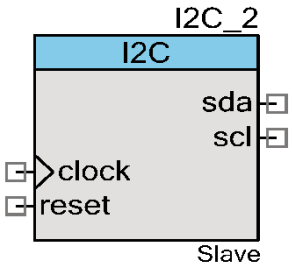
I²C Master/Multi-Master/Slave

2.0

Features



- Industry standard Philips® I²C bus compatible interface
- Supports Slave, Master, and Multi-Master operation
- Only two pins (sda and scl) required to interface to I²C bus
- Standard data rate of 100/400/1000 kbps
- High level API requires minimal user programming
- Supports PSoC 3 ES3 silicon revisions and above

General Description

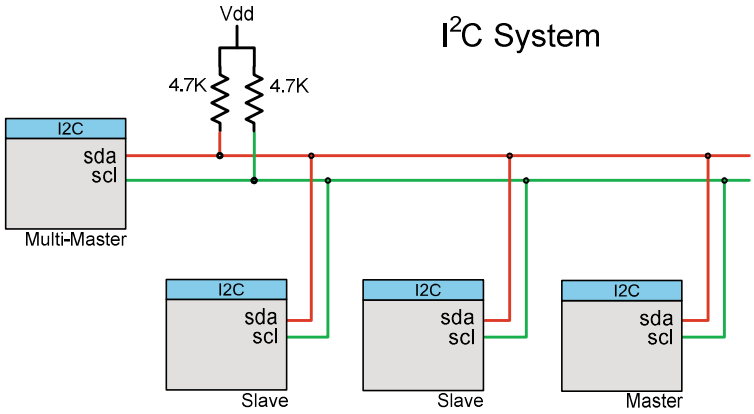
The I²C component supports I²C Slave, Master, and Multi-Master configurations. The I²C bus is an industry standard, two-wire hardware interface developed by Philips. The master initiates all communication on the I²C bus and supplies the clock for all slaved devices.

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

Note This version of the data sheet covers both the fixed hardware I²C block and the UDB version.

When to use a 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 multi-masters or a combination of multi-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

This is the I²C data signal. It is a bi-directional data signal used to transmit or receive all bus data.

scl – In/Out

The SCL signal is the master generated I²C clock. Although the slave never generates the clock signal, it may hold it low until it is ready to NAK or ACK the latest data or address.

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. If you want your bus to be 400 kHz, you need a 6.4 MHz clock. If you want a 100 kHz bus, you need a 1.6 MHz clock.

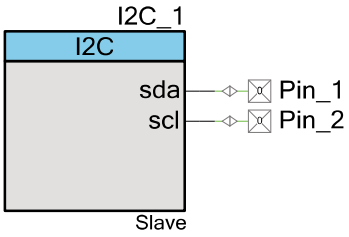
reset – Input *

The reset input is available when the ‘Implementation’ parameter is set to UDB. If the reset pin is held to a logic high, the I²C block would be held in reset, and communication over I²C would not be possible.

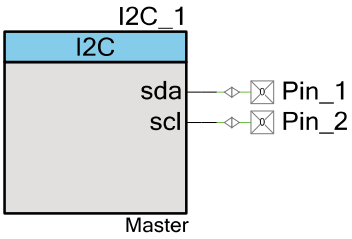
Schematic Macro Information

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

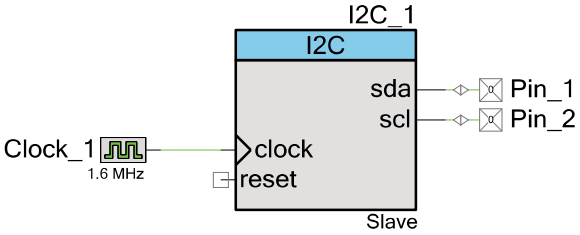
Fixed Function I²C Slave with Pins



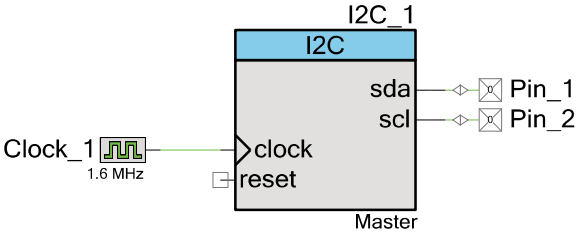
Fixed Function I²C Master Pins



UDB I²C Slave with Clock and Pins

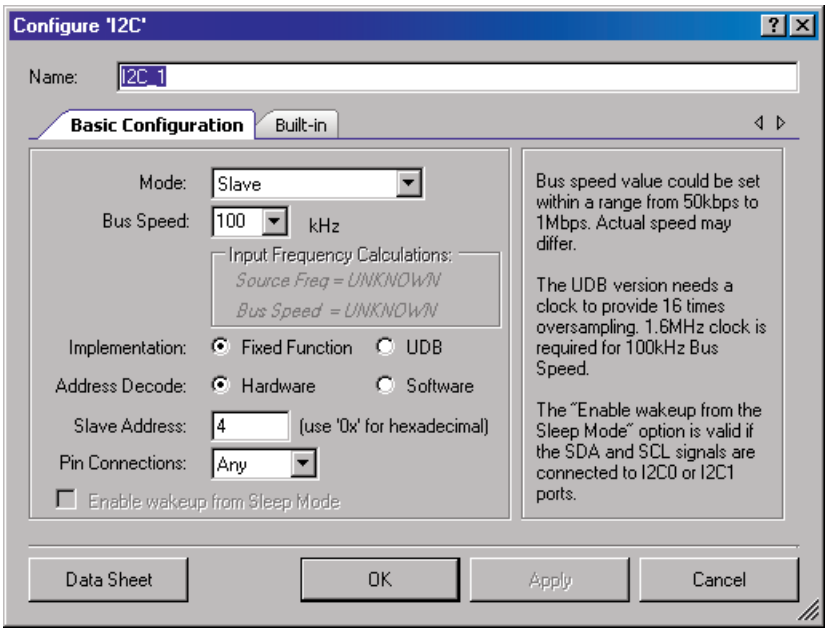


UDB I²C Master with Clock and Pins



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	Multi-Master only operation.

Bus Speed

An I²C bus speed between 50 to 1000 kHz may be selected; the actual speed may differ. The standard speeds are 50, 100 (default), 400 and 1000 kHz. This speed is referenced from the system bus clock.

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	Create the I²C in a UDB.

Address Decode

This parameter gives you the option to choose between software or hardware address decoding. For most applications where the provided API is sufficient, "Hardware" address decoding is preferred. In applications where you prefer to modify the source code to provide multiple slave address detection, "Software" address detection is preferred. Hardware is the default.

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 may be selected; the default is 4.

Pin Connections

This parameter determines which type of pin 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). If **Enable wakeup from Sleep Mode** is not required for any I/O pin, "Any" can be used for SDA and SCL. Otherwise, "I2C0" or "I2C1" must be used, which will allow you to configure the device for wakeup on I²C address match.

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 ports (I2C0 or I2C1). The default is disabled.

Refer to the device errata for this feature availability. The possibility of I²C to wake up device on slave address match should be enabled while switching to the sleep mode, refer to the "Power Management APIs" section of the *System Reference Guide*.

Clock Selection

If using a fixed function block, the clock is tied to the system bus clock and cannot be changed.

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

Resources

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

For the UDB implementation, see the following table.

Mode	Digital Blocks					API Memory (Bytes)		Pins (per External I/O)
	Datapaths	Macro cells	Status Registers	Control Registers	Counter 7	Flash	RAM	
Slave	1	25	1	1	1	TBD	TBD	TBD
Master	2	16	1	1	0	TBD	TBD	TBD
Multi-Master	2	22	1	1	0	TBD	TBD	TBD

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections 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_Int(void)	Initializes I²C registers with initial values provided from customizer.
void I2C_Enable(void)	Enables I²C operations.
void I2C_Start(void)	Starts responding to I²C traffic (Interrupt remains disabled).
void I2C_Stop(void)	Stops responding to I²C traffic (Disables interrupt).
void I2C_SaveCofig(void)	Saves none–retention I²C registers.
void I2C_Sleep(void)	Stops I²C operation and saves I²C configuration (Disables interrupt).
void I2C_RestoreConfig(void)	Restores none–retention I²C registers.
void I2C_Wakeup(void)	Restores I²C configuration and starts I²C operation (Interrupt remains disabled).
void I2C_EnableInt(void)	Enables interrupt.
void I2C_DisableInt(void)	Disables interrupt. Stop does this automatically.

void I2C_Init(void)

Description: Initializes I²C registers with initial values provided from customizer.

Parameters: None

Return Value: None

Side Effects: None

void I2C_Enable(void)

Description: Enables I²C operations.

Parameters: None

Return Value: None

Side Effects: None

void I2C_Start(void)

Description: Starts the I²C hardware. Enables Active mode power template bits or clock gating as appropriate. It is required to be executed before I²C bus operation.

The I²C interrupt remains disabled after this function call.

Parameters: None

Return Value: None

Side Effects: None

void I2C_Stop(void)

Description: Disables I²C hardware and disables I²C interrupt. Disables Active mode power template bits or clock gating as appropriate.

Parameters: None

Return Value: None

Side Effects: None

void I2C_SaveConfig(void)

Description: **Wakeup on address match enabled:** disables I²C master functionality, enables power regulator for all I²C registers, all transaction are NAKed while I2C will be ready go to sleep.

Wakeup on address match disabled: Saves I²C configuration and non-retention register values.

Parameters: None

Return Value: None

Side Effects: None

void I2C_Sleep(void)

Description: **Wakeup on address match enabled:** restores I²C master functionality, disables power regulator for all I²C registers.

Wakeup on address match disabled: Saves I²C configuration and non-retention register values. Disables active mode power template bits or clock gating as appropriate.

Disables I²C interrupt.

Parameters: None

Return Value: None

Side Effects: None

void I2C_RestoreConfig(void)

Description: **Wakeup on address match enabled:** restores I²C master functionality, disables power regulator for all I²C registers.

Wakeup on address match disabled: Restores I²C configuration and non-retention register values.

Parameters: None

Return Value: None

Side Effects: None

void I2C_Wakeup(void)

Description: **Wakeup on address match enabled:** disables I²C master functionality, enables power regulator for all I²C registers, all transaction are NAKed while I2C will be ready go to sleep.

Wakeup on address match disabled: Restores I²C configuration and non-retention register values. Restores Active mode power template bits or clock gating as appropriate.

The I²C interrupt remains disabled after function call.

Parameters: None

Return Value: None

Side Effects: None

void I2C_EnableInt(void)

Description: Enables I²C interrupt. Interrupts are required for most operations.

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

Parameters: None

Return Value: None

Side Effects: If the I²C interrupt is disabled and the master is addressing the current slave, the bus will be locked until the interrupt is re-enabled.

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 bits.
uint8 I2C_SlaveClearReadStatus(void)	Returns the read status and clear slave read status flags.
uint8 I2C_SlaveClearWriteStatus(void)	Returns the write status and clear the slave write status flags.
void I2C_SlaveSetAddress(uint8 address)	Sets slave address, a value between 0 and 127.
void I2C_SlaveInitReadBuf(uint8 * rdBuf, uint8 byteCount)	Setups the slave receive data buffer. (master -> slave)
void I2C_SlaveInitWriteBuf(uint8 * wrBuf, uint8 byteCount)	Setups the slave write buffer. (master <- slave)
uint8 I2C_SlaveGetReadBufSize(void)	Returns the amount of bytes read by the master since the buffer was reset.
uint8 I2C_SlaveGetWriteBufSize(void)	Returns the amount 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	For Master Read, sends 1 byte out Slave transmit buffer.

transmitDataByte)	
uint8 I2C_SlaveGetWriteByte (uint8 ackNak)	For a Master Write, ACKs or NAKs the previous byte and reads out the last byte transmitted.

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
I2C_SSTAT_RD_BUSY	Slave read transfer in progress
I2C_SSTAT_RD_ERR_OVFL	Master attempted to read more bytes than are in buffer.
I2C_SSTAT_RD_ERR	Slave read error.
I2C_SSTAT_WR_CMPT	Slave write transfer complete
I2C_SSTAT_WR_BUSY	Slave Write transfer in progress
I2C_SSTAT_WR_ERR_OVFL	Master attempted to write past end of buffer.
I2C_SSTAT_WR_ERR	Slave write Error

Side Effects: None

uint8 I2C_SlaveClearReadStatus(void)

Description: Returns read status flags then clears the read status flags.

Parameters: None

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

Side Effects: None

uint8 I2C_SlaveClearWriteStatus(void)

Description: Returns write status flags then clears the write status flags.

Parameters: None

Return Value: (uint8) Current write status of slave (See I2C_SlaveStatus command 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.

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 command 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: None

void I2C_SlaveInitWriteBuf(uint8 * wrBuf, uint8 bufSize)

Description: Sets the buffer pointer and size of the write buffer. This command 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 buffer exposed to the I²C master.

Return Value: None

Side Effects: None

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.

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.

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: For Master Read, sends 1 byte out Slave transmit buffer. Wait to send byte until buffer has room. Used to preload the transmit buffer. In byte by byte mode if the last byte was ACKed, stall the master (on the first bit of the next byte) if needed until the next byte is PutChared. If the last byte was NAKed it does not stall the bus because the master will generate a stop or restart condition.

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

Return Value: None

Side Effects: None

uint8 I2C_SlaveGetWriteByte (uint8 ackNak)

Description: For a Master Write, ACKs or NAKs the previous byte and reads out the last byte transmitted. The first byte read of a packet is the Address byte in which case there is no previous data so no ACK or NAK is generated. The bus is stalled until the next GetByte, therefore a GetByte must be executed after the last byte in order to send the final ACK or NAK before the Master can send a Stop or restart condition.

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: None

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_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
I2C_MSTAT_ERR_BUF_OVFL	Buffer overflow/underflow

Side Effects: None

uint8 I2C_MasterClearStatus(void)

Description: Returns the master status and clears all status flags

Parameters: None

Return Value: (uint8) Current status of master (See I2C_MasterStatus 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: Slave address.

(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 timeout occurred, process not started.
I2C_MSTR_SLAVE_BUSY	Slave operation in progress.
I2C_MSTR_ERR_LB_NAK	Last Byte Naked.
I2C_MSTR_ERR_ARB_LOST	Master lost arbitration while Start generation.

Side Effects: This function is entered without a ‘byte complete’ flag set in the I2C_CSR register, it does not exit.untill it will be set.

uint8 I2C_MasterSendRestart(uint8 slaveAddress, uint8 R_nW)

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

Parameters: (uint8) slaveAddress: 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 command for constants).

Side Effects: This function is entered without a 'byte complete' flag set in the I2C_CSR register, it does not exit.untill it will be 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: The Stop generation is required on read and write operations.

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.

Parameters: (uint8) slaveAddress: Slave address.

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

(uint8) cnt: Size of buffer to send.

(uint8) mode: Transfer mode, complete the transfer or halt before generating a stop.

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: This function is entered without a ‘byte complete’ flag set in the I2C_CSR register, it does not exit.untill it will be set.

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.

Parameters: (uint8) slaveAddress: Slave address.

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

(uint8) cnt: Size of buffer to read.

(uint8) mode: Transfer mode, complete the transfer or halt before generating a stop.

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: This function is entered without a ‘byte complete’ flag set in the I2C_CSR register, it does not exit.untill it will be set.

uint8 I2C_MasterWriteByte(uint8 theByte)

Description: Sends one byte to a slave. A Start or ReStart must be generated before this command is valid. 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	Slave operation in progress.

Side Effects: This function is entered without a ‘byte complete’ flag set in the I2C_CSR register, it does not exit.untill it will be set.

uint8 I2C_MasterReadByte(uint8 acknNak)

Description: Reads one byte from a slave and ACK or NAK the transfer. A Start or ReStart must be generated before executing this command. This function does nothing 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 buffer.

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

uint8 I2C_MasterGetReadBufSize(void)

Description: Returns the amount 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 amount of bytes that has been transferred with a 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

Sample Firmware Source Code

The following is a C language example demonstrating the basic functionality of the I²C component. This example assumes the component has been placed in a design with the default name "I2C_1."

Note If you rename your component you must also edit the example code as appropriate to match the component name you specify.

```
/*************************************************************
* Example code to demonstrate the use of the I2C Component
* as a master device. This example creates an array with the
* string "Hello World" then sends it to a slave device
* with an address of 4.
*************************************************************/
#include <device.h>   /* Part specific constants and macros */

void main()
{
   char wrData[] = "Hello World";

   I2C_1_MasterClearStatus(); /* Clear any previous status */
          I2C_1_Start();
   I2C_1_EnableInt();

   CYGlobalIntEnable;

   I2C_1_MasterWriteBuf(4, (uint8 *) wrData, 12, I2C_1_MODE_COMPLETE_XFER);
```

```
    for(;;)
    {
        if(I2C_1_MasterStatus() & I2C_MSTAT_WR_CMPLT)
        {
            /* Transfer complete */
            break;
        }
    }
}

/*************************************************************
* Example code to demonstrate the use of the I2C
* Component as a slave device.
* This example waits for an I2C master to send a packet
* of data.  When a transfer is complete, the data is
* copied into the userArray.
*************************************************************/
#include <device.h>   /* Part specific constants and macros */

void main()
{
    uint8 i;
    uint8 wrBuf[12];
    uint8 userArray[12];
    uint8 byteCnt;
    I2C_1_SlaveInitWriteBuf((uint8 *) wrBuf, 12);
    I2C_1_Start();
    I2C_1_EnableInt();
    CYGlobalIntEnable;

    /* Wait for I2C master to complete a write */
    for(;;)  /* loop forever  */
    {
        /* Wait for I2C master to complete a write */
        if(I2C_1_SlaveStatus() & I2C_SSTAT_WR_CMPT)
        {
            byteCnt = I2C_1_SlaveGetWriteBufSize();
            I2C_1_SlaveClearWriteStatus ();
            for(i=0; i < byteCnt; i++)
            {
                userArray[i] = wrBuf[i]; /* Transfer data */
            }
            I2C_1_SlaveClearWriteBuf();
        }
    }
}
```

Functional Description

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

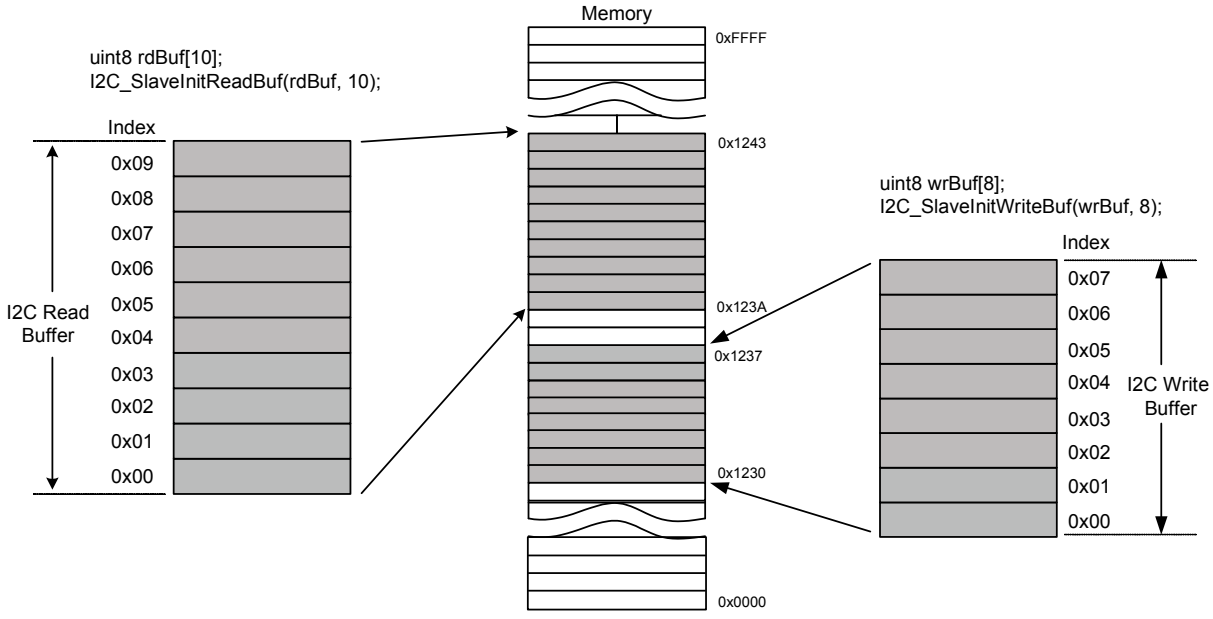
Slave Operation

The slave interface consists of two buffers in memory, one for data written to the slave by a master and a second buffer to contain data read by a master from the slave. Remember that reads and writes are from the perspective of the I²C Master. A read occurs when the master reads data from the slave. 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 programmer, 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 command.

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

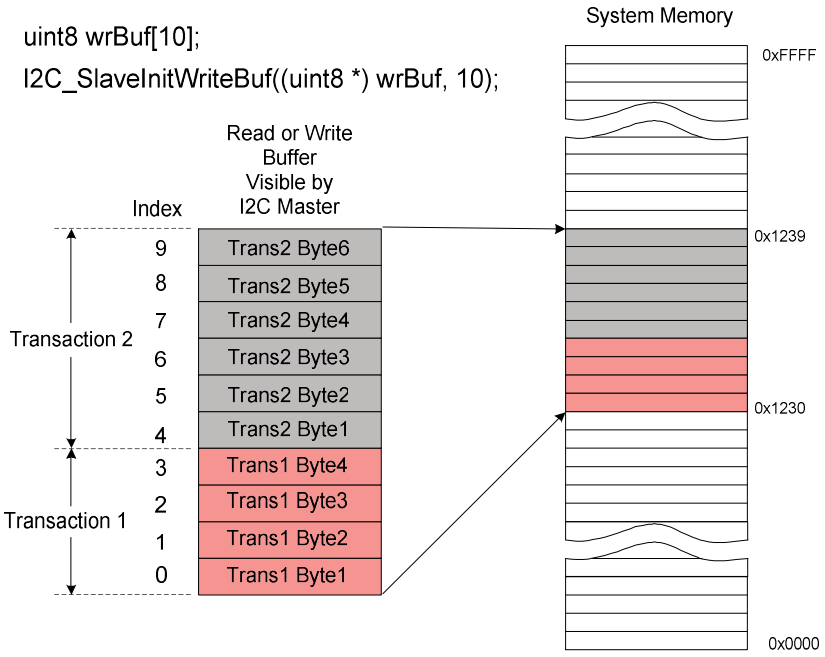
```
void I2C_SlaveClearReadBuf(void)
void I2C_SlaveClearWriteBuf(void)
```

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_RD_ERR	0x08	Slave read error.
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.
I2C_SSTAT_WR_ERR	0x80	Slave write Error

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 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(;1;)
{
   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();
```

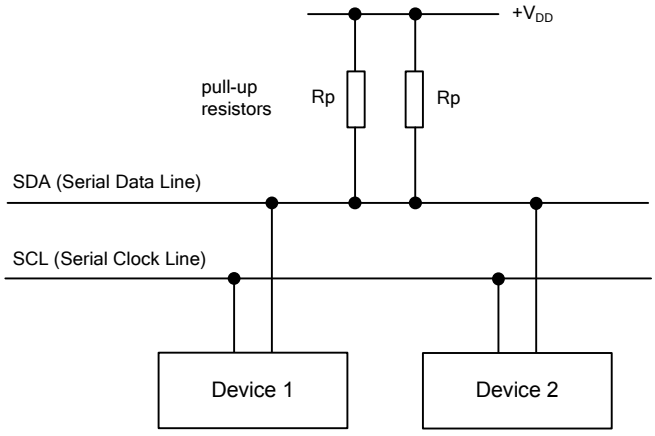
```
status = I2C_MasterSendStart(4, I2C_READ_XFER_MODE);
if(status == I2C_MSTR_NO_ERROR)   /* Check if transfer completed without errors */
{
   /* Read array of 5 bytes  */
   for(i=0; i<5; i++)
   {

      if(i < 4)
      {
         userArray[i] = I2C_MasterReadByte(I2C_ACK_DATA);
      }
      else
      {
         userArray[i] = I2C_MasterReadByte(I2C_NAK_DATA);
      }
   }
}
I2C_MasterSendStop();      /* Send Stop */
```

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 on "The I²C -Bus Specification", see 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 the CPU or DMA.

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.

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	RSVD						hw_addr_en

- csr_clk_en: Used to enable gating for the fixed function block core logic
- 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 or GPIO 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[1:0]: Used to select the bit-rate clock from 100Kbps, 400Kbps or 50Kbps
- 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: Hardware address match detection status bit: This must by cleared by writing a '0' to this bit position.
- 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 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				bus_busy	master_mode	restart_gen	start_gen

- 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, 0 indicates a stop condition was detected, 1 indicates a start condition was detected.
- 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: Used to generate a stop condition on the bus. This bit must be cleared by firmware after a suitable amount of time.
- restart_gen: Used to generate a restart condition on the bus. This bit must be cleared by firmware after a suitable amount of time.
- nack: Used to NAK the next read byte. This bit must be cleared by firmware between bytes.
- transmit: Used to set the current mode to transmit or receive a byte of data. This but must be cleared by firmware after the byte has started transmitting.
- en_master: Used to enable the hardware block.

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 giving all bits configured as Mode=1 the timing resolution of the counter, these bits are sticky and are cleared on a read of the status register. All other bits configured as mode=0 are transparent and read directly from the inputs to the status register, they are not sticky and therefore not clear 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*: Indicates the arbitration was lost (Multi-Master Modes).
- stop_status*: Stop detection status bit.

- bus_busy: Indicates the bus is busy. 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.
- byte_complete: Indicates 8-bits were received or transmitted.

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. Any write to this register will force a byte transmit.

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: Used to NACK the last byte received. This bit must be cleared by firmware between bytes.
- any_address: Used to enable the device to respond any device addresses it receives
- transmit: Used to set the mode to transmit or receive data. This bit must be cleared by firmware between bytes.
- en_slave: Used to enable or disable the hardware block.

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 giving all bits configured as Mode=1 the timing resolution of the counter, these bits are sticky and are cleared on a read of the status register. All other bits configured as mode=0 are transparent and read directly from the inputs to the status register, they are not sticky and therefore not clear 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*: Indicates a stop condition was detected on the bus.
- address: Address detection. Indicates that an address byte was received.
- lbr: Last Received Bit. Indicates the state of the last received bit which is the ACK/NACK received for the last byte transmitted.
- byte_complete: Indicates 8-bits were received or transmitted.

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. Any write to this register will force a byte transmit.

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.

Component Debug Window

The I²C component supports the PSoC Creator component debug window. Refer to the appropriate device data sheet for a detailed description of each register. The following registers are displayed in the I²C component debug window. Some registers are available in the UDB

implementation (indicated by *) and some registers are only available in the Fixed Function Implementation (indicated by **). All other registers are available for either configuration.

Register: I2C_1_XCFG*

Name: Extended Configuration Register

Description: Used to configure some of the advanced configuration options of the Fixed Function block. Refer to I2C_XCFG register description above for bit-field definitions.

Register: I2C_1_ADDR

Name: Slave Address Register

Description: Used to indicate the 7-bit slave address for hardware address match detection in both fixed function and UDB implementations.

Register: I2C_1_CFG

Name: Configuration Register

Description: Used to configure the standard configuration options of the Fixed Function and UDB implementations. Refer to I2C_CFG register descriptions above for bit-field definitions. Note that the bit-fields are not the same for UDB and Fixed Function implementations.

Register: I2C_1_CSR

Name: Status Register

Description: For the Fixed Function block this register is the status feedback register from hardware and includes some run-time control bits as a shared register. The UDB implementation of these registers are independent and become the CSR (Status Register) and CFG Control register. Refer to I2C_CSR register descriptions above for bit-field definitions.

Register: I2C_1_DATA

Name: Transmit and Receive Data Register

Description: Used to load transmit data and read received data. Refer to I2C_DATA register descriptions above for bit-field definitions.

Register: I2C_1_MCSR*

Name: Master Control and Status Register

Description: Used for runtime control and status feedback of Master mode operations within the Fixed Function Hardware Block. Refer to I2C_MCSR register description above for bit-field definitions.

References

Not applicable

DC and AC Electrical Characteristics

5.0V/3.3V DC and AC Electrical Characteristics

Parameter	Typical	Min	Max	Units	Conditions and Notes
Input					
Input Voltage Range	---		Vss to Vdd	V	
Input Capacitance	---		---	pF	
Input Impedance	---		---	Ω	
Maximum Clock Rate	---		67	MHz	

Component Changes

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

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

© Cypress Semiconductor Corporation, 2009-2013. 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.