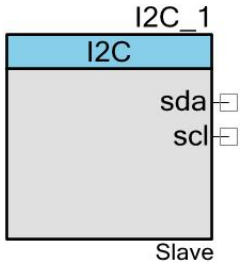
Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

I2C Master/Multi-Master/Slave

3.50



Features

- Industry-standard NXP® I²C bus interface
- Supports slave, master, multi-master and multi-master-slave operation
- Requires only two pins (SDA and SCL) to interface to I²C bus
- Supports standard data rates of 100/400/1000 kbps
- 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.

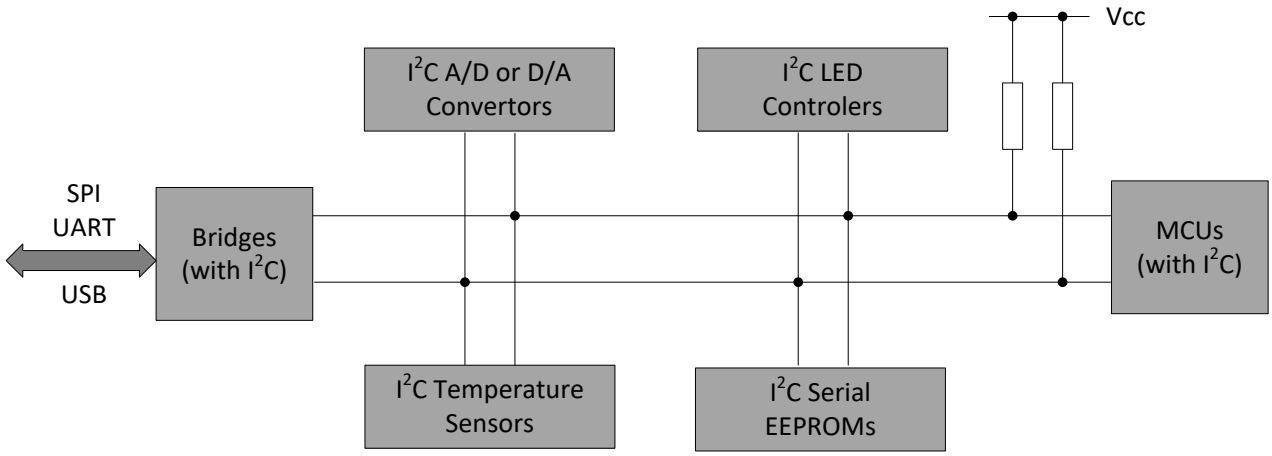
The I²C component supports standard clock speeds up to 1000 kbps. It is compatible [1] with I²C Standard-mode, Fast-mode, and Fast-mode Plus devices as defined in the NXP I²C-bus specification. The I²C component is compatible with other third-party slave and master devices.

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

1. The I2C peripheral is non-compliant with the NXP I2C specification in the following areas: analog glitch filter, I/O VOL/IOL, I/O hysteresis. The I2C Block has a digital glitch filter (not available in sleep mode). The Fast-mode minimum fall-time specification can be met by setting the I/Os to slow speed mode. See the I/O Electrical Specifications in "Inputs and Outputs" section of device datasheet for details.

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[2] the latest data or address. The pin connected to scl should be configured as Open-Drain-Drives-Low.

Note The default threshold voltages for the Pins component is CMOS. When the I²C lines are pulled up to 3.3 V and the PSoC is running on 5.0 V, the CMOS threshold levels may not guarantee reliable logic transitions. In this case, the SCL and SDA pin component thresholds should be set to TTL.

---

2 NAK is an abbreviation for negative acknowledgment or not acknowledged. I²C documents commonly use NACK while the rest of the networking world uses NAK. They mean the same thing.

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 is held in reset, and communication over I²C stops. This is a hardware reset only. Software must be independently reset calling the I2C_Stop() and I2C_Start() or I2C_Enable() APIs. The reset input may be left floating with no external connection. If nothing is connected to the reset line, the component will assign it a constant logic "0".

I²C Bus Multiplexing

The following inputs and outputs are only available when the **External OE buffer** option is selected (on the **Advanced** tab). The tri-state buffers (placed side the component) are removed, and bidirectional scl and sda terminals are replaced with separate sda_i and scl_i inputs, as well as sda_o and scl_o outputs. This allows I²C bus multiplexing inside PSoC.

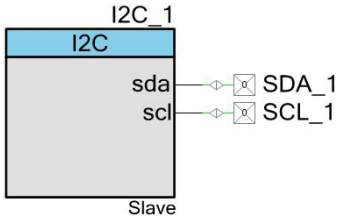
- **scl_i – Input * –** Serial clock input clock signal used to sense the bus clock line.
- **scl_o – Output * –** Serial clock output signal used to drive the bus clock line.
- **sda_i – Input * –** Serial data input signal used to sense the bus data line.
- **sda_o – Output * –** Serial data output signal used to drive the bus data line.

The appropriate pair of signals (input and output) must be connected to the pin component to drive the pin. The scl_i and scl_o has to be connected to SCL pin and sda_i and sda_o to SDA pin appropriately. The signals to pin connection options are depicted on the Figure 5.

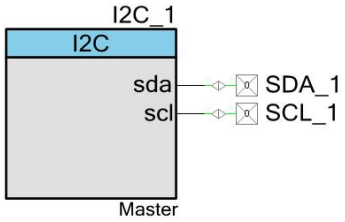
Schematic Macro Information

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.

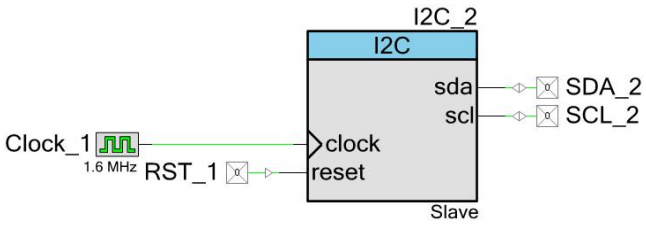
Fixed-Function I²C Slave with Pins



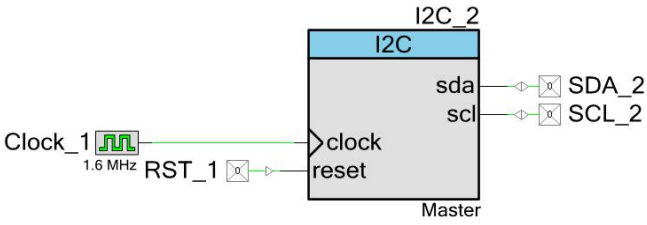
Fixed-Function I²C Master Pins



UDB I²C Slave with Clock and Pins



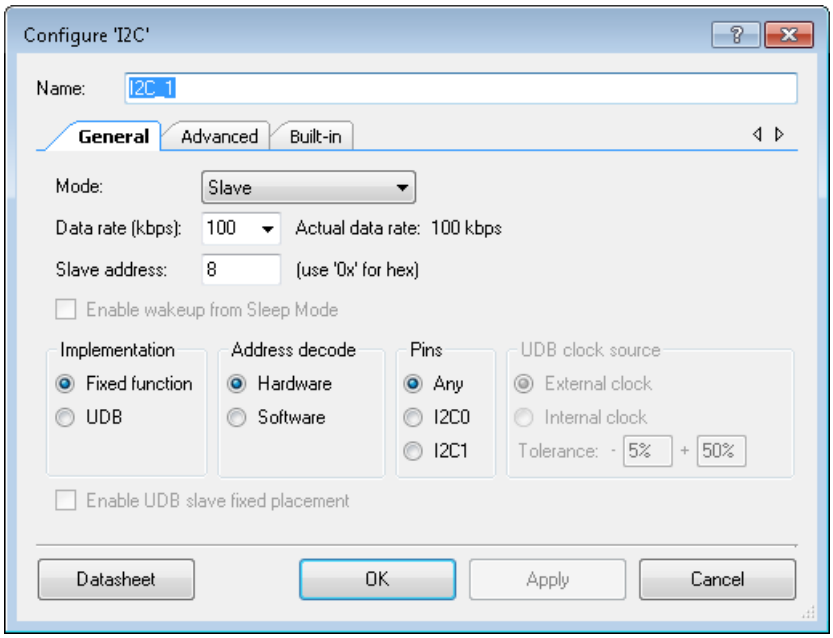
UDB I²C Master with Clock and Pins



Component Parameters

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

General Tab



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

Mode

Use this option to select the I²C mode: slave, master, multi-master, or multi-master-slave.

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

Note For slave modes (slave or multi-master-slave), **Implementation** is Fixed function and **Address decode** is Hardware. The usage of I²C repeated Start condition to join transactions to different devices is not supported. If the I²C master accesses other I²C devices and then generates a repeated Start to access the component, the component fails to operate properly. If I²C transactions of this type cannot be avoided, set **Implementation** to UDB or change **Address decode** to Software.

Data rate

This parameter is used to set the I²C data rate value up to 1000 kbps. The standard data rates [3] are 50, 100 (default), 400, and 1000 kbps. The component also displays the actual data rate at which the component will operate with current settings. The factors that affect the actual data rate calculation include the accuracy of the component clock (internal or external) and oversampling factor. For more information about these parameters, refer to the Clock Selection section.

If **Implementation** is set to **UDB** and the **UDB clock source** parameter is set to **External clock**, the **Data rate** parameter is ignored; the 16x input clock determines the data rate.

There are cases when the system cannot build the desired I²C clock and the selected data rate cannot be supported. The component does not provide an error or warning in this case, but the actual data rate may differ significantly from the selected data rate. For example, if the I²C fixed-function block selected data rate is 400 kbps and the BUS_CLK = 3MHz, the required I²C clock is 16 * 400 kHz = 6.4 MHz, which cannot be created from the BUS_CLK.

Note For master modes (master, multi-master, or multi-master-slave), **Implementation** is UDB. The real master speed for data rates above 400 kbps may differ depending on the BUS_CLK value, rise and fall times of f_{SCL}[4], and component placement.

Slave address

This is the I²C address that will be recognized by the slave. If slave operation is not selected, this parameter is ignored. You can select a slave address between 0 and 127 (0x00 and 0x7F); the default is **8**. This address is the 7-bit right-justified slave address and does not include the R/W bit. You can enter the value as decimal or hexadecimal; for hexadecimal numbers type ‘0x’ before the address. If a 10-bit slave address is required, you must use software address decoding and provide decode support for the second byte of the 10-bit address in the ISR.

Implementation

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

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

3 The fixed-function implementation supports only standard data rates 50, 100, and 400 kbps for PSoC 5LP devices. The UDB-based implementation should be used for different data rates up to 1000 kbps.

4 Refer to the I²C-Bus Specification Rev. 6, section 7.2.1 Reduced f_{SCL}.

Address decode

This parameter allows you 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 (default) is preferred. In applications where you prefer to modify the source code to provide detection of multiple slave addresses or 10-bit addresses, you must use software address detection.

If hardware address decode is enabled, the block automatically NAKs addresses that are not its own without CPU intervention. It automatically interrupts the CPU on correct address reception, and holds the SCL line low until CPU intervention.

Pins

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

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

Any means general-purpose I/O (GPIO or SIO) are used for SCL and SDA pins. This is a general usage case when wakeup from Sleep mode on slave address match is not required. Otherwise, select the **Enable wakeup from Sleep Mode** option and set **Pins** to I2C0 or I2C1, depending on the placement capabilities.

Note The I^2C component does not check the correct pin assignments.

Enable wakeup from Sleep Mode

This option allows the system to be awakened from Sleep when an address match occurs. This option is only valid if **Address decode** is set to hardware and the SDA and SCL signals are connected to SIO pins (I2C0 or I2C1). The option is disabled by default. This option is supported by the PSoC 3 and PSoC 5LP devices.

You must enable the possibility for the I^2C to wake up the device on slave address match while switching to the sleep mode. You can do this by calling the 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*.

UDB clock source

This parameter allows you to choose between an internally configured clock and an externally configured clock for data rate generation. When set to **Internal clock**, PSoC Creator calculates and configures the required clock frequency based on the **Data rate** parameter, taking into account 16 times oversampling. In **External clock** mode the component does not control the

data rate but displays the actual data rate based on the user-connected clock source. If this parameter is set to **Internal clock** then the clock input is not visible on the symbol.

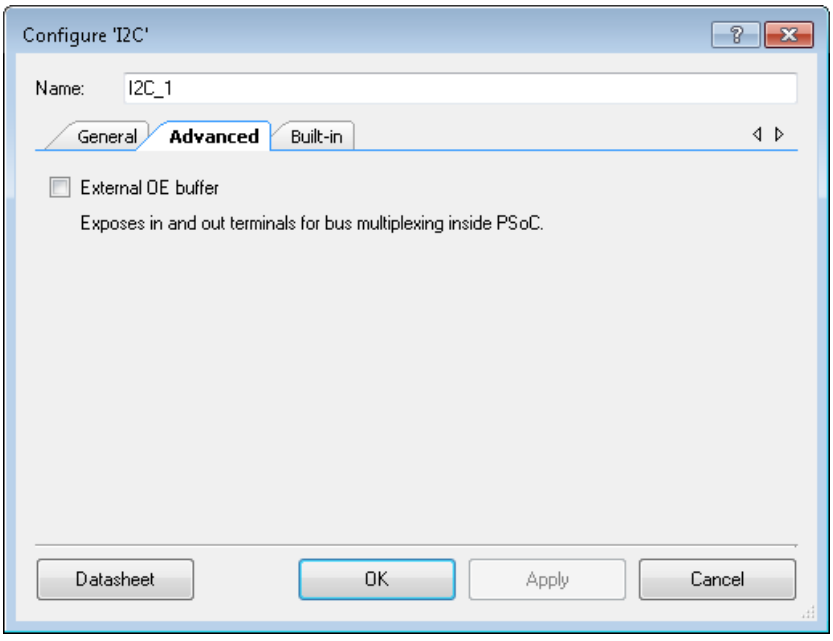
You can enter the desired tolerance values for the internal clock. Clock tolerances are specified as a percentage. The default range for slave mode is **-5% to +50%**. The clock can be fast in this mode. For the remaining modes, the default range is -25% to +5%. Again, the master can be slow. At the maximum data rate (1000 kbps), the clock should be equal or slower, but not faster than expected. This could cause unexpected behavior.

Enable UDB slave fixed placement

This parameter allows you to choose a fixed component placement that improves the component performance over unconstrained placement. If this parameter is set, all of the component resources are fixed in the top right corner of the device. This parameter controls the assignment of pins connected to the component. The choice of pin assignment is not a determining factor for component performance. This option is only valid if **Mode** is set to Slave and **Implementation** is set to **UDB.** This option is disabled by default.

The fixed placement aspect of the component removes the routing variability. It also allows the fixed placement to continue to operate the same as a non-fixed placed design would in a fairly empty design.

Advanced Tab



The **Advanced** tab has the following parameter:

External OE buffer

This parameter allows I^2C bus multiplexing inside the PSoC. The tri-state buffers (placed inside the component) are removed, and bidirectional scl and sda terminals are replaced with separate inputs (sda_i and scl_i) and outputs (sda_o and scl_o). Refer to Internal I^2C Bus Multiplexing section for more information and usage cases.

Clock Selection

When the internal clock configuration is selected, PSoC Creator calculates the needed frequency of the I^2C clock source and generates the resource for implementation. Otherwise, you must supply the I^2C clock and calculate the required clock frequency (only the UDB implementation provides a choice between internal and external clock). That frequency is 16x the desired data rate available. For example, a 1.6 MHz clock is required for a 100 kbps data rate.

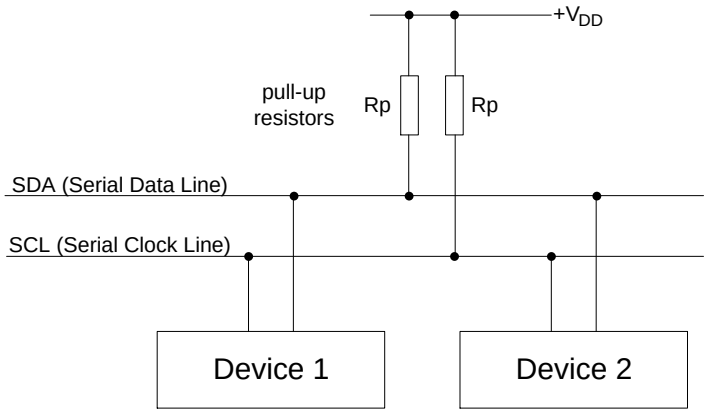
The fixed-function block uses a divided BUS_CLK. The divider is calculated to achieve the 16/32 oversampling of the selected data rate (50 kbps data rate requires 32 oversampling; all other data rates require 16 oversampling).

External Electrical Connections

As shown in the following figure, the I^2C bus requires external pull-up resistors. The pull-up resistors (R_P) are primarily determined by the supply voltage, bus speed, and bus capacitance. For detailed information on how to calculate the optimum pull-up resistor value for your design we recommend using the UM10204 I^2C-bus specification and user manual Rev. 6, available from the NXP website at www.nxp.com.

An alternative to using external pull-up resistors is to use the pins internal pull-up resistors. To use this option, the SCL and SDA pins drive modes must be set to resistive pull-up. Typical internal PSoC pull-ups are 5.6 kΩ with a tolerance exceeding 5%. Therefore, it is not recommended to use them in other than Standard mode, after validating that the minimum and maximum values meet the requirements of your system.

Figure 1. Connection of Devices to the I²C Bus



For most designs, the default values shown in the following table provide excellent performance without any calculations. The default values were chosen to use standard resistor values between the minimum and maximum limits.

Table 1. Recommended Default Pull-up Resistor Values

Standard Mode (0 – 100 kbps)	Fast Mode (0 – 400 kbps)	Fast Mode Plus (0 – 1000 kbps)	Units
4.7 k, 5%	1.74 k, 1%	620, 5%	Ω

These values work for designs with 1.8 V to 5.0V $V_{DD,}$ less than 200 pF bus capacitance (C_B), up to 25 µA of total input leakage (I_{IL}), up to 0.4 V output voltage level (V_{OL}), and a max V_{IH} of 0.7 * V_{DD}.

Standard Mode and Fast Mode can use either GPIO or SIO PSoC pins. Fast Mode Plus requires use of SIO pins to meet the V_{OL} spec at 20 mA. Calculation of custom pull-up resistor values is required if; your design does not meet the default assumptions, you use series resistors (R_S) to limit injected noise, or you want to maximize the resistor value for low power consumption.

Calculation of the ideal pull-up resistor value involves finding a value between the limits set by three equations detailed in the NXP I²C specification. These equations are:

Equation 1: $R_{PMIN} = (V_{DD}(max) – V_{OL}(max)) / I_{OL}(min)$

Equation 2: $R_{PMAX} = T_R(max) / 0.8473 \times C_B(max)$

Equation 3: $R_{PMAX} = V_{DD}(min) – (V_{IH}(min) + V_{NH}(min)) / I_{IH}(max)$

Equation parameters:

- V_{DD} = Nominal supply voltage for I²C bus
- V_{OL} = Maximum output low voltage of bus devices.

- I_{OL}= Low level output current from I^2C specification
- T_R = Rise Time of bus from I^2C specification
- C_B = Capacitance of each bus line including pins and PCB traces
- V_{IH} = Minimum high level input voltage of all bus devices
- V_{NH} = Minimum high level input noise margin from I^2C specification
- I_{IH} = Total input leakage current of all devices on the bus

The supply voltage (V_{DD}) limits the minimum pull-up resistor value due to bus devices maximum low output voltage (V_{OL}) specifications. Lower pull-up resistance increases current through the pins and can therefore exceed the spec conditions of V_{OH}. Equation 1 is derived using Ohm's law to determine the minimum resistance that will still meet the V_{OL} specification at 3 mA for standard and fast modes, and 20 mA for fast mode plus at the given V_{DD}.

Equation 2 determines the maximum pull-up resistance due to bus capacitance. Total bus capacitance is comprised of all pin, wire, and trace capacitance on the bus. The higher the bus capacitance the lower the pull-up resistance required to meet the specified bus speeds rise time due to RC delays. Choosing a pull-up resistance higher than allowed can result in failing timing requirements resulting in communication errors. Most designs with five of fewer I^2C devices and up to 20 centimeters of bus trace length have less than 100 pF of bus capacitance.

A secondary effect that limits the maximum pull-up resistor value is total bus leakage calculated in Equation 3. The primary source of leakage is I/O pins connected to the bus. If leakage is too high, the pull-ups will have difficulty maintaining an acceptable V_{IH} level causing communication errors. Most designs with five or fewer I^2C devices on the bus have less than 10 µA of total leakage current.

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component during run time. The following table lists and describes the interface to each function. The subsequent sections discuss each function in more detail.

By default, PSoC Creator assigns the instance name "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 I^2C 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
I2C_Start()	Initializes and enables the I²C component. The I²C interrupt is enabled, and the component can respond to I²C traffic.
I2C_Stop()	Stops responding to I²C traffic (disables the I²C interrupt).
I2C_EnableInt()	Enables interrupt, which is required for most I²C operations.
I2C_DisableInt()	Disables interrupt. The I2C_Stop() API does this automatically.
I2C_Sleep()	Stops I²C operation and saves I²C nonretention configuration registers (disables the interrupt). Prepares wake on address match operation if Wakeup from Sleep Mode is enabled (disables the I²C interrupt).
I2C_Wakeup()	Restores I²C nonretention configuration registers and enables I²C operation (enables the I²C interrupt).
I2C_Init()	Initializes I²C registers with initial values provided from the customizer.
I2C_Enable()	Activates I²C hardware and begins component operation.
I2C_SaveConfig()	Saves I²C nonretention configuration registers (disables the I²C interrupt).
I2C_RestoreConfig()	Restores I²C nonretention configuration registers saved by I2C_SaveConfig() or I2C_Sleep() (enables the I²C interrupt).

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 enables the I²C interrupt. Interrupts are required for most I²C operations.

You must set up the I²C Slave buffers before this function call to avoid reading or writing partial data while the buffers are setting up.

I²C slave behavior is as follows when enabled and buffers are not set up:

- I²C Read transfer – Returns 0xFF until the read buffer is set up. Use the I2C_SlaveInitReadBuf() function to set up the read buffer.
- I²C Write transfer – Send NAK because there is no place to store received data. Use the I2C_SlaveInitWriteBuf() function to set up the read buffer.

Parameters: None

Return Value: None

Side Effects: None

void I2C_Stop(void)

Description: This function disables I^2C hardware and component interrupt.
Releases the I^2C bus if it was locked up by the device and sets it to the idle state.

Parameters: None

Return Value: None

Side Effects: None

void I2C_EnableInt(void)

Description: This function enables the I^2C interrupt. Interrupts are required for most operations.

Parameters: None

Return Value: None

Side Effects: None

void I2C_DisableInt(void)

Description: This function disables the I^2C interrupt. This function is not normally required because the I2C_Stop() function disables the interrupt.

Parameters: None

Return Value: None

Side Effects: If the I^2C interrupt is disabled while the I^2C is still running, it can cause the I^2C bus to lock up.

void I2C_Sleep(void)

Description: This is the preferred method to prepare the component before device enters sleep mode.

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

- Unchecked: Checks current I²C component state, saves it, and disables the component by calling I2C_Stop() if it is currently enabled. I2C_SaveConfig() is then called to save the component nonretention configuration registers.
- Checked: If a transaction intended for component executes during this function call, it waits until the current transaction is completed. All subsequent I²C traffic intended for component is NAKed until the device is put to sleep mode. The address match event wakes up the device.

Call the I2C_Sleep() function before calling the CyPmSleep() or the CyPmHibernate() function. See 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 method to prepare the component for active mode operation (when device exits sleep mode).

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

- Unchecked: Restores the component nonretention configuration registers by calling I2C_RestoreConfig(). If the component was enabled before the I2C_Sleep() function was called, I2C_Wakeup() re-enables it.
- Checked: Enables master functionality if it was enabled before sleep, and disables the backup regulator of the I²C hardware. The incoming transaction continues as soon as the regular I²C interrupt handler is set up (global interrupts has to be enabled to service I²C component interrupt).

Parameters: None

Return Value: None

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

void I2C_Init(void)

Description: This function 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: This function 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: The Enable wakeup from Sleep Mode selection influences this function implementation:

- Unchecked: Stores the component nonretention configuration registers.
- Checked: Disables the master, if it was enabled before, and enables backup regulator of the I^2C hardware. If a transaction intended for component executes during this function call, it waits until the current transaction is completed and I^2C hardware is ready to enter sleep mode. All subsequent I^2C traffic is NAKed until the device is put into sleep mode.

Parameters: None

Return Value: None

Side Effects: None

void I2C_RestoreConfig(void)

Description: The Enable wakeup from Sleep Mode selection influences this function implementation:

- Unchecked: Restores the component nonretention configuration registers to the state they were in before I2C_Sleep() or I2C_SaveConfig() was called.
- Checked: Enables master functionality, if it was enabled before, and disables the backup regulator of the I²C hardware. Sets up the regular component interrupt handler and generates the component interrupt if it was wake up source to release the bus and continue in-coming I²C transaction.

Parameters: None

Return Value: None

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

Slave Functions

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

Function	Description
I2C_SlaveStatus()	Returns the slave status flags.
I2C_SlaveClearReadStatus()	Returns the read status flags and clears the slave read status flags.
I2C_SlaveClearWriteStatus()	Returns the write status and clears the slave write status flags.
I2C_SlaveSetAddress()	Sets the slave address, a value between 0 and 127 (0x00 to 0x7F).
I2C_SlaveInitReadBuf()	Sets up the slave receive data buffer. (master <- slave)
I2C_SlaveInitWriteBuf()	Sets up the slave write buffer. (master -> slave)
I2C_SlaveGetReadBufSize()	Returns the number of bytes read by the master since the buffer was reset.
I2C_SlaveGetWriteBufSize()	Returns the number of bytes written by the master since the buffer was reset.
I2C_SlaveClearReadBuf()	Resets the read buffer counter to zero.
I2C_SlaveClearWriteBuf()	Resets the write buffer counter to zero.

uint8 I2C_SlaveStatus(void)

Description: This function returns the slave's communication status.

Parameters: None

Return Value: uint8: Current status of I²C slave. This status incorporates read and write status flags.

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

Slave Status Constants	Description
I2C_SSTAT_RD_CMPLT[5]	Slave read transfer complete. Set when the master sends a NAK to say that it is done reading.
I2C_SSTAT_RD_BUSY	Slave read transfer in progress. Set when the master addresses the slave with a read, cleared when RD_CMPLT is set.
I2C_SSTAT_RD_ERR_OVFL	The master attempted to read more bytes than are in the buffer.
I2C_SSTAT_WR_CMPLT[6]	Slave write transfer complete. Set when a Stop condition is received.
I2C_SSTAT_WR_BUSY	Slave write transfer in progress. Set when the master addresses the slave with a write and cleared when WR_CMPLT is set.
I2C_SSTAT_WR_ERR_OVFL	The master attempted to write past the end of the buffer. The incoming byte is NAKed by the slave.

Side Effects: None

uint8 I2C_SlaveClearReadStatus(void)

Description: This function clears the read status flags and returns their values.

The I2C_SSTAT_RD_BUSY flag is not affected by this function call.

Parameters: None

Return Value: uint8: Current read status of the I²C slave. See the I2C_SlaveStatus() function for constants.

Side Effects: None

---

5 The definition was changed from I2C_SSTAT_RD_CMPT to I2C_SSTAT_RD_CMPLT to comply with the master read complete definition. The component supports both definitions, but the I2C_SSTAT_RD_CMPT will become obsolete.

6 The definition was changed from I2C_SSTAT_WR_CMPT to I2C_SSTAT_WR_CMPLT to comply with the master write complete definition. The component supports both definitions, but the I2C_SSTAT_WR_CMPT will become obsolete.

uint8 I2C_SlaveClearWriteStatus(void)

Description: This function clears the write status flags and returns their values.

The I2C_SSTAT_WR_BUSY flag is not affected by this function call.

Parameters: None

Return Value: uint8: Current write status of the I²C slave. See the I2C_SlaveStatus() function for constants.

Side Effects: None

void I2C_SlaveSetAddress(uint8 address)

Description: This function sets the I²C slave address

Parameters: uint8 address: I²C slave address for the primary device. This value can be any address between 0 and 127 (0x00 to 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: This function 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: This function 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: This function returns the number of bytes read by the I²C master since an I2C_SlaveInitReadBuf() or I2C_SlaveClearReadBuf() function was executed.
The maximum return value is the size of the read buffer.

Parameters: None

Return Value: uint8: Bytes read by the master.

Side Effects: None

uint8 I2C_SlaveGetWriteBufSize(void)

Description: This function returns the number of bytes written by the I²C master since an I2C_SlaveInitWriteBuf() or I2C_SlaveClearWriteBuf() function was executed.
The maximum return value is the size of the write buffer.

Parameters: None

Return Value: uint8: Bytes written by the master.

Side Effects: None

void I2C_SlaveClearReadBuf(void)

Description: This function resets the read pointer to the first byte in the read buffer. The next byte the master reads will be the first byte in the read buffer.

Parameters: None

Return Value: None

Side Effects: None

void I2C_SlaveClearWriteBuf(void)

Description: This function resets the write pointer to the first byte in the write buffer. The next byte the master writes will be the first byte in the write buffer.

Parameters: None

Return Value: None

Side Effects: None

Master and Multi-Master Functions

These functions are only available if master or multi-master mode is enabled.

Function	Description
I2C_MasterStatus()	Returns the master status.
I2C_MasterClearStatus()	Returns the master status and clears the status flags.
I2C_MasterWriteBuf()	Writes the referenced data buffer to a specified slave address.
I2C_MasterReadBuf()	Reads data from the specified slave address and places the data in the referenced buffer.
I2C_MasterSendStart()	Sends only a Start to the specific address.
I2C_MasterSendRestart()	Sends only a Restart to the specified address.
I2C_MasterSendStop()	Generates a Stop condition.
I2C_MasterWriteByte()	Writes a single byte. This is a manual command that should only be used with the I2C_MasterSendStart() or I2C_MasterSendRestart() functions.
I2C_MasterReadByte()	Reads a single byte. This is a manual command that should only be used with the I2C_MasterSendStart() or I2C_MasterSendRestart() functions.
I2C_MasterGetReadBufSize()	Returns the byte count of data read since the I2C_MasterClearReadBuf() function was called.
I2C_MasterGetWriteBufSize()	Returns the byte count of the data written since the I2C_MasterClearWriteBuf() function was called.
I2C_MasterClearReadBuf()	Resets the read buffer pointer back to the beginning of the buffer.
I2C_MasterClearWriteBuf()	Resets the write buffer pointer back to the beginning of the buffer.

uint8 I2C_MasterStatus(void)

Description: This function returns the master's communication status.

Parameters: None

Return Value: uint8: Current status of the I²C master. Each constant is a bit field value. The value returned may have multiple bits set to indicate the status of the transfer along with the generation of error conditions.

Master status constants	Description
I2C_MSTAT_RD_CMPLT	Read transfer complete. The error condition bits must be checked to ensure that the read transfer was successful.
I2C_MSTAT_WR_CMPLT	Write transfer complete. The error condition bits must be checked to ensure that the write transfer was successful.
I2C_MSTAT_XFER_INP	Transfer in progress
I2C_MSTAT_XFER_HALT	Transfer has been halted. The I²C bus is waiting for the master to generate a Restart or Stop condition.
I2C_MSTAT_ERR_SHORT_XFER	Error condition: Write transfer completed before all bytes were transferred.
I2C_MSTAT_ERR_ADDR_NAK	Error condition: The slave did not acknowledge the address.
I2C_MSTAT_ERR_ARB_LOST	Error condition: The master lost arbitration during communication with the slave.
I2C_MSTAT_ERR_XFER	Error condition: This is the ORed value of error conditions provided in this table. If all error condition bits are cleared, but this bit is set, the transfer was aborted because of slave operation.

Side Effects: None

uint8 I2C_MasterClearStatus(void)

Description: This function clears all status flags and returns the master status.

Parameters: None

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

Side Effects: None

uint8 I2C_MasterWriteBuf(uint8 slaveAddress, uint8 * wrData, uint8 cnt, uint8 mode)

Description: This function automatically writes an entire buffer of data to a slave device. After the data transfer is initiated by this function, the included ISR manages further data transfer in byte-by-byte mode. Enables the I²C interrupt.

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

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

uint8 cnt: Number of bytes of the 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 the I2C_MasterSendStart() function for constants.

Side Effects: None

uint8 I2C_MasterReadBuf(uint8 slaveAddress, uint8 * rdData, uint8 cnt, uint8 mode)

Description: This function automatically reads an entire buffer of data from a slave device. Once this function initiates the data transfer, the included ISR manages further data transfer in byte by byte mode. Enables the I²C interrupt.

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

uint8 rdData: Pointer to the buffer in which to put the data from the slave.

uint8 cnt: Number of bytes of the 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 the I2C_MasterSendStart() function for constants.

Side Effects: None

uint8 I2C_MasterSendStart(uint8 slaveAddress, uint8 R_nW)

Description: This function generates a Start condition and sends the slave address with the read/write bit. Disables the I²C interrupt.

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

uint8 R_nW: Set to zero, send write command; set to nonzero, send read command.

Return Value: uint8: Error Status.

Mode Constants	Description
I2C_MSTR_NO_ERROR	Function completed without error.
I2C_MSTR_BUS_BUSY	Bus is busy, Start condition was not generated.
I2C_MSTR_NOT_READY	The master is not a valid master on the bus, or a slave operation is in progress.
I2C_MSTR_ERR_LB_NAK	The last byte was NAKed.
I2C_MSTR_ERR_ARB_LOST	The master lost arbitration while the Start was generated. (This status is only valid if multi-master is enabled.)
I2C_MSTR_ERR_ABORT_START_GEN	Start condition generation was aborted because of the start of slave operation. (This status is only valid in multi-master-slave mode.)

Side Effects: This function is blocking and does not exit until the byte_complete bit is set in the I2C_CSR register.

uint8 I2C_MasterSendRestart(uint8 slaveAddress, uint8 R_nW)

Description: This function generates a restart condition and sends the slave address with the read/write bit.

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

uint8 R_nW: Set to zero, send write command; set to nonzero, send read command.

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

Side Effects: This function is blocking and does not exit until the byte_complete bit is set in the I2C_CSR register.

uint8 I2C_MasterSendStop(void)

Description: Generates Stop condition on the bus. The NAK is generated before Stop in case of a read transaction. At least one byte has to be read if a Start or ReStart condition with read direction was generated before.

This function does nothing if Start or Restart conditions failed before this function was called.

Parameters: None

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

Side Effects: This function is blocking and does not exit until:

Master: This function waits while a stop condition is generated.

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

uint8 I2C_MasterWriteByte(uint8 theByte)

Description: This function sends one byte to a slave.

A valid Start or Restart condition must be generated before calling this function. This function does nothing if the Start or Restart conditions failed before this function was called.

Parameters: uint8 theByte: 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_NOT_READY	The master is not a valid master on the bus or slave operation is in progress.
I2C_MSTR_ERR_LB_NAK	The last byte was NAKed.
I2C_MSTR_ERR_ARB_LOST	The master lost arbitration. (This status is valid only if multi-master is enabled.)

Side Effects: This function is blocking and does not 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 generates ACK or prepares to generate NAK. The NAK will be generated before Stop or ReStart condition by SCB_MasterSendStop() or SCB_MasterSendRestart() function appropriately.

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

A valid Start or Restart condition must be generated before calling this function. This function does nothing and returns a zero value if the Start or Restart conditions failed before this function was called.

Parameters: uint32 acknNack: Response to received byte.

Response constants	Description
I2C_ACK_DATA	Generates ACK. The master notifies slave that transfer continues.
I2C_NAK_DATA	Prepares to generate NAK. The master will notify slave that transfer is completed.

Return Value: uint8: Byte read from the slave.

Side Effects: This function is blocking and does not exit until the byte_complete bit is set in the I2C_CSR register

uint8 I2C_MasterGetReadBufSize(void)

Description: This function returns the number of bytes that have been transferred with an I2C_MasterReadBuf() function.

Parameters: None

Return Value: uint8: Byte count of the transfer. If the transfer is not yet complete, this function returns the byte count transferred so far.

Side Effects: None

uint8 I2C_MasterGetWriteBufSize(void)

Description: This function returns the number of bytes that have been transferred with an I2C_MasterWriteBuf() function.

Parameters: None

Return Value: uint8: Byte count of the transfer. If the transfer is not yet complete, this function returns the byte count transferred so far.

Side Effects: None

void I2C_MasterClearReadBuf (void)

Description: This function resets the read buffer pointer back to the first byte in the buffer.

Parameters: None

Return Value: None

Side Effects: None

void I2C_MasterClearWriteBuf (void)

Description: This function 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.

Global Variables

Knowledge of these variables is not required for normal operations.

Variable	Description
I2C_initVar	I2C_initVar indicates whether the I²C component 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, then the I2C_Init() function can be called before the I2C_Start() or I2C_Enable() function.
I2C_slAddress	Software address of the I²C slave.

Bootloader Support

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

- **Mode**: Slave
- **Implementation**: Either fixed-function or UDB-based
- **Data Rate**: Must match Host (boot device) data rate.

- **Slave Address**: Must match Host (boot device) selected slave address.
- **Address Match**: Hardware is preferred but not required

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 the Bootloader Protocol Interaction with I2C Communication Component section.

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

Function	Description
I2C_CyBtldrCommStart	Starts the I²C component and enables its interrupt.
I2C_CyBtldrCommStop	Disables the I²C component and disables its interrupt.
I2C_CyBtldrCommReset	Sets read and write I²C buffers to the initial state and resets the slave status.
I2C_CyBtldrCommRead	Allows the caller to read data from the bootloader host. This function manages polling to allow a block of data to be completely received from the host device.
I2C_CyBtldrCommWrite	Allows the caller to write data to the bootloader host. This function manages polling to allow a block of data to be completely sent to the host device.

void I2C_CyBtldrCommStart(void)

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

Parameters: None

Return Value: None

Side Effects: None

void I2C_CyBtldrCommStop(void)

Description: This function disables the I²C component and disables its interrupt.

Parameters: None

Return Value: None

Side Effects: None

void I2C_CyBtldrCommReset(void)

Description: This function sets the read and write I^2C buffers to the initial state and resets the slave status.

Parameters: None

Return Value: None

Side Effects: None

cystatus I2C_CyBtldrCommRead(uint8 pData[], uint16 size, uint16 * count, uint8 timeOut)

Description: This function allows the caller to read data from the bootloader host. The function manages polling to allow a block of data to be completely received from the bootloader host.

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

uint16 size: Number of bytes to be read

uint16 *count: Pointer to the variable to write the number of bytes actually read

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

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

Side Effects: None

cystatus I2C_CyBtldrCommWrite(const uint8 pData[], uint16 size, uint16 * count, uint8 timeOut)

Description: This function allows the caller to write data to the bootloader host. The function manages polling to allow a block of data to be completely sent to the bootloader host.

Parameters: const uint8 pData[]: Pointer to the block of data to be written to the bootloader host

uint16 size: Number of bytes to be written

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

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

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

Side Effects: None

Macro Callbacks

Macro callbacks allow users to execute code from the API files that are automatically generated by PSoC Creator. Refer to the PSoC Creator Help and *Component Author Guide* for the more details.

In order to add code to the macro callback present in the component's generated source files, perform the following:

- Define a macro to signal the presence of a callback (in *cyapicallbacks.h*). This will "uncomment" the function call from the component's source code.
- Write the function declaration (in *cyapicallbacks.h*). This will make this function visible by all the project files.
- Write the function implementation (in any user file).

Callback Function [7]	Associated Macro	Description
I2C_ISR_EntryCallback	I2C_ISR_ENTRY_CALLBACK	Used at the beginning of the I2C_ISR() interrupt handler to perform additional application-specific actions.
I2C_ISR_ExitCallback	I2C_ISR_EXIT_CALLBACK	Used at the end of the I2C_ISR() interrupt handler to perform additional application-specific actions.
I2C_WAKEUP_ISR_EntryCallback	I2C_WAKEUP_ISR_ENTRY_CALLBACK	Used at the beginning of the I2C_WAKEUP_ISR() interrupt handler to perform additional application-specific actions.
I2C_WAKEUP_ISR_ExitCallback	I2C_WAKEUP_ISR_EXIT_CALLBACK	Used at the end of the I2C_WAKEUP_ISR() interrupt handler to perform additional application-specific actions.
I2C_TMOUT_ISR_EntryCallback	I2C_TMOUT_ISR_ENTRY_CALLBACK	Used at the beginning of the I2C_ISR4() interrupt handler to perform additional application-specific actions.
I2C_TMOUT_ISR_ExitCallback	I2C_TMOUT_ISR_EXIT_CALLBACK	Used at the end of the I2C_ISR4() interrupt handler to perform additional application-specific actions.
I2C_SwPrepareReadBuf_Callback	I2C_SW_PREPARE_READ_BUF_CALLBACK	Used in the I2C_ISR() interrupt handler to perform additional application-specific actions.
I2C_SwAddrCompare_EntryCallback	I2C_SW_ADDR_COMPARE_ENTRY_CALLBACK	Used in the I2C_ISR() interrupt handler to perform additional application-specific actions.
I2C_SwAddrCompare_ExitCallback	I2C_SW_ADDR_COMPARE_EXIT_CALLBACK	Used in the I2C_ISR() interrupt handler to perform additional application-specific actions.
I2C_HwPrepareReadBuf_Callback	I2C_HW_PREPARE_READ_BUF_CALLBACK	Used in the I2C_ISR() interrupt handler to perform additional application-specific actions.

7 The callback function name is formed by component function name optionally appended by short explanation and "Callback" suffix.

Callback Function [7]	Associated Macro	Description
I2C_TimeoutReset_Callback	I2C_TIMEOUT_RESET_CALLBACK	Used in the I2C_TimeoutReset() function to perform additional application-specific actions.

Sample Firmware Source Code

PSoC Creator provides many example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

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

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the component. There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator components
- specific deviations – deviations that are applicable only for this component

This section provides information on component-specific deviations. Project deviations are described in the MISRA Compliance section of the *System Reference Guide* along with information on the MISRA compliance verification environment.

The I²C component has the following specific deviations:

MISRA-C: 2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Description of Deviation(s)
10.1	R	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a) it is not a conversion to a wider integer type of the same signedness, or b) the expression is complex, or c) the expression is not constant and is a function argument, or d) the expression is not constant and is a return expression	The library function memcpy has a generic int argument for the number of bytes to be copied. An unsigned 16-bit integer is passed as an argument to this function. This action does not cause any side effects because the number of bytes to copy is always less than 256.

MISRA-C: 2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Description of Deviation(s)
11.5	R	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.	The library function memcpy has pointer to void arguments for the source and destination. A pointer to a constant array is passed as the source argument and the constant qualification is lost. The memcpy function implementation never changes the source and is therefore safe to use with a constant argument.
17.4	R	Array indexing shall be the only allowed form of pointer arithmetic.	The application allocates buffers and sets them up for the component providing the pointer and size. The component uses array indexing operations to access these buffers. The buffer size is checked before accessing the buffers. This implementation is safe as long as the correct buffer size is provided by the application.
19.7	A	A function should be used in preference to a function-like macro.	Deviations with function-like macros to allow for more efficient code. The component incorporates the Fixed Function and UDB implementations. Macros with arguments are used to support these two implementations in a flexible way.

This component has the following embedded component: Clock. Refer to the corresponding component datasheet for information on their MISRA compliance and specific deviations.

API Memory Usage

The component memory usage varies significantly, depending on the compiler, device, number of APIs used and component configuration. The following table provides the memory usage for all APIs available in the given component configuration.

The measurements have been done with associated compiler configured in Release mode with optimization set for Size. For a specific design the map file generated by the compiler can be analyzed to determine the memory usage.

Configuration		PSoC 3 (Keil_PK51)		PSoC 5LP (GCC)	
		Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Slave	UDB	1044	16	1196	21
	Fixed Function	1259	20	1432	25

Configuration		PSoC 3 (Keil_PK51)		PSoC 5LP (GCC)	
		Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Master	UDB	1854	16	2100	20
	Fixed Function	1871	20	2084	24
Multi-Master	UDB	2065	16	2348	20
	Fixed Function	1983	20	2212	24
Multi-Master-Slave	UDB	2961	28	3268	37
	Fixed Function	2854	32	3152	41

Functional Description

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

This component requires that you enable global interrupts because the I²C hardware is interrupt driven. Although 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.

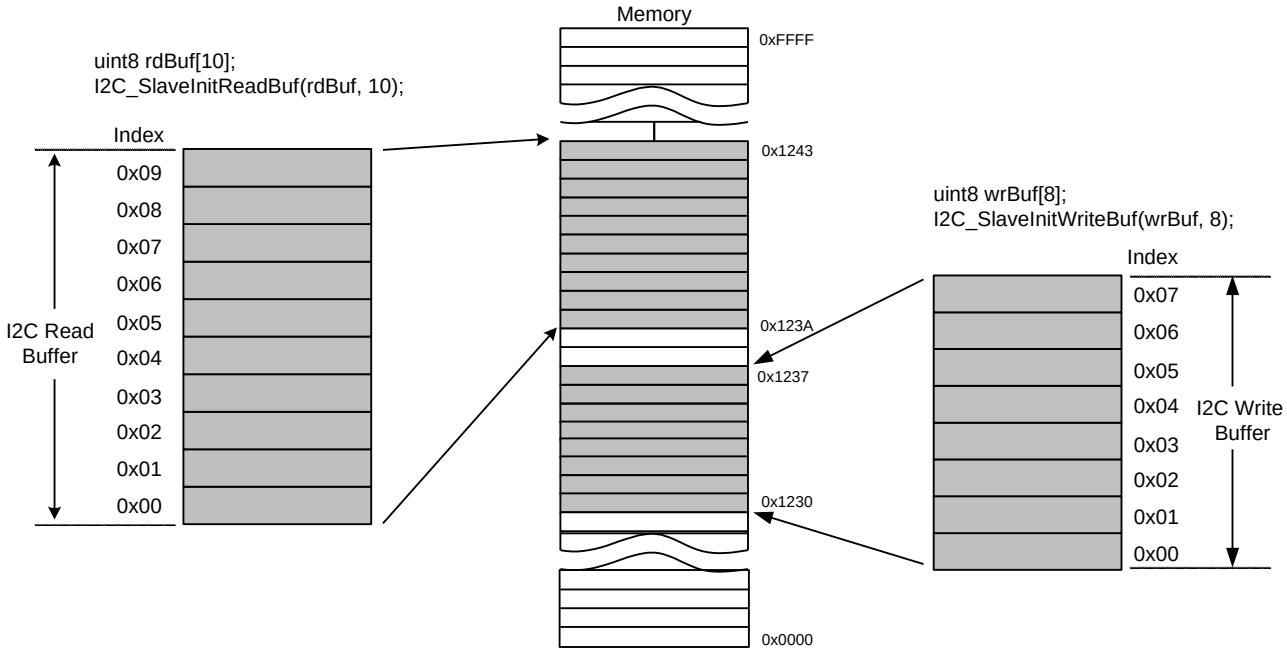
Slave Operation

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. You must instantiate the arrays used for the buffers because they are not automatically generated by the component. You can use the same buffer for both read and write buffers, but you must be careful 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 it should never be larger than the available memory pointed to by the rdBuf or wrBuf pointers.

Figure 2. 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 the I²C master reads or writes the bytes, the index is incremented until the offset is one less than the byteCount. At any time, the number of bytes transferred can be queried by calling either I2C_SlaveGetReadBufSize() or I2C_SlaveGetWriteBufSize() for the read and write buffers, respectively. Reading or writing more bytes than are in the buffers causes an overflow error. The error is set in the slave status byte and can be read with the I2C_SlaveStatus() API.

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

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

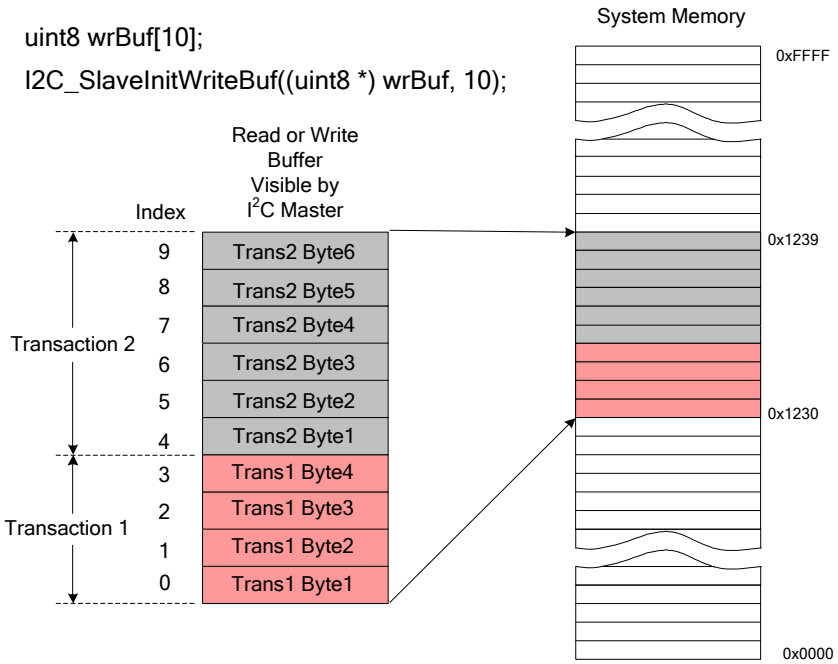
This resets the index back to zero. The next byte the I²C master reads or writes to is the first byte in the array. Before using these clear buffer commands, the data in the arrays should be read or updated.

Multiple reads or writes by the I²C master continue to increment the array index until the clear buffer commands are used or the array index tries to grow beyond the array size. Figure 3 shows an example where an I²C master has executed two write transactions. The first write was four bytes and the second write was six bytes. The sixth byte in the second transaction was ACKed by the slave because buffer has a room to store byte. If the master tried to write a seventh byte for the second transaction or started to write more bytes with a third transaction, each byte would be NAKed and discarded until the buffer is reset.

Using the I2C_SlaveClearWriteBuf() function after the first transaction resets the index back to zero and causes the second transaction to overwrite the data from the first transaction. Make

sure data is not lost by overflowing the buffer. The data in the buffer should be processed by the slave before resetting the buffer index.

Figure 3. System Memory



Both the read and write buffers have four status bits to signal transfer complete, transfer in progress, and buffer overflow. Starting a transfer sets the busy flag. 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 can be set at the same time. The following table shows read and write status flags.

Slave Status Constants	Value	Description
I2C_SSTAT_RD_CMPLT	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 the buffer
I2C_SSTAT_WR_CMPLT	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 the end of the buffer

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

```
uint8 wrBuf[10];
uint8 userArray[10];
uint8 byteCnt;

/* Initialize write buffer before call I2C_Start */
I2C_SlaveInitWriteBuf((uint8 *) wrBuf, 10);

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

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

for(;;)  /* loop forever  */
{
   /* Wait for I2C master to complete a write */
   if(0u != (I2C_SlaveStatus() & I2C_SSTAT_WR_CMPLT))
   {
      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 [8] [9] operation are basically the same, with two exceptions. When operating in multi-master mode, the program should always check the return status for a Start transaction. Another multi-master may already be communicating with another slave. In this case, the program must wait until that communication is completed and the bus becomes free. The program can wait in two ways: generate a Start transaction until the return status indicates success, or check the bus state until the bus becomes free and then generate a Start transaction. The multi-master transaction can be queued if another multi-master generates the Start faster. In this case, the error condition is not returned and a multi-master transaction is generated. This transaction is issued as soon as the bus becomes free.

The second difference is that, in multi-master mode, two masters can start at the same time. If this happens, one of the two masters loses arbitration.

- Automatic multi-master transaction: The component automatically checks for this condition and responds with an error if arbitration was lost. The multi-master transaction is considered complete (appropriate completion status flags are set) when arbitration is lost.
- Manual multi-master transaction: You must check for the return condition after each byte is transferred.

There are two 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 is prefilled with the data to be sent. If data is to be read from the slave, you need to allocate a buffer at least the size of the packet. To write an array of bytes to a slave in automatic mode, use the following function.

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

The slaveAddress variable is a right-justified 7-bit slave address of 0 to 127. The component API automatically appends the write flag to the LSb of the address byte. The second parameter, xferData, points to the array of data to transfer. The cnt parameter is the number of bytes to transfer. The last parameter, mode, determines how the transfer starts and stops. A transaction can 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.

---

8 In fixed-function implementation for PSoC 5LP in master or multi-master mode, if the software sets the Stop condition immediately after the Start condition, the module generates the Stop condition. This happens after the address field (sends 0xFF if data write), and the clock line remains low. To avoid this condition, do not set the Stop condition immediately after Start; transfer at least a byte and set the Stop condition after NAK or ACK.

9 Fixed-function implementation does not support undefined bus conditions. Avoid these conditions, or use the UDB-based implementation instead.

A read operation is almost identical to the write operation. It uses the same parameters with the same constants.

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

Both of these functions return status. See the status table for the I2C_MasterStatus() function return value. Because the read and write transfers complete in the background during the I²C interrupt code, you can use the I2C_MasterStatus() function to determine when the transfer is complete. A code snippet that shows a typical write to a slave follows.

```
I2C_MasterClearStatus(); /* Clear any previous status */
I2C_MasterWriteBuf(0x08, (uint8 *) wrData, 10, I2C_MODE_COMPLETE_XFER);
for(;;)
{
   if(0u != (I2C_MasterStatus() & I2C_MSTAT_WR_CMPLT))
   {

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

      break;
   }
}
```

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

```
status = I2C_MasterSendStart(0x08, I2C_WRITE_XFER_MODE);
if(I2C_MSTR_NO_ERROR == status)   /* 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 following example shows a typical manual read transaction.

```
status = I2C_MasterSendStart(0x08, I2C_READ_XFER_MODE);
if(I2C_MSTR_NO_ERROR == status)   /* 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 work in this mode. The component can be addressed as a slave, but firmware can 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, see the Slave Operation and Master/Multi-Master Operation sections.

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

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

Start of Multi-Master-Slave Transfer

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

- The bus is busy (slave operation is in progress or other traffic is on the bus) before Start generation. The multi-master does not try to generate a Start condition. Slave operation proceeds when the I^2C interrupt is enabled. The I2C_MasterWriteBuf(), I2C_MasterReadBuf(), or I2C_MasterSendStart() call returns the status **I2C_MSTR_BUS_BUSY**.
- The bus is free before Start generation. The multi-master generates a Start condition on the bus and proceeds with operation when the I^2C interrupt is enabled. The

I2C_MasterWriteBuf(), I2C_MasterReadBuf(), or I2C_MasterSendStart() call returns the status **I2C_MSTR_NO_ERROR**.

- The bus is free before Start generation. The multi-master tries to generate a Start but another multi-master addresses the slave before this and the bus becomes busy. The Start condition generation is queued. The slave operation stops at the address stage because of a disabled I²C interrupt. When the I²C interrupt is enabled, the multi-master transaction is aborted from the queue and the slave operation proceeds. The I2C_MasterWriteBuf() or I2C_MasterReadBuf() call does not notice this and returns **I2C_MSTR_NO_ERROR**. The I2C_MasterStatus() returns **I2C_MSTAT_WR_CMPLT** or **I2C_MSTAT_RD_CMPLT** with **I2C_MSTAT_ERR_XFER** (all other error condition bits are cleared) after the multi-master transaction is aborted. The I2C_MasterSendStart() call returns the error status **I2C_MSTR_ERR_ABORT_START_GEN**.

Interrupt Function Operation

```
I2C_MasterWriteBuf();
I2C_MasterReadBuf();
I2C_MasterClearStatus();    /* Clear any previous status */
I2C_DisableInt();           /* Disable interrupt */

status = I2C_MasterWriteBuf(0x08, (uint8 *) wrData, 10, I2C_MODE_COMPLETE_XFER);
/* Try to generate, start. The disabled I2C interrupt halt the transaction on
address stage in case of Slave addressed or Master generates start condition */
I2C_EnableInt();    /* Enable interrupt and proceed Master or Slave transaction */

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

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

Manual Function Operation

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

```
I2C_DisableInt();        /* Disable interrupt */
/* Try to generate start condition */
status = I2C_MasterSendStart(0x08, I2C_WRITE_XFER_MODE);
/* Check if start generation completed without errors */
if (I2C_MSTR_NO_ERROR ==status)
{
   /* Proceed the write operation */
   /* Send array of 5 bytes */
   for (i=0; i<5; i++)
   {
      status = I2C_MasterWriteByte(userArray[i]);
      if (status != I2C_MSTR_NO_ERROR)
      {
         break;
      }
   }
   I2C_MasterSendStop();   /* Send Stop */
}
I2C_EnableInt();            /* Enable interrupt, if it was enabled before */
```

Wakeup on Hardware Address Match

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

- The I²C slave is enabled. Slave or multi-master-slave mode is selected.
- 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[4], SDA P12[5] and I2C1 – SCL P12[0], SDA P12[1].

The I²C component customizer controls these conditions, **except correct pin assignments**.

How it Works

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

Wakeup and Clock Stretching

The I²C slave stretches the clock while exiting sleep mode. All clocks in the system must be restored before continuing the I²C transaction after wakeup. The I²C interrupt remains enabled but interrupt handler is changed before going to sleep. Wakeup on address match triggers I²C interrupt and I²C wakeup flag is set to notify that. Call I2C_Wakeup() function changes interrupt handler to regular I²C and generates interrupt based on I²C wakeup flag to process in-coming transaction. The SCL line remains low after wakeup until I2C_Wake() is called.

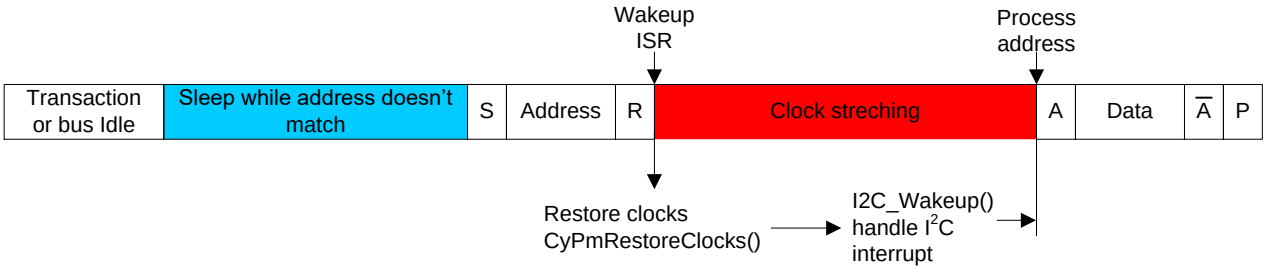
Sample code:

```
I2C_Sleep();            /* Prepare I2C to be able wake up from Sleep mode */

CyPmSaveClocks();       /* Save clocks settings */
CyPmSleep(PM_SLEEP_TIME_NONE, PM_SLEEP_SRC_I2C);
CyPmRestoreClocks();    /* Restore clocks */

I2C_Wakeup();           /* Retuns I2C for operation in Active mode */
…
```

Figure 4. Wakeup of Address Match



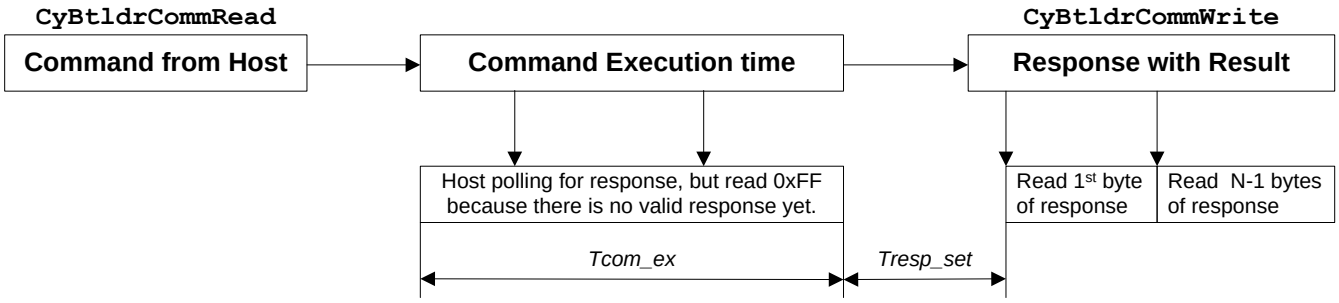
Bootloader Protocol Interaction with I²C Communication Component

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

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

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

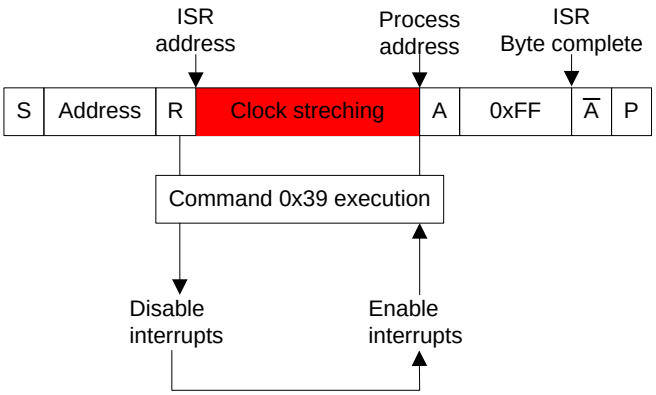
Bootloader process: The host is issued the command with a single write transaction and starts polling for a response. The I²C communication component answers with 0xFF until a valid response is passed by the bootloader. After receiving 0x01, the host must perform another read to get the remaining N –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



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

How to avoid polling: You should measure the command execution time (Tcom_ex) plus the response setup time (Tresp_set) according to the system settings (CPU speed, compiler, compiler optimization level). The host must ask for the response after this time. The command execution time changes across the commands, so you should choose the greater time.

Clock stretching while polling: The I^2C communication component requires that interrupts be enabled while in operation. The Command Program Row (0x39), which writes one row of flash data to the device, requires interrupts to be disabled. Clock stretching occurs if the address is accepted by the I^2C communication component while interrupts are disabled.



How to avoid clock stretching: To avoid clock stretching, measure the Command Program Row (0x39) execution time (Tcom_ex) according to the system settings (CPU speed, compiler, and compiler optimization level). The host must ask for a response after this time.

Internal I²C Bus Multiplexing

Selecting the **External OE buffer** option allows implementing internal I²C bus multiplexing. The tri-state buffers inside component are removed and the bidirectional scl and sda terminals are replaced with inputs terminals sda_i and scl_i and outputs terminals sda_o and scl_o.

Figure 5 shows the configuration of input and output signals using a bidirectional pin (left) and by using a digital input/output pin (right).

The left figure shows the connection of input and output signals using a tri-state buffer (annotated for illustrative purposes) which can be implemented using a bidirectional pin. The pin configuration is provided below the connection image. The same connection must be established for SCL and SDA pins.

Alternatively the same connection can use hardware digital input and output terminals of the pin. The OE signal is tied to logic ‘1’ inside the pin to allow the output signal to drive the pin state. This alternative connection will be used in the examples presented in this section.

The operation of the connection is as follows:

- The scl_i and sda_i signals are each connected to yfb input to sense the bus. This allows receiving the data as well as sense the actual state of the bus. It is required to check if the level which is driven really corresponds to the actual level on the bus. This is important for detection of clock stretching on the bus or lost arbitration by the master.
- The scl_o and sda_o are connected to x output to drive the bus. The output enable of the tri-state buffer is connected to logic ‘1’ to not impact the output. The drive mode “Open Drain, drives low” of the bidirectional pin provides the desired output behavior. Logic low drives the pin low whereas logic high level results in High-Z state on the pin. Then the pull-up resistors on the I²C bus tie level on the pin high when the output state is High-Z.

Figure 5. Input (sda_i / scl_i) and output (sda_o / scl_o) signals connection to the pin

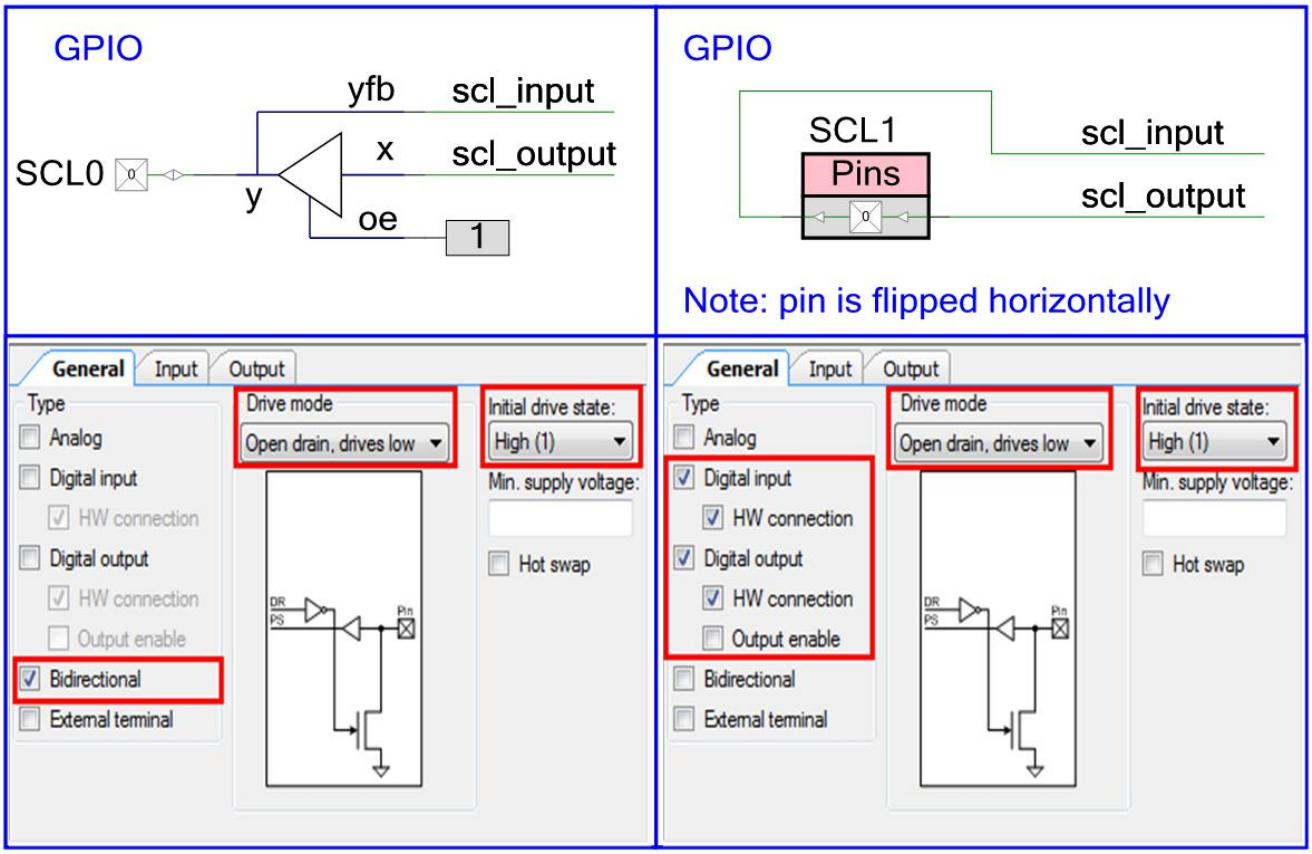


Figure 6 shows an example of 2:1 I²C bus multiplexing. It can easily be extended to support N:1 muxes. The de-multiplexers are used to select which I²C bus scl_o and sda_o will drive the bus. The multiplexers are used to select which I²C bus scl_i and sda_i will sense. The same idea can be used to establish connection to multiple downstream I²C buses. The de-multiplexer assigns all inactive signals to logic '0' that cause lockup of inactive buses. To prevent this from occurring the inactive pins output drivers must be disabled.

Figure 6. I²C Bus Multiplexing inside PSoC

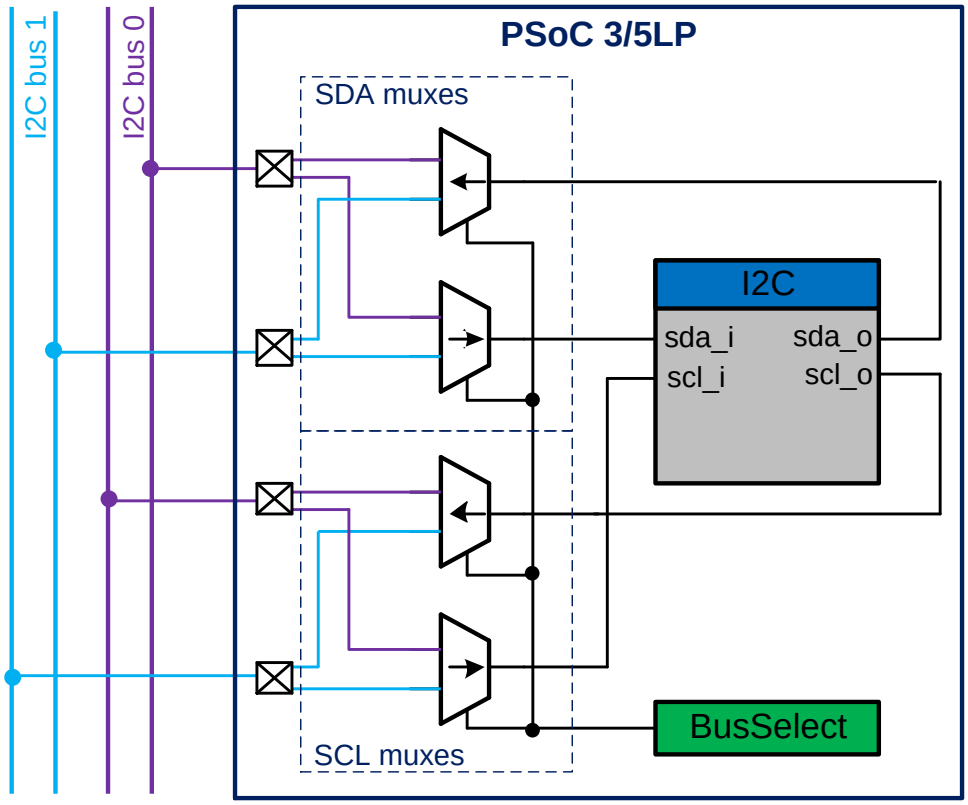


Figure 7 shows the PSoC Creator schematic which implements 2:1 I²C bus multiplexing as described previously. The 8-wire bus is used for signal connection from the I²C component to the pins. The bus signals with even index numbers are used for I²C bus 0 control and signals with odd index numbers are used for I²C bus 1. The following function is an example of bus switching (the drive mode of the pins connected to the inactive bus is changed to High-Z digital to not cause lockup).

```
#define I2C_BUS0_ACTIVE (0u)
#define I2C_BUS1_ACTIVE (1u)
void I2CBusSwtich(uint8 activeBus)
{
    switch(activeBus)
    {
    case I2C_BUS0_ACTIVE:
        /* Set drive mode to Digital High-Z to not imapct BUS 1 */
        SDA1_SetDriveMode(PIN_DM_DIG_HIZ);
        SCL1_SetDriveMode(PIN_DM_DIG_HIZ);

        /* Enable BUS 0 to drive the bus */
        BusSelect_Write(I2C_BUS0_ACTIVE);

        /* Set drive mode to Open Drain Drives Low to drive BUS 0 */
        SDA0_SetDriveMode(PIN_DM_OD_LO);
        SCL0_SetDriveMode(PIN_DM_OD_LO);
        break;

    case I2C_BUS1_ACTIVE:
        /* Set drive mode to Digital High-Z to not imapct BUS 1 */
        SDA0_SetDriveMode(PIN_DM_DIG_HIZ);
        SCL0_SetDriveMode(PIN_DM_DIG_HIZ);

        /* Enable BUS 1 to drive the bus */
        BusSelect_Write(I2C_BUS1_ACTIVE);

        /* Set drive mode to Open Drain Drives Low to drive BUS 0 */
        SDA1_SetDriveMode(PIN_DM_OD_LO);
        SCL1_SetDriveMode(PIN_DM_OD_LO);
        break;

    default:
        /* Do nothing: incorrect bus index */
        break;
    }
}
```

Figure 7. I²C Bus Multiplexing inside PSoC (PSoC Creator schematic)

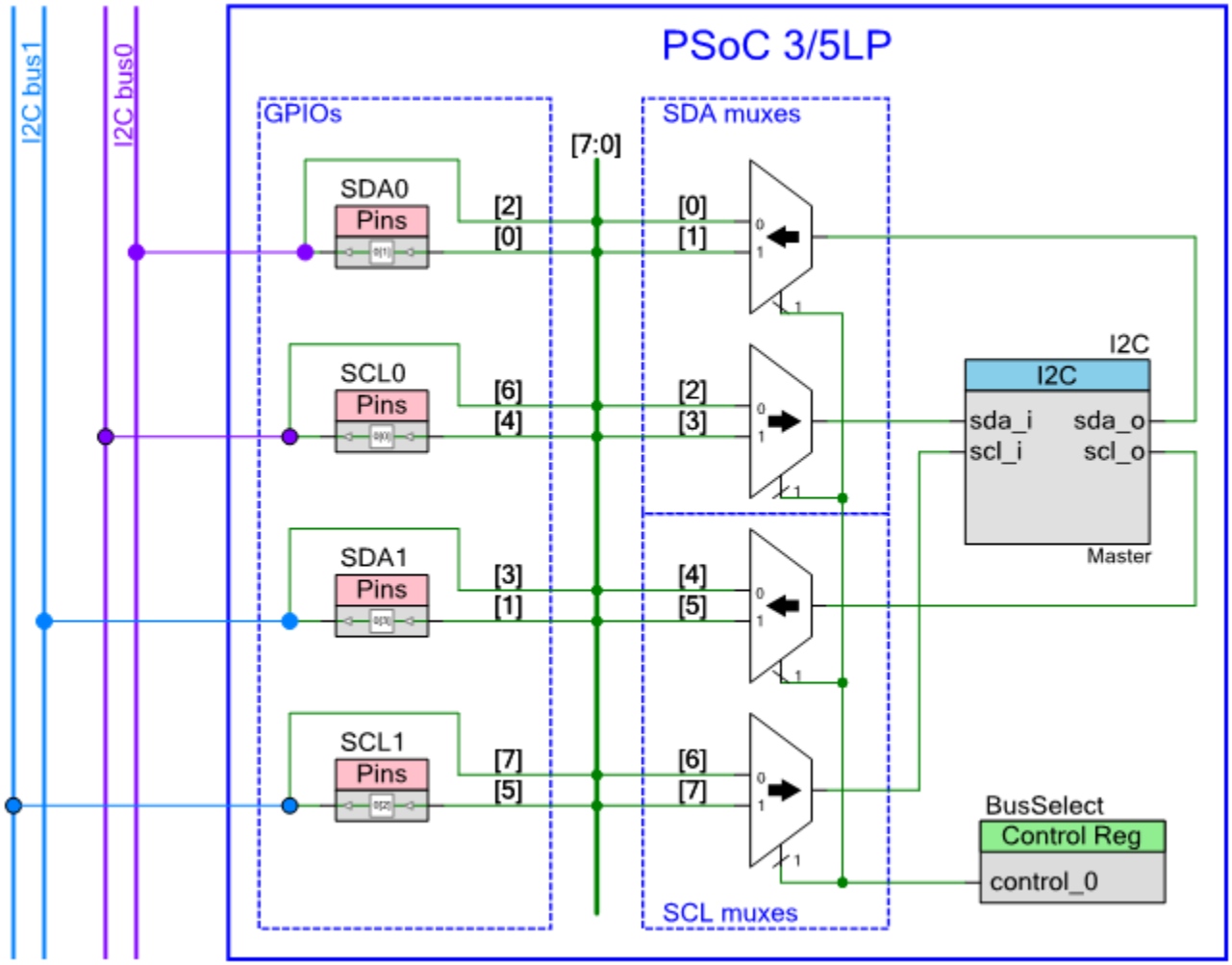
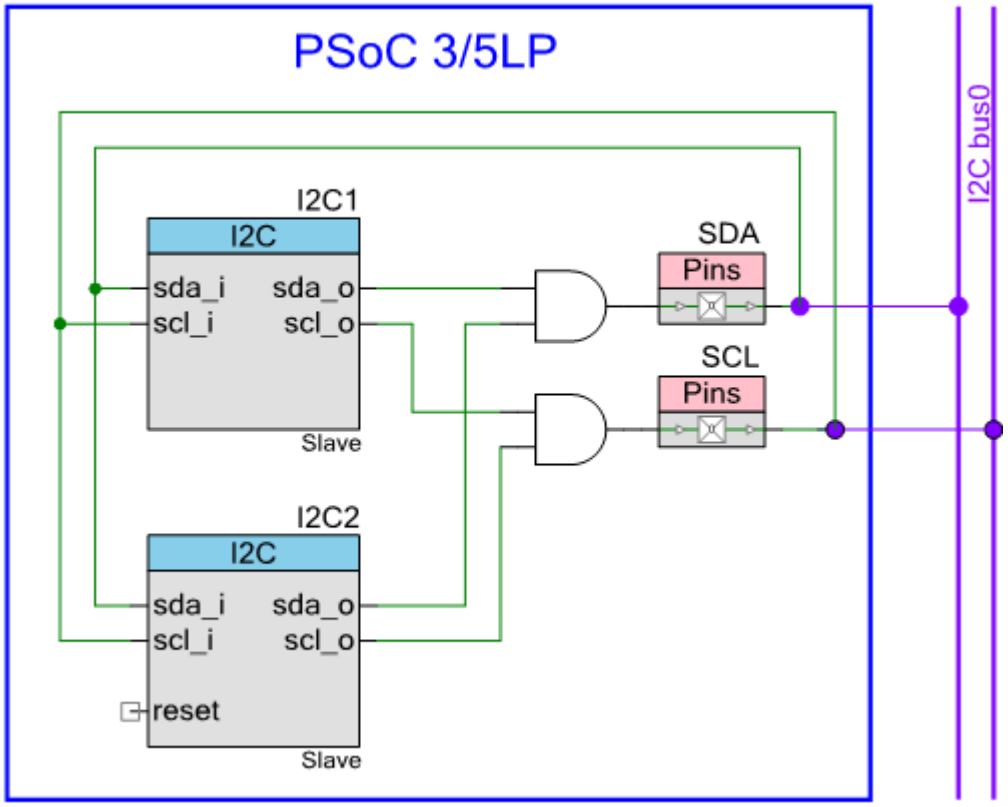


Figure 8 shows the PSoC Creator schematic which implements sharing of single I²C bus with two I²C components inside PSoC. It can be extended to support multiple components to share the same bus. The I2C1 and I2C2 components sense the same inputs and they outputs are ANDed together to drive the bus.

Figure 8. I²C Bus Sharing inside PSoC (PSoC Creator schematic)



Interrupt Service Routine

The interrupt service routine is used by the component code. Do not change it.

The following user sections are provided for slave operations in the *I2C_INT.c* file:

- Custom includes and definitions
- Additional address compare
- Prepare read buffer

There are no user sections provided for master operations.

Registers

The functions provided support the common run-time 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 can be useful for either CPU or DMA use.

The registers available to each of the configurations of the I^2C component are grouped according to the implementation as fixed function or UDB.

Fixed-Function Master/Slave Registers

Refer to the device *Technical Reference Manual (TRM)* for information about I^2C Fixed Function block registers.

UDB Master

The UDB register definitions are derived from the Verilog implementation of I^2C. See the specific mode implementation Verilog for more information about these registers' definitions.

I2C_CFG

The control register is available in the UDB implementation for run-time control of the hardware

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

- start_gen: Set to 1 to generate a Start condition on the bus. This bit must be cleared by firmware before initiating the next transaction.
- 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.
- ack: 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 run-time 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 following definitions.

Bits	7	6	5	4	3	2	1	0
Value	RSVD	lost_arb*	stop_status*	bus_busy	address	master_mode	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: If set, indicates the bus is busy. Data is currently being transmitted or received.
- address: Address detection. If set, indicates that an address byte was sent.
- master_mode: Indicates that a valid Start condition was generated and a hardware device is operating as bus master.
- 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 the last read of this register. In Transmit mode this bit indicates that eight bits of data plus ACK/NAK have been transmitted since the last read. In Receive mode this bit indicates that eight bits of data have been received since the last read of this register.

I2C_INT_MASK

The interrupt mask register is available in the UDB implementation to specify which status bits are enabled as interrupt sources. Any of the status register bits can be enabled as an interrupt source with a one-to-one bit correlation to the status register's bit-field definitions in I2C_CSR.

I2C_DATA

The data register is available in the UDB implementation block for run-time transmission 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 the data in the data register to be transmitted when the master transmits. The Go register forces the data to be received in the data register when the master receives. Any write to this register forces this action, no matter which value is written.

UDB Slave

The UDB register definitions are derived from the Verilog implementation of I²C. See the specific mode implementation Verilog for more information about these registers' definitions.

I2C_CFG

The control register is available in the UDB implementation for run-time control of the hardware

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

- nak: If set, used to NAK 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 and cleared = receive.
- en_slave: Set to 1 to enable the slave functionality.

I2C_CSR

The status register is available in the UDB implementation for run-time 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 the last read of this register. In transmit mode this bit indicates that eight bits of data plus ACK/NAK have been transmitted since the last read. In Receive mode this bit indicates that eight bits of data have been received since the last read of this register.

I2C_INT_MASK

The interrupt mask register is available in the UDB implementation to specify which status bits are enabled as interrupt sources. Any of the status register bits can be enabled as an interrupt source with a one-to-one bit correlation to the status register bit-field definitions in the I2C_CSR register. Two interrupt sources are used during operation: byte_complete and stop.

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 run-time transmission 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 the data register to receive data when the master receives. Any write to this register forces this action, no matter which value is written.

Component Debug Window

PSoC Creator allows you to view debug information about components in your design. Each component window lists the memory and registers for the instance. For detailed hardware registers descriptions, refer to the appropriate device *Technical Reference Manual (TRM)*. For detailed UDB registers descriptions used in the component, refer to the Registers section of this datasheet.

To open the Component Debug window:

1. Make sure the debugger is running or in break mode.
2. Choose **Windows > Components…** from the **Debug** menu.
3. In the Component Window Selector dialog, select the component instances to view and click **OK**.

The selected Component Debug window(s) will open within the debugger framework. Refer to the "Component Debug Window" topic in the PSoC Creator Help for more information.

Resources

The fixed I²C block and one interrupt are used for the fixed-function implementation.

The UDB version of the component uses the following resources:

Configuration	Resource Type					
	Datapath Cells	Macrocells	Status Cells	Control Cells	DMA Channels	Interrupts
Slave	1	25	1	2	–	1
Master	2	33	1	1	–	1
Multi-Master	2	36	1	1	–	1
Multi-Master-Slave	2	65	1	2	–	1

DC and AC Electrical Characteristics

Specifications are valid for –40 °C ≤ T_A ≤ 85 °C and T_J ≤ 100 °C, except where noted.
Specifications are valid for 1.71 V to 5.5 V, except where noted.

DC Characteristics (FF Implementation)

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

DC Characteristics (UDB Implementation)

Parameter	Description		Min	Typ[10]	Max	Unit[11]
$I_{DD(Slave)}$	Component current consumption (Slave)	Standard mode	–	200	–	µA
		Fast mode	–	290	–	µA
		Fast mode plus	–	335	–	µA
$I_{DD(Master)}$	Component current consumption (Master)	Standard mode	–	210	–	µA
		Fast mode	–	305	–	µA

10 Device IO and clock distribution current not included. The values are at 25 °C.

11 Current consumption is specified with respect to the incoming component clock.

Parameter	Description			Min	Typ[10]	Max	Unit[11]
			Fast mode plus	–	465	–	µA
$I_{DD(Multi-Master)}$	Component current consumption (Multi-Master)		Standard mode	–	215	–	µA
			Fast mode	–	320	–	µA
			Fast mode plus	–	515	–	µA
$I_{DD(Multi-Master-Slave)}$	Component current consumption (Multi-Master-Slave)	Slave operation	Standard mode	–	200	–	µA
			Fast mode	–	290	–	µA
			Fast mode plus	–	335	–	µA
		Multi-Master operation	Standard mode	–	215	–	µA
			Fast mode	–	320	–	µA
			Fast mode plus	–	515	–	µA

AC Characteristics (FF Implementation)

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

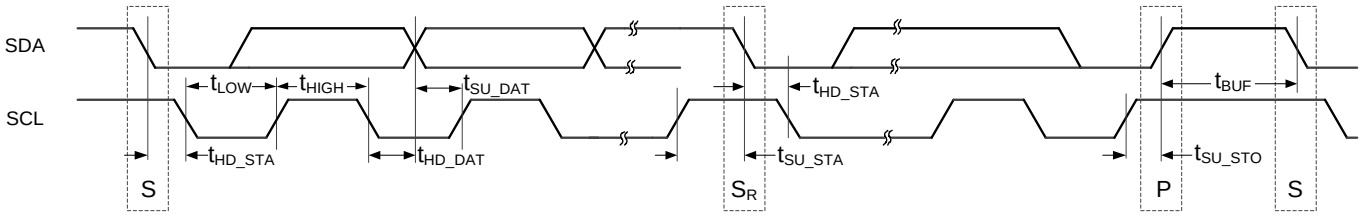
AC Characteristics (UDB Implementation)

Parameter	Description	Min	Typ	Max	Unit
f_{SCL}	SCL clock frequency	– – –	– – –	100 400 1000	kHz
f_{CLOCK}	Component input clock frequency	–	16 × f_{SCL}	–	kHz
t_{RESET}	Reset pulse width	–	2	–	t_{CY_clock}[12]
t_{LOW}	Low period of the SCL clock	4.7 1.3 0.5	– – –	– – –	µs
t_{HIGH}	High period of the SCL clock	4.0 0.6 0.26	– – –	– – –	µs

12 $t_{CY_clock} = 1/f_{CLOCK}$. This is the cycle time of one clock period.

Parameter	Description	Min	Typ	Max	Unit
t_{HD_STA}	Hold time (repeated) start condition	4.0 0.6 0.26	– – –	– – –	µs
t_{SU_STA}	Setup time for a repeated start condition	4.7 0.6 0.26	– – –	– – –	µs
t_{HD_DAT}	Data hold time	5.0 – –	– – –	– – –	µs
t_{SU_DAT}[13]	Data setup time	250 100 50	– – –	– – –	ns
t_{SU_STO}	Setup time for stop condition	4.0 0.6 0.26	– – –	– – –	µs
t_{BUF}	Bus free time between a stop and start condition	4.7 1.3 0.5	– – –	– – –	µs

Figure 9. Data Transition Timing Diagram



13 Refer to component errata section Cypress ID 195462.

Component Errata

This section lists known problems with the component.

Cypress ID	Component Version	Problem	Workaround
127428	All	The I²C Multi-Master-Slave implemented on UDBs fails to operate as master when Address decode is Software.	Open the Configure dialog General tab and change Implementation to Fixed function or Address decode to Hardware.
195462	All	The I²C slave control logic implemented on UDBs releases SCL and SDA lines simultaneously after SCL line has been stretched by the slave. As a result, this can cause violation of $t_{SU;DAT}$ parameter and the master samples current SDA line state wrong. This typically occurs when the slave generates ACK to the address or data (master writes).	There is no workaround. However, the master and slave can communicate successfully because SCL line raises high with RC delay provided by I²C bus. This delay can prove enough setup time for the master to sample current SDA line state.
DRIVERS-4223	3.40, 3.50	When the I2C Component is used in I2C Master mode with Fixed function implementation, the I2C_Stop() API does not disable the master.	This issue is due to an error in the I2C_Stop() function definition in the Generated Source. The workaround is to manually modify the Generated Source. Follow the steps mentioned in KBA231344.

Component Changes

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

Version	Description of Changes	Reason for Changes / Impact
3.50.d	Datasheet update.	Added errata item for DRIVERS-4223.
3.50.c	Minor datasheet edits.	
3.50.b	Datasheet update.	Added a note about threshold selection for I²C pins.
3.50.a	Datasheet update.	Added Macro Callbacks section.
3.50	Addressed an issue where the I2C_MasterSendStop() function in v3.40 of the component never returns if arbitration is lost when I²C multi-master executes read transaction.	This issue is resolved in v3.50.
	Addressed an issue where the I²C master (UDB-based) manual functions can never return if they are interrupted by an ISR.	This issue is resolved in v3.50.

Version	Description of Changes	Reason for Changes / Impact
3.40	Changed execution flow in the I2C_MasterWriteBuf() and I2C_MasterReadBuf() functions to do all preparation for transaction execution before starting it and enabling the interrupt.	The I2C_MasterWriteBuf() and I2C_MasterReadBuf() can clear the write or read completion event of an initiated transaction.
	Fixed conditional check which defines that stop condition has been generated by the master.	The I2C_MasterSendStop() can cause lockup if its execution is interrupted. Only applicable for I²C master (implemented on UDBs).
	Add protection from the component interruption to the following APIs: I2C_SlaveInitReadBuf() I2C_SlaveInitWriteBuf() I2C_SlaveClearReadStatus() I2C_SlaveClearWriteStatus() I2C_MasterClearStatus() I2C_MasterStatus() I2C_MasterClearWriteBuf() I2C_MasterClearReadBuf()	I²C operations executed by the listed functions are atomic.
	Datasheet updates.	Updated Internal I2C Bus Multiplexing section. Updated External Electrical Connections section and placed it closer to the top of the document. Improved description of the APIs (no functional changes). Added Component Debug Window and Component Errata sections. Reduced the number of global variables shown in the Global Variables section.
3.30.b	Edited datasheet to add Component Errata section.	Document that the component was changed, but there is no impact to designs.
3.30.a	Edited the datasheet.	Updated the diagram in When to Use an I²C Component section. Clarified that the fixed-function implementation uses one interrupt.
3.30	Added MISRA Compliance section.	The component has specific deviations described.
	Fixed incorrect behavior of the master manual APIs when lost arbitration occurs.	On a lost arbitration event, the master manual APIs returned the proper status but did not release the I²C bus. Code was added to release the bus when the lost arbitration event takes place.
	Added footnote about non-compliant with the NXP I²C specification in some areas.	Documentation enhancement.

Version	Description of Changes	Reason for Changes / Impact
	Minor datasheet edits and updates.	
3.20	New feature was added. Removed the internal OE buffer and exposed the input and output terminals.	This feature allows I²C buses multiplexing inside PSoC.
	Changed the control flow of the wake up sequence to avoid disabling the I²C interrupt.	PSoC 5LP requires an I²C interrupt to be enabled in order to wake up the device at the event of an address match.
	Moved the Stop interrupt to be handled at the start of a new transaction.	The Stop interrupt was occurring while the next transaction was beginning. This caused the interrupt code to get into an improper state and it did not catch the Stop interrupt. This issue applied to only the slave devices.
3.10	Added support PSoC 5LP.	
	Fixed wrong SDA behavior (the line drives low) after address byte was received.	When master generates transaction with slave address expected to be NAKed, the wrong Stop detection is possible. The issue only appears in Slave mode with Software Address Decode and UDB-based implementation.
3.1.a	Documentation change describing how the effective data rate will vary.	For data rates above 400 kbps, the effective clock rate can vary.
	Documentation change describing the difference between master and multi-master modes.	When operating in multi-master mode there are special considerations to take into account to handle correct interaction with other masters.
3.1	Changed the definition from I2C_SSTAT_RD_CMPT to I2C_SSTAT_RD_CMPLT Changed the definition from I2C_SSTAT_WR_CMPT to I2C_SSTAT_WR_CMPLT	To comply with the master definition of read and write complete flags. The component supports both definitions, but the I2C_SSTAT_RD_CMPT and I2C_SSTAT_WR_CMPT will become obsolete.
	Added the CYREENTRANT keyword to all APIs when they are included in the .cyre file.	Not all APIs are truly reentrant. Comments in the component API source files indicate which functions are not candidates. This change is required to eliminate compiler warnings for functions that are not reentrant used in a safe way: protected from concurrent calls by flags or Critical Sections.
3.0.a	Minor datasheet edits and updates	
3.0	Changed customizer appearance	More intuitive and easy to use.
	Added the UDB clock tolerance setting.	Avoids the appearance of clock warning for many configurations.

Version	Description of Changes	Reason for Changes / Impact
	The component in FF implantation with Enable from Sleep option restores configuration correctly after exit hibernate.	Fix component behavior in hibernate mode.
	The I²C interrupt is enabled after I2C_Start() is called.	No errors appear when the user forgets to enable interrupt after I2C_Start() in slave mode.
	Added support of internal clock for UDB implementation.	Functionality enhancement.
	Removed functions I2C_SlaveGetWriteByte() and I2C_SlavePutReadByte()	These functions are not usable.
2.20	Added bootloader communication support to UDB-based implementation of component.	Allows more than one I²C component that supports bootloading in the design. This can be used with the custom bootloader feature included with cy_boot v2.21.
	Fixed misplaced start condition detection during transaction due zero data hold time.	The slave operates correctly with zero data hold time from the master.
2.10	Added 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.
	Changed I²C bootloader communication component behavior to suppress clock stretching on read.	I²C bootloader communication component holds SCL low forever if a read command is issued before the start boot process.
	Added characterization data to datasheet.	
	Minor datasheet edits and updates	
2.0.a	Moved the 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.
	Updated the component to support Production PSoC 3 and above. Updated the Configure dialog:	New requirement to support the Production PSoC 3 device, thus a new 2.0 version was created. Version 1.xx supports PSoC 3 ES2 and PSoC 5 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.
	Updated the datasheet.	Updated the Parameters and Setup, Clock Selection, and Resources sections 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 (an Infineon company), 2015-2021. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.

