written.

I2C_TX_DATA

The data register is available in the UDB implementation block for runtime transmit of data. It is defined as the same address as I2C_DATA as they are interchangeable.

Bits	7	6	5	4	3	2	1	0
Value	data							

- data: In Transmit mode this register is written with the data to transmit. In Receive mode this register is read upon status receipt of byte_complete.

References

Not applicable

DC and AC Electrical Characteristics (FF Implementation)

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	Enabled, configured for 100 kbps	--	--	250	µA
		Enabled, configured for 400 kbps	--	--	260	µA
		Wake from sleep mode	--	--	30	µA

I²C AC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
	Bit rate		--	--	1	Mbps

DC and AC Electrical Characteristics (UDB Implementation)

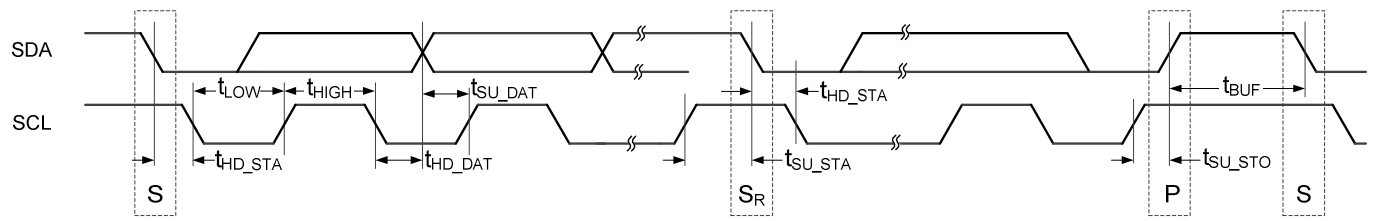
The following values indicate expected performance and are based on initial characterization data.

Timing Characteristics "Maximum with All Routing"

Parameter	Description		Min	Typ	Max	Unit
f_{SCL}	SCL clock frequency					
		Standard mode	–	100	–	kHz
		Fast mode	–	400	–	kHz
		Fast mode plus	–	1000	–	kHz
f_{clock}	Component input clock frequency		–	16 * f_{SCL}	–	kHz
t_{LOW}	Low period of the SCL clock		–	8	–	$t_{CY\ clock}$ [1]
t_{HIGH}	High period of the SCL clock		–	8	–	$t_{CY\ clock}$ [1]
$t_{HD\ STA}$	Hold time (repeated) START condition		–	15	–	$t_{CY\ clock}$
$t_{SU\ STA}$	Setup time for a repeated START condition		–	9	–	$t_{CY\ clock}$
$t_{HD\ DAT}$	Data hold time		–	1	–	$t_{CY\ clock}$
$t_{SU\ DAT}$	Data setup time		–	7	–	$t_{CY\ clock}$
$t_{SU\ STO}$	Setup time for STOP condition		–	9	–	$t_{CY\ clock}$
t_{BUF}	Bus free time between a STOP and START condition		–	32	–	t_{CY_clock}
t_{RESET}	Reset pulse width		–	2	–	$t_{CY\ clock}$

[1] $t_{CY_clock} = 1/f_{clock}$ - Cycle time of one clock period

Figure 4. Data transition timing diagram



How to Use STA Results for Characteristics Data

Nominal route maximums are gathered through multiple test passes with Static Timing Analysis (STA). The maximums may be calculated for your designs using the STA results with the following mechanisms

f_{CLOCK} Maximum Component Clock Frequency is provided in Timing results in the clock summary as the named component clock (CLK in this case). The maximum component clock is limited to f_{clock} = 16 * f_{SCL} = 16 * 1000 kHz = 16 MHz, so STA report must be used only to check that the reported maximum clock frequency (Max Freq) is not violated. An example of the component clock limitations from the *_timing.html* file is below:

+Clock Summary

Clock	Actual Freq	Max Freq	Violation
BUS_CLK	48.000 MHz	112.664 MHz	
Clock	16.000 MHz	20.571 MHz	

The rest parameters are implementation specific and are measured in clock cycles. The I2C component is compatible with I2C-bus specification Rev. 3 from June 2007.

t_{SCL} Defines the I²C data rate value up to 1000 kbps; The standard data rates are 50, 100, 400, and 1000 kbps. The 16x input clock is required to get a needed data rate.

t_{LOW} Low period of the SCL clock. The component generates 50 % duty clock cycle.

t_{HIGH} High period of the SCL clock. The component generates 50 % duty clock cycle.

t_{HD_STA} The minimum amount of time the SCL signal is high after a high to low transition of SDA to generate the start condition. After this period, the first clock pulse is generated.

t_{SU_STA} The minimum amount of time the SCL signal is high before a high to low transition of SDA to generate the start condition.

t_{HD_DAT} The minimum amount of time the data should be valid after the falling edge of the SCL signal.

t_{SU_DAT} The minimum amount of time the data should be valid before the raising edge of the SCL signal.

t_{SU_STO} The minimum amount of time the SCL should be high before a low to high transition of the SDA signal to generate the stop condition.

t_{BUF} The period of time the bus is considered to be free after the STOP condition.

t_{RESET} The component implementation requires two cycles width of the reset signal.

Component Changes

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

Version	Description of Changes	Reason for Changes / Impact
2.10	Add Multi-Master-Slave mode	The support of Multi-Master-Slave functionality is added to component.
	Customizer labels and description edits	Improve feel and content of component customizer.
	I²C bootloader communication component behavior was changed to suppress clock stretching on read.	I²C bootloader communication component holds SCL Low forever if read command issued before start boot process.
	Added characterization data to datasheet.	
	Minor datasheet edits and updates	
2.0.a	Moved component into subfolders of the component catalog	
	Minor datasheet edits and updates	
2.0	Added Sleep/Wakeup and Init/Enable APIs.	To support low power modes, as well as to provide common interfaces to separate control of initialization and enabling of most components.
	Component was updated to support PSoC 3 ES3 and above. Updated the Configure dialog:	New requirements to support the PSoC 3 ES3 device, thus a new 2.0 version was created. Version 1.xx supports PSoC 3 ES2 and PSoC 5 ES1 silicon revisions
	Added configuration of I2C pins connection port for the wakeup on I²C address match feature.	The I²C component will be able to wake up the device from Sleep mode on I²C address match.
	Data sheet was updated.	The Parameters and Setup, Clock Selection, and Resources sections have been updated to reflect the UDB Implementation. Error in sample code has been fixed.

Version	Description of Changes	Reason for Changes / Impact
	Add Reentrancy support to the component.	Allows users to make specific APIs reentrant if reentrancy is desired.

© Cypress Semiconductor Corporation, 2009-2010. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC® is a registered trademark, and PSoC Creator™ and Programmable System-on-Chip™ are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.