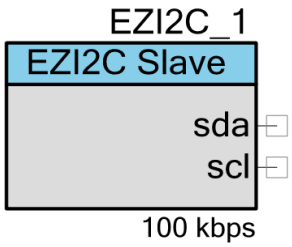
# EZI2C Slave
**1.70**




## Features

- Industry standard Philips I²C bus interface
- Emulates common I²C EEPROM interface
- Only two pins (SDA and SCL) required to interface to I²C bus
- Standard data rates of 50/100/400/1000 kbps
- High level APIs require minimal user programming
- Supports one or two address decoding with independent memory buffers
- Memory buffers provide configurable Read/Write and Read Only regions

## General Description

The EZI2C Slave component implements an I²C register-based slave device. 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 EZI2C Slave supports standard data rates up to 1000 kbps and is compatible with multiple devices on the same bus.

The EZI2C Slave is a unique implementation of an I²C slave in that all communication between the master and slave is handled in the ISR (Interrupt Service Routine) and requires no interaction with the main program flow. The interface appears as shared memory between the master and slave. Once the EZI2C_Start() function is executed, there is little need to interact with the API.

### When to Use an EZI2C Slave

Use this component when you want a shared memory model between the I²C Slave and I²C Master. You may define the EZI2C Slave buffers as any variable, array, or structure in your code without worrying about the I²C protocol. The I²C master may view any of the variables in this buffer and modify the variables defined by the EZI2C_SetBuffer1() or EZI2C_SetBuffer2() functions.

# Input/Output Connections

This section describes the various input and output connections for EZI2C Slave.

### sda – In/Out

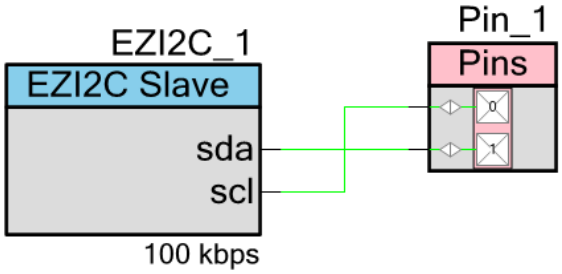
Serial data (SDA) is the $I^2C$ data signal. It is a bidirectional data signal used to transmit or receive all bus data.

### scl – In/Out

Serial clock (SCL) is the master generated $I^2C$ clock. Although the slave never generates the clock signal, it may hold it low stalling the bus until it is ready to send data or NAK/ACK the latest data or address.

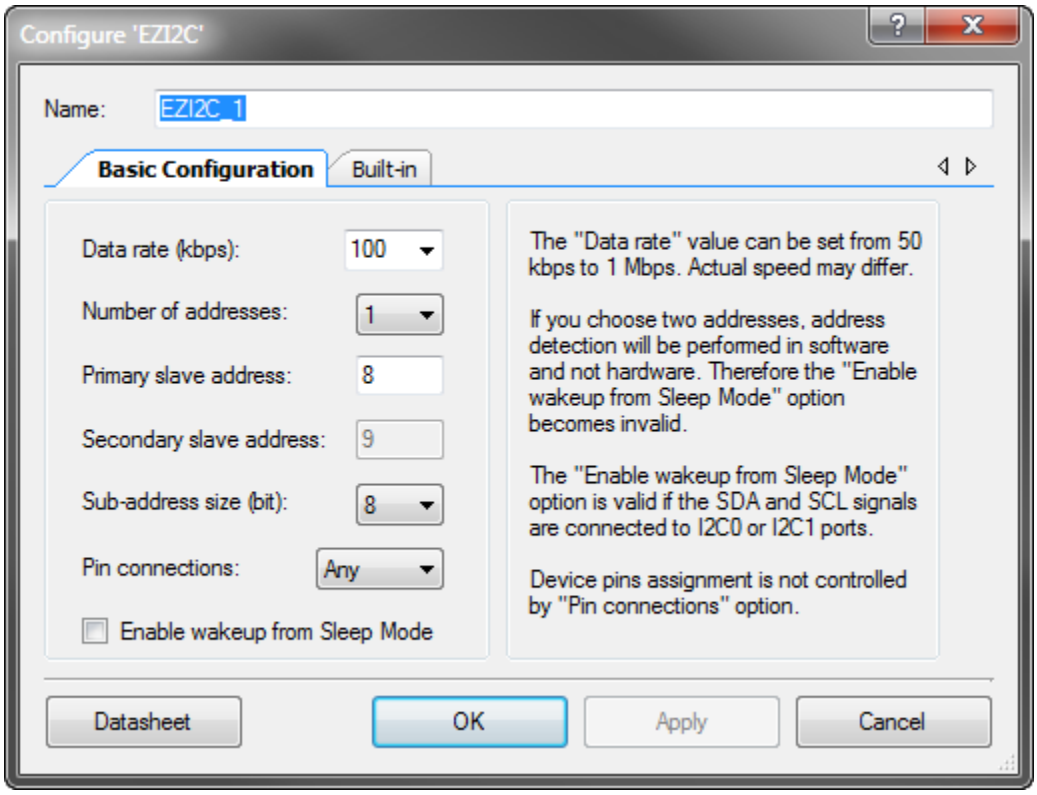
# Schematic Macro Information

The default EZI2C Slave in the Component Catalog is a schematic macro using an EZI2C Slave component with default settings. The EZI2C Slave component is connected to a Pins component, which is configured as an SIO pair.



# Component Parameters

Drag an EZI2C component onto your design and double-click it to open the **Configure** dialog.




The EZI2C component provides the following parameters.

## Data rate

This parameter is used to set the I²C data rate value up to 1000 kbps; the actual rate may differ, based on available clock speed and divider range. The standard data rates are 50, 100 (default), 400, and 1000 kbps.

## Number of addresses

This option determines whether **1** (default) or **2** independent I²C slave addresses are recognized. If two addresses are recognized, address detection will be performed in software, not hardware; therefore, the **Enable wakeup from Sleep Mode** option is not available.

## Primary slave address

This is the primary I²C slave address (default is **8**). This value can be entered in decimal or hexadecimal format. For hexadecimal, type '0x' before the number. This address is the 7-bit right-justified slave address and does not include the R/W bit.

### Secondary slave address

This is the secondary $I^2C$ slave address (default is **9**). This value can be entered in decimal and hexadecimal format. For hexadecimal, type '0x' before the number. This second address is only valid when the **Number of addresses** parameter is set to **2**. The primary and secondary slave addresses must be different. This address is the 7-bit right-justified slave address and does not include the R/W bit.

### Sub-address Size

This option determines what range of data can be accessed. You can select a sub-address of **8** bits (default) or **16** bits. If you use an address size of 8 bits, the master can only access data offsets between 0 and 255. You may also select a sub-address size of 16 bits. That will allow the $I^2C$ master to access data arrays of up to 65,535 bytes at each slave address.

### Pin connections

This parameter determines which type of pin to use for SDA and SCL signal connections. This option is supplemental for the **Enable wakeup from Sleep mode** option and is available only if single $I^2C$ address is selected in the **Number of addresses** option. There are three possible values: **Any**, **I2C0**, and **I2C1**. The default is **Any**.

**Any** means general-purpose I/O (GPIO). If **Enable wakeup from Sleep Mode** is not required, **Any** should be used for SDA and SCL. If you need **Enable wakeup from Sleep Mode**, you must use the pairs of pins I2C0 (P12[4], P12[5]) or I2C1 (P12[0], P12[1]), which allows you to configure the device for wakeup on $I^2C$ address match.

### Enable wakeup from Sleep Mode

This parameter allows the device to be awakened from sleep mode on slave address match. This option is disabled by default. The wake up on address match option is valid if a single $I^2C$ address is selected and the SDA, and SCL signals are connected to SIO ports (pin pairs I2C0 or I2C1). **Enable wakeup from Sleep mode** will be supported in PSoC 3 and PSoC 5LP.

## Clock Selection

The clock is tied to the system bus clock and cannot be changed by the user.

# Application Programming Interface

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

By default, PSoC Creator assigns the instance name “EZI2C_1” to the first instance of a component in a given design. You can rename it to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “EZI2C.”

| Function | Description |
|---|---|
| EZI2C_Start() | Starts responding to I²C traffic. Enables interrupt. |
| EZI2C_Stop() | Stops responding to I²C traffic. Disables interrupt. |
| EZI2C_EnableInt() | Enables interrupt, which is required for most I²C operations. |
| EZI2C_DisableInt() | Disables interrupt. The EZI2C_Stop() API does this automatically. |
| EZI2C_SetAddress1() | Sets the primary I²C address. |
| EZI2C_GetAddress1() | Returns the primary I²C address. |
| EZI2C_SetBuffer1() | Sets the buffer pointer for the primary I²C. |
| EZI2C_GetActivity(void) | Checks component activity status. |
| EZI2C_Sleep() | Stops I²C operation and saves I²C configuration. Disables interrupt. |
| EZI2C_Wakeup() | Restores I²C configuration and starts I²C operation. Enables interrupt. |
| EZI2C_Init() | Initializes I²C registers with initial values provided from customizer. |
| EZI2C_Enable() | Activates the hardware and begins component operation. |
| EZI2C_SaveConfig() | Saves the current user configuration of the EZI2C component. |
| EZI2C_RestoreConfig() | Restores nonretention I²C registers. |

## Optional Second Address API

These commands are present only if two I²C addresses are enabled.

| Function | Description |
|---|---|
| EZI2C_SetAddress2() | Sets the secondary I²C address. |
| EZI2C_GetAddress2() | Returns the secondary I²C address. |
| EZI2C_SetBuffer2() | Sets the buffer pointer for the secondary I²C. |

## Optional Sleep/Wake modes

These functions are only available if a single address is used, the SCL and SDA signals are routed to the SIO pins, and **Enable wakeup from Sleep Mode** is selected.

| Function | Description |
|---|---|
| EZI2C_SlaveSetSleepMode() | Enables EZI2C sleep address decode and saves $I^2C$ configuration. Disables interrupt. |
| EZI2C_SlaveSetWakeMode() | Disables EZI2C sleep address decode and restores $I^2C$ configuration and starts $I^2C$ operation. Enables interrupt. |

## Global Variables

Knowledge of these variables is not required for normal operations.

| Function | Description |
|---|---|
| EZI2C_initVar | Indicates whether the EZI2C has been initialized. The variable is initialized to 0 and set to 1 the first time EZI2C_Start() is called. This allows the component to restart without reinitialization after the first call to the EZI2C_Start() routine.<br>If reinitialization of the component is required the variable should be set to 0 before the EZI2C_Start() routine is called. Alternatively, the EZI2C can be reinitialized by calling the EZI2C_Init() and EZI2C_Enable() functions. |
| EZI2C_dataPtrS1 | Stores pointer to the data exposed to an $I^2C$ master for the first slave address. |
| EZI2C_rwOffsetS1 | Stores offset for read and write operations. It is set at each write sequence of the first slave address. |
| EZI2C_rwIndexS1 | Stores pointer to the next value to be read or written for the first slave address. |
| EZI2C_wrProtectS1 | Stores offset where data is read only for the first slave address. |
| EZI2C_bufSizeS1 | Stores size of data array exposed to an $I^2C$ master for the first slave address. |
| EZI2C_dataPtrS2 | Stores pointer to the data exposed to an $I^2C$ master for the second slave address. |
| EZI2C_rwOffsetS2 | Stores offset for read and write operations, is set at each write sequence of the second slave device. |
| EZI2C_rwIndexS2 | Stores pointer to the next value to be read or written for the second slave address. |
| EZI2C_wrProtectS2 | Stores offset where data is read only for the second slave address. |
| EZI2C_bufSizeS2 | Stores size of data array exposed to an $I^2C$ master for the second slave address. |
| EZI2C_curState | Stores current state of an $I^2C$ state machine. |
| EZI2C_curStatus | Stores current status of the component. |

## void EZI2C_Start(void)

**Description:** This is the preferred method to begin component operation. EZI2C_Start() sets the initVar variable, calls the EZI2C_Init() function, and then calls the EZI2C_Enable() function. It must be executed before $I^2C$ bus operation.

After EZI2C_Start() calls EZI2C_Enable(), EZI2C_Enable() calls the EZI2C_EnableInt(), which enables the $I^2C$ interrupt.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

## void EZI2C_Stop(void)

**Description:** Disables $I^2C$ hardware and disables $I^2C$ interrupt. Disables Active mode power template bits or clock gating as appropriate.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

## void EZI2C_EnableInt(void)

**Description:** Enables $I^2C$ interrupt. Interrupts are required for most operations. Called when calling the EZI2C_Start() API.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

## void EZI2C_DisableInt(void)

**Description:** Disables $I^2C$ interrupts. This function is not normally required because the 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, the $I^2C$ bus may lock up.

## void EZI2C_SetAddress1(uint8 address)

**Description:** This function sets the I²C address for the primary memory buffer. This value can be any value between 0 and 127.

**Parameters:** address: The 7-bit slave address between 0 and 127. The address is right justified and does not include the R/W bit.

**Return Value:** None

**Side Effects:** None

## uint8 EZI2C_GetAddress1(void)

**Description:** Returns the I²C slave address for the primary memory buffer.

**Parameters:** None

**Return Value:** The same I²C slave address set by SetAddress1 or the default I²C address.

**Side Effects:** None

## void EZI2C_SetBuffer1(uint16 bufSize, uint16 rwBoundry, void * dataPtr)

**Description:** This function sets the buffer pointer, size, and read/write area for the slave data. This is the data that is exposed to the I²C Master.

**Parameters:** bufSize: Size of the buffer in bytes.

rwBoundry: Sets how many bytes are writable in the beginning of the buffer. This value must be less than or equal to the buffer size. Data located at offset rwBoundry and greater are read only.

dataPtr: Pointer to the data buffer.

**Return Value:** None

**Side Effects:** The EZI2C_Start() must be called before this function.

## uint8 EZI2C_GetActivity(void)

**Description:** This function returns a nonzero value if an I²C read or write cycle has occurred since the last time this function was called. The activity flag resets to zero at the end of this function call.

The Read and Write busy flags are cleared when read, but the “BUSY” flag is only cleared by an I²C Stop.

**Parameters:** A nonzero value is returned if activity is detected.

**Return Value:** Status of I²C activity.

| Constant | Description |
|---|---|
| EZI2C_STATUS_READ1 | Set if Read sequence is detected for first address. Cleared when status is read. |
| EZI2C_STATUS_WRITE1 | Set if Write sequence is detected for first address. Cleared when status is read. |
| EZI2C_STATUS_READ2 | Set if Read sequence is detected for second address (if enabled). Cleared when status is read. |
| EZI2C_STATUS_WRITE2 | Set if Write sequence is detected for second address (if enabled). Cleared when status is read. |
| EZI2C_STATUS_BUSY | Set if Start is detected. Cleared when stop is detected. |
| EZI2C_STATUS_ERR | Set when I²C hardware error is detected. Cleared when status is read. |

**Side Effects:** None

## void EZI2C_SetAddress2(uint8 address)

**Description:** Sets the I²C slave address for the secondary memory buffer. This value may be any value between 0 and 127. This function is only provided if two I²C addresses have been selected in the user parameters.

**Parameters:** address: The 7-bit slave address between 0 and 127. The address is right justified and does not include the R/W bit.

**Return Value:** None

**Side Effects:** None

## uint8 EZI2C_GetAddress2(void)

**Description:** Returns the I²C slave address for the secondary memory buffer. This function is only provided if two I²C addresses have been selected in the user parameters.

**Parameters:** None

**Return Value:** The same I²C slave address set by SetAddress2 or the default I²C address.

**Side Effects:** None

## void EZI2C_SetBuffer2(uint16 bufSize, uint16 rwBoundry, void * dataPtr)

**Description:** This function sets the buffer pointer, size, and read/write area for the second slave data. This is the data that is exposed to the I²C Master for the second I²C address. This function is only provided if two I²C addresses have been selected in the user parameters.

**Parameters:** bufSize: Size of the buffer exposed to the I²C master.

rwBoundry: Sets how many bytes are readable and writable by the I²C master. This value must be less than or equal to the buffer size. Data located at offset rwBoundry and greater are read only.

dataPtr: This is a pointer to the data array or structure that is used for the I²C data buffer.

**Return Value:** None

**Side Effects:** The EZI2C_Start() must be called before this function.

## void EZI2C_SlaveSetSleepMode(void)

**Description:** This is the preferred API to prepare the component for sleep, if **Enable wakeup from Sleep Mode** is selected. This API enables I²C sleep address decode. It will wait until all I²C traffic has stopped before completing. All subsequent I²C traffic will be NAKed until the device is put to sleep.

**Parameters:** None

**Return Value:** None

**Side Effects:** The I²C interrupt will be disabled if Wake up from Sleep mode option is enabled (only for PSoC 3 and PSoC 5LP silicon).

## void EZI2C_SlaveSetWakeMode(void)

**Description:** This is the preferred API to restore the component to the state when EZI2C_SlaveSetSleepMode() was called. It disables the sleep EZI2C slave and re-enables the run time EZI2C. It should be called just after awaking from sleep. This function is only provided if a single I²C address is used.

**Parameters:** None

**Return Value:** None

**Side Effects:** The I²C interrupt will be enabled if Wake up from Sleep mode option is enabled (only for PSoC 3 and PSoC 5LP silicon).

## void EZI2C_Sleep(void)

**Description:** This is the preferred API to prepare the component for sleep if **Enable wakeup from Sleep Mode** is not selected. The EZI2C_Sleep() API saves the current component state. Then it calls the EZI2C_Stop() function and calls EZI2C_SaveConfig() to save the hardware configuration.

Call the EZI2C_Sleep() function before calling the CyPmSleep() or the CyPmHibernate() function. Refer to the PSoC Creator *System Reference Guide* for more information about power-management functions.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

## void EZI2C_Wakeup(void)

**Description:** This is the preferred API to restore the component to the state when EZI2C_Sleep() was called. The EZI2C_Wakeup() function calls the EZI2C_RestoreConfig() function to restore the hardware configuration. If the component was enabled before the EZI2C_Sleep() function was called, the EZI2C_Wakeup() function will also re-enable the component.

**Parameters:** None

**Return Value:** None

**Side Effects:** Calling this function before EZI2C_SaveConfig() or EZI2C_Sleep() may produce unexpected behavior.

## void EZI2C_Init(void)

**Description:** Initializes or restores the component according to the customizer **Configure** dialog settings. It is not necessary to call EZI2C_Init() because the EZI2C_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 EZI2C_Enable(void)

**Description:** Activates the hardware and begins component operation. It is not necessary to call EZI2C_Enable() because the EZI2C_Start() API calls this function, which is the preferred method to begin component operation. Calls EZI2C_EnableInt() to enable the $I^2C$ interrupt.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

### void EZI2C_SaveConfig(void)

**Description:** This function saves the component configuration and nonretention registers. It also saves the current component parameter values, as defined in the Configure dialog or as modified by appropriate APIs. This function is called by the EZI2C_Sleep() function.

**Parameters:** None

**Return Value:** None

**Side Effects:** None

### void EZI2C_RestoreConfig(void)

**Description:** This function restores the component configuration and nonretention registers. It also restores the component parameter values to what they were prior to calling the EZI2C_Sleep() function.

**Parameters:** None

**Return Value:** None

**Side Effects:** Calling this function before EZI2C_Sleep() or EZI2C_SaveConfig() may produce unexpected behavior.

# Sample Firmware Source Code

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

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

# Functional Description

This component supports an $I^2C$ slave device with one or two $I^2C$ addresses. Either address may access a memory buffer defined in RAM, EEPROM, or flash data space. EEPROM and flash memory buffers are read only, while RAM buffers may be read/write. The addresses are right justified.

When using this component, you must enable global interrupts because the $I^2C$ hardware is interrupt driven. Even though this component requires interrupts, you do not need to add any code to the ISR (Interrupt Service Routine). The module services all interrupts (data transfers) independent of your code. The memory buffers allocated for this interface look like simple dual-port memory between your application and the $I^2C$ Master.

If required, you can create a higher-level interface between a master and slave by defining semaphores and command locations in the data structure.

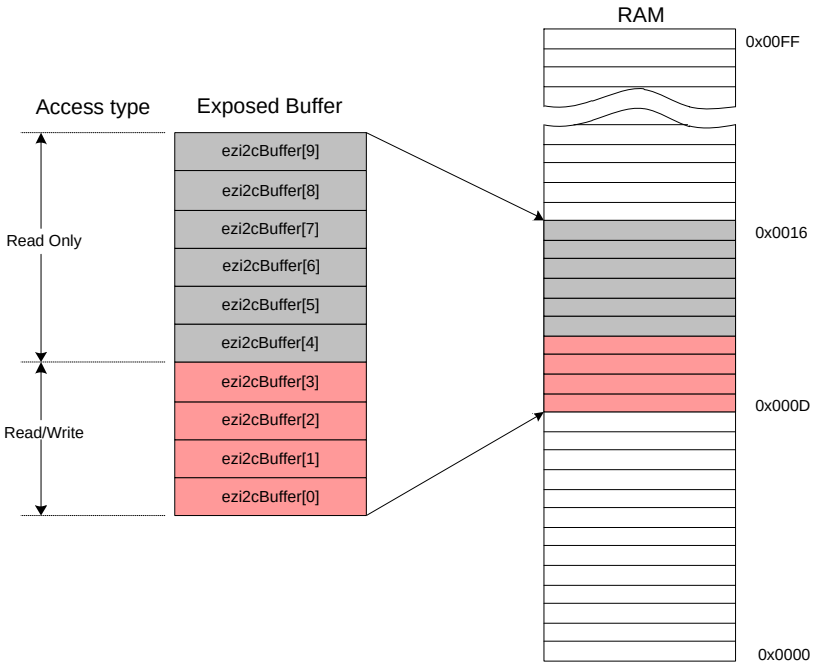
## Memory Interface

To an I²C master, the interface looks very similar to a common I²C EEPROM. The EZI2C interface can be configured as simple variables, arrays, or structures. In a sense it acts as one or two shared memory interfaces between your program and an I²C master through the I²C bus. The component only allows the I²C master to access the specified area of memory and prevents any reads or writes outside that area. For example, if the buffer for the primary slave address is configured as shown in the following code example, the buffer representation in memory could be represented as shown in Figure 1.

```
#define BUFFER_SIZE             (0x0Au)
#define BUFFER_RW_AREA_SIZE     (0x04u)

EZI2C_SetBuffer1(BUFFER_SIZE, BUFFER_RW_AREA_SIZE, (void *) ezi2cBuffer);
```

**Figure 1. Memory Representation of the EZI2C Buffer Exposed to an I²C Master**




The interface (I²C Master) only sees the structure as an array of bytes, and cannot access any memory outside the defined area. Using the example structure above, a supplied API is used to expose the data structure to the I²C interface.

```
char ezi2cBuffer2[15u];
EZI2C_SetBuffer2(15u, 8u, (void *) ezi2cBuffer2);
```

The data is transmitted in different endianess for different architectures. Therefore, extra code must be put to send in a specific endianess. For example, the CY_GET_REGXX()/ CY_SET_REGXX() macros (XX stands for 16/24/32) can be used to match little-endian ordering regardless of device architecture. For more information about endianess, see the Register Access section of the *System Reference Guide*.

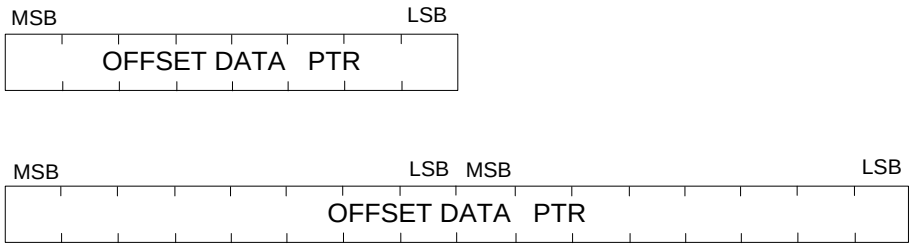
The following simple example shows only a single integer (two bytes) is exposed. Both bytes are readable and writable by the I²C master.

```
uint16 ezi2cVariable1;
CY_SET_REG16(&ezi2cVariable1, 0xABCD);
EZI2C_SetBuffer1(2u, 2u, (void *) (&ezi2cVariable1));
```

## Interface as Seen by External Master

The EZI2C Slave component supports basic read and write operations for the read/write area and read-only operations for the read-only area. The two I²C address interfaces contain separate data buffers that are addressed with separate offset data pointers. The offset data pointers are written by the master as the first one or two data bytes of a write operation, depending on the **Sub-address size** parameter. For the rest of this discussion, we will concentrate on an 8-bit sub-address size.

**Figure 2. The 8-bit and 16-bit Sub-Address Size (from top to bottom)**



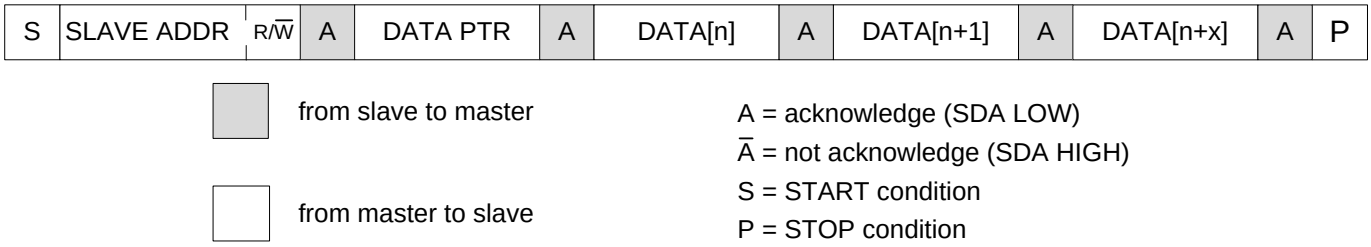

For write operations, the first data byte is always the offset data pointer (two bytes for **Sub-address size** = 16). The byte after the offset data pointer is written into the location pointed to by the offset data pointer. The second data byte is written to the offset data pointer plus one, and so on, until the write is complete. The length of a write operation is only limited by the maximum buffer read/write region size. For write operations, the offset data pointer must always be provided.

Read operations always begin at the offset data pointer provided by the most recent write operation. The offset data pointer increments for each byte read, the same way as a write operation. A new read operation will not continue from where the last read operation stopped. A new read operation always begins to read data at the location pointed to by the last write operation offset data pointer. The length of a read operation is only limited by the maximum size of the data buffer.

Typically, a read will contain a write operation of only the offset data pointer followed by a restart (or stop/start) and then the read operation. If the offset data pointer does not require update, as

in the case of repeatedly reading the same data, no additional write operations are required after the first. This greatly speeds read operations by allowing them to directly follow each other.

**Figure 3. Write x Bytes to I²C Slave**




For example, if the offset data pointer is set to four, a read operation begins to read data at location four and continues sequentially until the data ends or the host completes the read operation. This is true whether single or multiple read operations are performed. The offset data pointer is not changed until a new write operation is initiated.

If the I²C master tries to write data past the area specified by the EZI2C_SetBuffer1() or EZI2C_SetBuffer2() functions, the data is discarded and does not affect any RAM inside or outside the designated RAM area. Data cannot be read outside the allowed range. Any read requests by the master outside the allowed range results in the return of invalid data.

Figure 4 illustrates the data pointer write for an 8-bit offset data pointer.

**Figure 4. Set Slave Data Pointer**

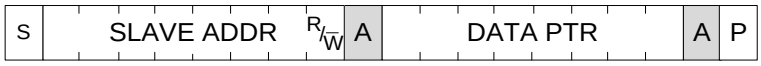



Figure 5 illustrates the read operation for an 8-bit offset data pointer. Remember that a data write operation always rewrites the offset data pointer.

**Figure 5. Read x Bytes from I²C Slave**




At reset, or power on, the EZI2C Slave component is configured and APIs are supplied, but the resource must be explicitly turned on using the EZI2C_Start() function.

Detailed descriptions of the I²C bus and the implementation are available in the complete I²C specification available on the Philips website, and by referring to the device datasheet.

## Data Coherency

Although a data buffer may include a data structure larger than a single byte, a Master read or write operation consists of multiple single-byte operations. This can cause a data coherency problem, because there is no mechanism to guarantee that a multi-byte read or write will be synchronized on both sides of the interface (Master and Slave). For example, consider a buffer

that contains a single two-byte integer. While the master is reading the two-byte integer one byte at a time, the slave may have updated the entire integer between the time the master read the first byte of the integer (LSB) and was about to read the second byte (MSB). The data read by the master may be invalid, since the LSB was read from the original data and the MSB was read from the updated value.

You must provide a mechanism on the master, slave, or both that guarantees that updates from the master or slave do not occur while the other side is reading or writing the data. The EZI2C_GetActivity() function can be used to develop an application-specific mechanism.

## Wakeup from Sleep Mode

If you want to use the **Enable wakeup from Sleep mode** feature, you may need to design the $I^2C$ Master to handle clock stretching procedure (hold SCL low).

The device clock's configuration (bus clock frequency) can be modified (by the CyPmSaveClocks() function) as part of the Sleep mode entry procedure. It must be restored (by the CyPmRestoreClocks() function) before the $I^2C$ transaction can continue in Active mode.

To meet these requirements, the $I^2C$ interrupt is disabled in EZI2C_SlaveSetSleepMode() and enabled in EZI2C_SlaveSetWakeMode(). As a result, when a hardware address match event occurs, the transaction pauses by holding the SCL line LOW (clock stretching procedure).

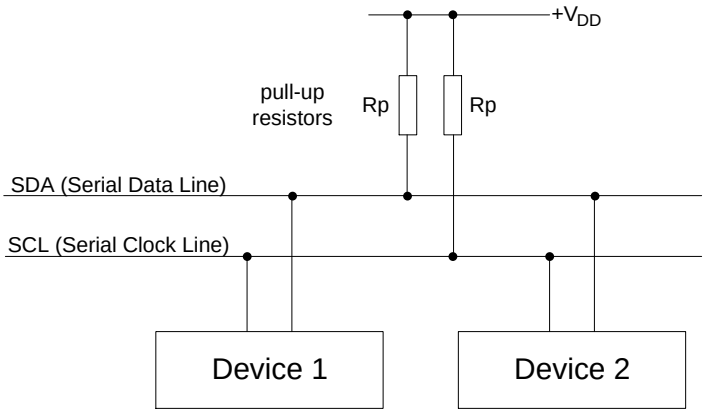
The following is the correct Sleep mode entry procedure if **Enable wake up from Sleep mode** is enabled:

```
/* Prepares EZI2C to wake up from Sleep Mode */
EZI2C_SlaveSetSleepMode();
/* Switches to the Sleep mode */
CyPmSaveClocks();
CyPmSleep(PM_SLEEP_TIME_NONE, PM_SLEEP_SRC_I2C);
CyPmRestoreClocks();
/* Prepares EZI2C to work in Active mode */
EZI2C_SlaveSetWakeMode();
```

## External Electrical Connections

As Figure 6 illustrates, the $I^2C$ bus requires external pull-up resistors. The pull-up resistors ($R_P$) are determined by the supply voltage, clock speed, and bus capacitance. Make the minimum sink current for any device (master or slave) no less than 3 mA at $V_{OLmax}$ = 0.4 V for the output stage. This limits the minimum pull-up resistor value for a 5-V system to about 1.5 kΩ. The maximum value for RP depends upon the bus capacitance and clock speed. For a 5-V system with a bus capacitance of 150 pF, the pull-up resistors should be no larger than 6 kΩ. For more information see *The $I^2C$-Bus Specification* on the Philips web site at www.philips.com.

**Figure 6. Connection of Devices to the I²C-Bus**




**Note** Purchase of I²C components from Cypress or one of its sublicensed Associated Companies, conveys a license under the Philips I²C Patent Rights to use these components in an I²C system, provided that the system conforms to the I²C Standard Specification as defined by Philips.

# Interrupt Service Routine

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

# Resources

The fixed-function I²C block is used for this component.

# 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 5 (GCC) | | PSoC 5LP (GCC) | |
|---|---|---|---|---|---|---|
| | Flash Bytes | SRAM Bytes | Flash Bytes | SRAM Bytes | Flash Bytes | SRAM Bytes |
| One address | 1019 | 18 | 972 | 22 | 984 | 22 |
| Two addresses | 1679 | 37 | 1508 | 43 | 1500 | 43 |

# 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 Specifications

| Parameter | Description | Conditions | Min | Typ | Max | Units |
|---|---|---|---|---|---|---|
| $I_{DD}$ | Block current consumption | Enabled, configured for 100 kbps | – | – | 250 | µA |
| | | Enabled, configured for 400 kbps | – | – | 260 | µA |
| | | Wake from sleep mode | – | – | 30 | µA |

## AC Specifications

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

# Component Changes

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

| Version | Description of Changes | Reason for Changes / Impact |
|---|---|---|
| 1.70.a | Minor datasheet edit. | |
| 1.70 | Added PSoC 5LP support. | |
| 1.61 | Enhanced verification of the options configured within the customizer and related to the **Enable wakeup from Sleep Mode** option. | Prevents components from being configured with an unsupported mode. |
| | Updated EZI2C_Stop() implementation for PSoC 3 devices. | Makes EZI2C_Stop() release the bus if it was locked. |
| | Updated the default $I^2C$ addresses to 8 and 9 to comply with $I^2C$ bus specification requirements. | Previously used addresses are reserved according to the $I^2C$ bus specification. |
| | Updated the component debugger tool window support. | Enhanced debug window support. |
| | Added the possibility to declare every function as reentrant for PSoC 3 by adding the function name to the .cyre file. | Not all APIs are truly reentrant. Comments in the component API source files indicate which functions cannot be truly reentrant.<br>This change is required to eliminate compiler warnings for functions that are used in a safe way (protected from concurrent calls by flags or Critical Sections) and are not reentrant. |
| 1.60.b | Datasheet corrections | |

| Version | Description of Changes | Reason for Changes / Impact |
|---|---|---|
| 1.60.a | Updated the Pin Connections section with information about dependencies between the Enable wakeup from Sleep mode and Number of addresses options. | Explained that option is supplemental for the Enable wakeup from Sleep mode option and is also available only if a single I²C address is selected in Number of addresses option. |
| | Figures 2, 3, and 5 were updated to show bit fields. | Visibility enhancement, |
| | Clarified the method of writing portable code regardless of the PSoC device architecture. | Documentation enhancement. |
| 1.60 | The method of working with the slave enable bit was changed: EZI2C_Stop() does not clear this bit now and setting this bit was moved from EZI2C_Enable() to EZI2C_Init(). The I²C configuration register is now restored in EZI2C_RestoreConfig() function. | To achieve correct result of EZI2C_Start() - EZI2C_Stop() - EZI2C_Start() and EZI2C_Sleep() - EZI2C_Wakeup() sequences. No functional impact is expected. |
| | The label I2C Bus Speed: in the customizer was replaced with Data Rate. The Wakeup from Sleep Mode section was added to the Functional Description. | Consistency between I²C-Bus Specification naming and I²C/EZI2C components. |
| | The label "I2C pins connected to" in customizer was replaced with "Pin Connections" | The text was fixed for consistency with requirements. |
| | The label "Enable wakeup from the Sleep mode" in customizer was replaced with "Enable wakeup from Sleep mode" | The text was fixed for consistency with requirements. |
| | The component symbol and catalog placement name was updated: the “EZ I2C” was renamed to “EZI2C”. | The text was fixed for consistency with requirements. |
| | Fixed issues when global variables used in both code and ISR could potentially be optimized out by compiler. | Prevents optimization issues that could lead to unexpected result. |
| | Added characterization data to datasheet | |
| | Minor datasheet edits and updates | |
| 1.50.a | Moved component into subfolders of the component catalog | |
| 1.50 | Standard data rate has been updated to support up to 1 Mbps. | Allows setting up I²C bus speed up to 1 Mbps. |
| | Keil reentrancy support was added. | Support for PSoC 3 with the Keil compiler the capability for functions to be called from multiple flows of control. |
| | Added Sleep/Wakeup and Init/Enable APIs. | To support low-power modes and to provide common interfaces to separate control of initialization and enabling of most components. |

| Version | Description of Changes | Reason for Changes / Impact |
|---|---|---|
| | The XML description of the component has been added. | This allows PSoC Creator to provide a mechanism for creating new debugger tool windows for this component. |
| | Added support for the PSoC 3 Production devices. | The required changes have been applied to support hardware changes between PSoC 3 ES2 and Production devices. |
| | The default schematic template has been added to the component catalog. | Every component should have a schematic template. |
| | The EZI2C's bus speed generation was fixed. Previously it was x4 greater than should be. Added more comments in the source code to describe bus speed calculation. | The proper $I^2C$ bus speed calculation and generation. |
| | Optimized form height for Microsoft Windows 7. | In Windows 7 scrollbar appeared just after customizer start. |
| | Added tooltips for address input boxes with 'Use 0x prefix for hexadecimals' text. | To inform user about possibility of hexadecimal input. |
| 1.20.a | Moved component into subfolders of the component catalog. | |
| | Added information to the component that advertizes its compatibility with silicon revisions. | The tool reports an error/warning if the component is used on incompatible silicon. If this happens, update to a revision that supports your target device. |
| 1.20 | Updated the Configure dialog. | |
| | Changed Digital Port to Pins component in the schematic | |

© Cypress Semiconductor Corporation, 2012-2015. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

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

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

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

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