EZI2C Slave
2.0



Features

- Industry standard NXP® 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. It is compatible [1] with I²C Standard-mode, Fast-mode, and Fast-mode Plus devices as defined in the NXP I2C-bus specification. 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.

---

1 The I²C peripheral is non-compliant with the NXP I²C specification in the following areas: analog glitch filter, I/O V_{OL}/I_{OL}, 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 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

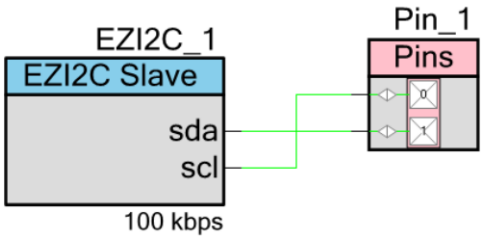
Serial data (SDA) is the I²C data signal. It is a bidirectional data signal used to transmit or receive all bus data.

scl – In/Out

Serial clock (SCL) is the master generated I²C 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.

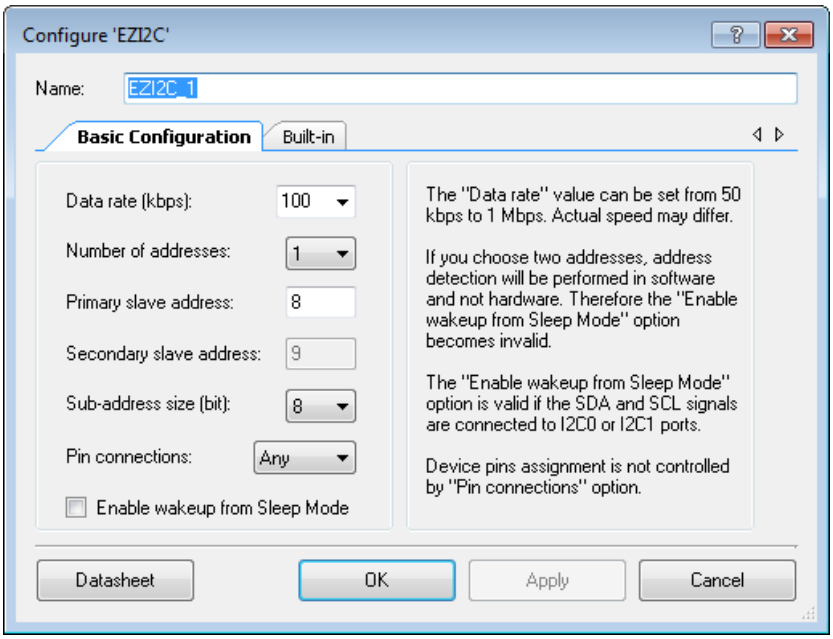
Schematic Macro Information

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 I^2C 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 I^2C 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.

Note If **Number of address** is **1**, the usage of I^2C repeated Start condition to join transactions to different devices is not supported. If the I^2C master accesses other I^2C devices and then generates a repeated Start to access the component, the component fails to operate properly. The I^2C transactions of this type must be avoided.

Primary slave address

This is the primary I^2C slave address (default is **8**). You can select a slave address between 0 and 127 (0x00 and 0x7F). 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.

Secondary slave address

This is the secondary I^2C slave address (default is **9**). It is only valid when the Number of addresses parameter is set to **2**. You can select a slave address between 0 and 127 (0x00 and 0x7F). 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.

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. If you select a sub-address size of 16 bits, the master can access data offsets between 0 and 65535.

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** (default), I2C0, and I2C1.

Value	Pins
Any	Any general-purpose I/O (GPIO or SIO) can be used for SCL and SDA pins.
I2C0	Restricted placement: SCL = SIO pin P12[4], SDA = SIO pin P12[5]
I2C1	Restricted placement: 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 EZI2C component does not check the correct pin assignments.

Enable wakeup from Sleep Mode

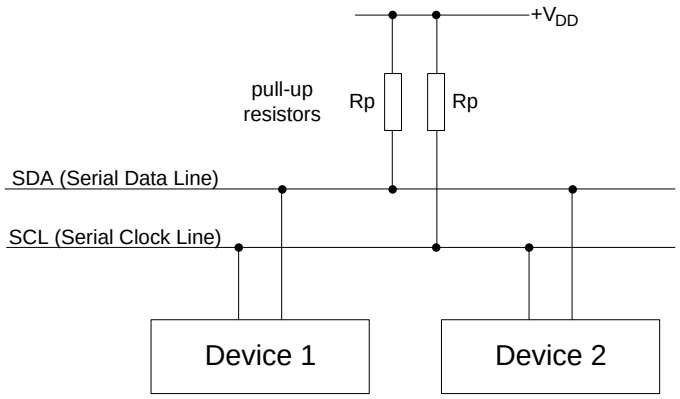
This parameter allows the device to be awakened from sleep mode on slave address match. This option is disabled by default. The wake up on address match option is valid if a single I^2C address is selected and the SDA and SCL signals are connected to SIO ports (pin pairs I2C0 or I2C1).

Refer to the Wakeup from Sleep Mode section in this document; refer also to the *Power Management APIs* section of the *System Reference Guide* for more information.

External Electrical Connections

As Figure 1 shows, the I2C 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.

Figure 1. Connection of Devices to the I^2C 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 I^2C 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 I^2C 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 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.”

Basic Functions

Function	Description
EZI2C_Start()	Starts responding to I^2C traffic. Enables interrupt.
EZI2C_Stop()	Stops responding to I^2C traffic. Disables interrupt.
EZI2C_EnableInt()	Enables component interrupt, which is required for most component operations.
EZI2C_DisableInt()	Disables component interrupt. The EZI2C_Stop() API does this automatically.
EZI2C_SetAddress1()	Sets the primary I^2C slave address.
EZI2C_GetAddress1()	Returns the primary I^2C slave address.
EZI2C_SetBuffer1()	Sets up the data buffer to be exposed to the master on a primary slave address request.
EZI2C_GetActivity()	Checks component activity status.
EZI2C_Sleep()	Stops I^2C operation and saves I^2C configuration. Disables component interrupt.
EZI2C_Wakeup()	Restores I^2C configuration and starts I^2C operation. Enables component interrupt.
EZI2C_Init()	Initializes I^2C 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 I^2C registers.

void EZI2C_Start(void)

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

This function enables the component interrupt because interrupt is required for most component operations.

Parameters: None

Return Value: None

Side Effects: None

void EZI2C_Stop(void)

Description: Disables I^2C hardware and component interrupt. The I^2C bus is released if it was locked up by the component.

Parameters: None

Return Value: None

Side Effects: None

void EZI2C_EnableInt(void)

Description: Enables component interrupt. Interrupts are required for most operations. Called inside EZI2C_Start() function.

Parameters: None

Return Value: None

Side Effects: None

void EZI2C_DisableInt(void)

Description: Disables component interrupt. This function is not normally required because the EZI2C_Stop() function calls this function.

Parameters: None

Return Value: None

Side Effects: If the component interrupt is disabled while the EZI2C communicates with master or master tries to access the EZI2C slave. The bus can be locked up unit component interrupt is enabled or component is disabled calling EZI2C_Stop().

void EZI2C_SetAddress1(uint8 address)

Description: Sets the primary I^2C slave address. This address is used by the master to access the primary data buffer.

Parameters: uint8 address: Primary I^2C slave address.

This address is the 7-bit right-justified slave address and does not include the R/W bit. This value can be any address between 0 and 127 (0x00 to 0x7F).

Return Value: None

Side Effects: None

uint8 EZI2C_GetAddress1(void)

Description: Returns the primary I^2C slave address.

This address is the 7-bit right-justified slave address and does not include the R/W bit.

Parameters: None

Return Value: uint8: Primary I^2C slave address.

Side Effects: None

void EZI2C_SetBuffer1(uint16 bufSize, uint16 rwBoundary, volatile uint8* dataPtr)

Description: Sets up the data buffer to be exposed to the master on a primary slave address request.

Parameters: uint16 bufSize: Size of the buffer in bytes.

uint16 rwBoundary: Number of data bytes starting from the beginning of the buffer with read and write access. Data bytes located at offset rwBoundary or greater are read only.

This value must be less than or equal to the buffer size.

uint8* dataPtr: Pointer to the data buffer.

Return Value: None

Side Effects: None

uint8 EZI2C_GetActivity(void)

Description: Returns a non-zero value if an I^2C 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 when slave is free (that is, the master finishes communication with the slave generating Stop or repeated Start condition).

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

Return Value:

uint8: Status of I^2C 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 when master starts communication with slave (the slave is addressed with Start or repeated Start) and cleared when master finishes communication (Stop or repeated Start condition is generated).
EZI2C_STATUS_ERR	Set when I^2C hardware error is detected. Cleared when status is read.

Side Effects: None

void EZI2C_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 EZI2C component state, saves it, and disables the component by calling EZI2C_Stop() if it is currently enabled. EZI2C_SaveConfig() is then called to save the component nonretention configuration registers.
- Checked: If a transaction intended for component is in progress 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 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 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 EZI2C_RestoreConfig(). If the component was enabled before the EZI2C_Sleep() function was called, EZI2C_Wakeup() re-enables it.
- Checked: Disables the backup regulator of I^2C hardware. The incoming transaction continues as soon as the regular EZI2C interrupt handler is set up (global interrupts has to be enabled to service EZI2C component interrupt).

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 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: None

void EZI2C_Enable(void)

Description: Activates the hardware and begins component operation. Calls EZI2C_EnableInt() to enable the component interrupt. 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.

Parameters: None

Return Value: None

Side Effects: None

void EZI2C_SaveConfig(void)

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

- Unchecked: Stores the component nonretention configuration registers.
- Checked: 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 EZI2C_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: Disables the backup regulator of the I^2C 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 I^2C transaction.

Parameters: None

Return Value: None

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

Optional Second Address APIs

These commands are present only if two I^2C addresses are enabled.

Function	Description
EZI2C_SetAddress2()	Sets the secondary I^2C slave address.
EZI2C_GetAddress2()	Returns the secondary I^2C slave address.
EZI2C_SetBuffer2()	Sets up the data buffer to be exposed to the master on a secondary slave address request.

void EZI2C_SetAddress2(uint8 address)

Description: Sets the secondary I^2C slave address. This address is used by the master to access the secondary data buffer.

Parameters: uint8 address: Secondary I^2C slave address.

This address is the 7-bit right-justified slave address and does not include the R/W bit. This value can be any address between 0 and 127 (0x00 to 0x7F).

Return Value: None

Side Effects: None

uint8 EZI2C_GetAddress2(void)

Description: Returns the secondary I^2C slave address.

This address is the 7-bit right-justified slave address and does not include the R/W bit.

Parameters: None

Return Value: uint8: Secondary I^2C slave address.

Side Effects: None

void EZI2C_SetBuffer2(uint16 bufSize, uint16 rwBoundary, volatile uint8* dataPtr)

Description: Sets up the data buffer to be exposed to the master on a secondary slave address request.

Parameters: uint8 bufSize: Size of the data buffer in bytes.

uint8 rwBoundary: Number of data bytes starting from the beginning of the buffer with read and write access. Data bytes located at offset rwBoundary or greater are read only.

This value must be less than or equal to the buffer size.

uint8* dataPtr: Pointer to the data buffer.

Return Value: None

Side Effects: None

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. 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.

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.

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 EZI2C component has the following specific deviations:

Rule	Class [2]	Rule Description	Description of Deviation(s)
17.4	R	Array indexing shall be the only allowed form of pointer arithmetic.	Component uses array indexing operation to access buffers. The buffer size is checked before access. It is safe operation unless user provides incorrect buffer size.
19.7	A	A function should be used in preference to a function-like macro.	Deviated since function-like macros are used to allow more efficient code.

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

2 Required / Advisory

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
One address	1175	21	1240	24
Two addresses	1743	37	1620	41

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.

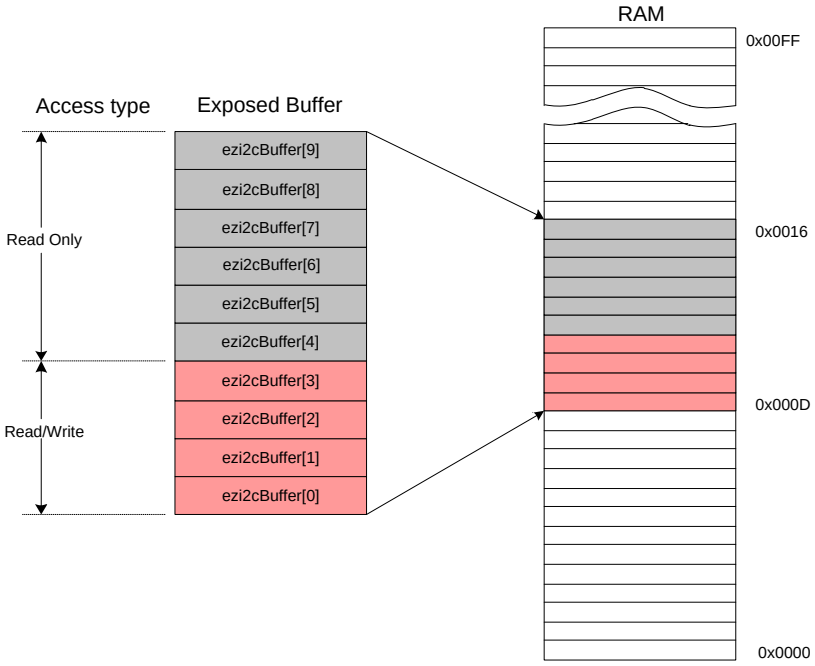
Memory Interface

To an I^2C master, the interface looks very similar to a common I^2C EEPROM. The EZI2C interface can be configured as simple variables, arrays, or structures but the safer method is using arrays. In a sense it acts as one or two shared memory interfaces between your program and an I^2C master through the I^2C bus. The component only allows the I^2C 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 2.

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

uint8 ezi2cBuffer[BUFFER_SIZE];
EZI2C_SetBuffer1(BUFFER_SIZE, BUFFER_RW_AREA_SIZE, ezi2cBuffer);
```

Figure 2. Memory Representation of the EZI2C Buffer Exposed to an I²C Master



To make whole buffer with read and write access the buffer size and read/write boundary need to be the same size. For example:

```
EZI2C_SetBuffer1(BUFFER_SIZE, BUFFER_SIZE, ezi2cBuffer);
```

Handle structures

The EZI2C buffer can be set up as structure. The interface (I²C Master) only sees the structure as an array of bytes, and cannot access any memory outside the defined area.

The compiler lays out structures in the memory and may add extra bytes. This is called byte padding. The compiler will add these bytes to align the fields of the structure to match the requirements of the Cortex-M3. When using a structure, the application must take this alignment into account. To avoid padding bytes, the attribute “packed” should be used:

```
struct
{
    uint8 status;
    uint8 data0;
    uint32 data1;
} __attribute__ ((packed)) ezi2cBuffer;

SCB_EzI2CSetBuffer1(sizeof(ezi2cBuffer), sizeof(ezi2cBuffer),(uint8 *)
&ezi2cBuffer);
```

Handling endianess

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*.

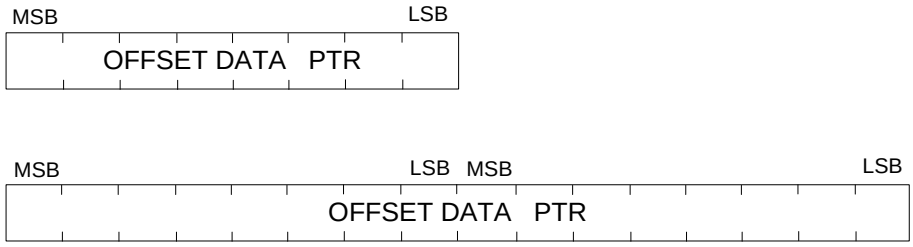
The following simple example shows only a single integer (two bytes) is exposed. Both bytes are readable and writable by the I^2C master.

```
uint16 ezi2cVariable1;
CY_SET_REG16(&ezi2cVariable1, 0xABCD);
EZI2C_SetBuffer1(2u, 2u, (uint8 *) (&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 I^2C 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. The sub-address size of 8-bits is used to access buffers up to 256 bytes and sub-address size of 16-bits is used for buffers up to 65536 bytes. For the rest of this discussion, we will concentrate on an 8-bit sub-address size.

Figure 3. The 8-bit and 16-bit Sub-Address Size (from top to bottom)



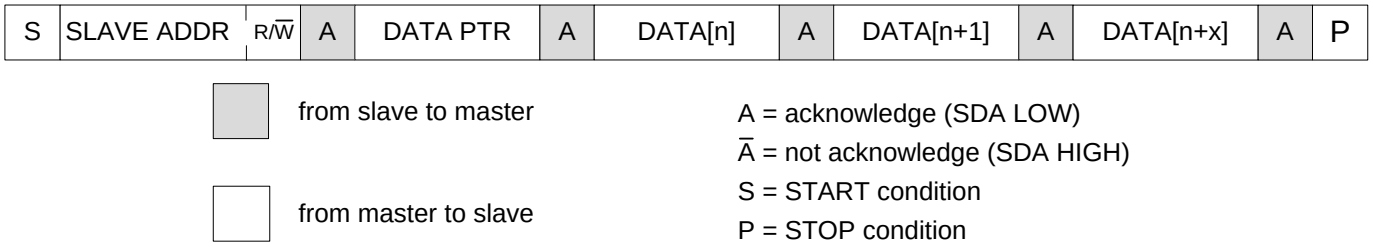
For write operations, the first data byte is always the offset data pointer (two bytes for sub-address size = 16). The byte after the offset data pointer is written into the location pointed to by the offset data pointer. The second data byte is written to the offset data pointer plus one, and so on, 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 4. 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 5 illustrates the data pointer write for an 8-bit offset data pointer.

Figure 5. Set Slave Data Pointer

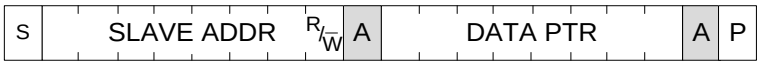
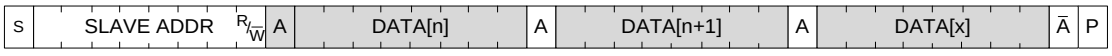


Figure 6 illustrates the read operation for an 8-bit offset data pointer. Remember that a data write operation always rewrites the offset data pointer.

Figure 6. 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

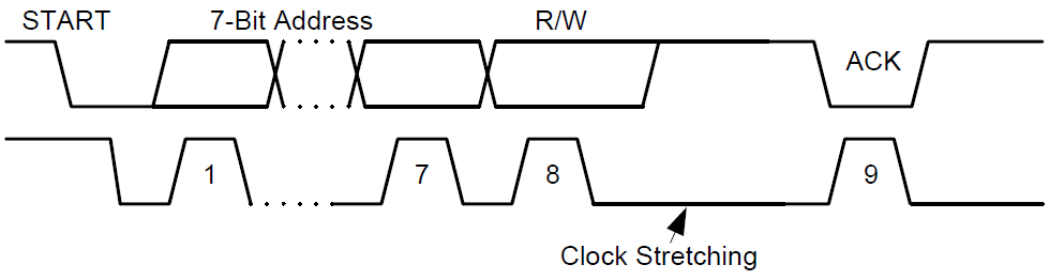
Although a data buffer may include a data structure larger than a single byte, a Master read or write operation consists of multiple single-byte operations. This can cause a data coherency problem, because there is no mechanism to guarantee that a multi-byte read or write will be synchronized on both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

Clock Stretching

Clock stretching pauses a transaction by holding the SCL line low. The transaction cannot continue until the line is released high again. Figure 7 shows clock stretching after an address byte has been received. However, the clock stretching feature is optional according to the I^2C specification but the EZI2C slave component supports it. The slave can stretch the SCL line on every received address or data byte from the master because each received byte requires interrupt processing. When the CPU is not able to service the slave's interrupt fast enough, the SCL line is stretched and tied low until the CPU commands the hardware to release the bus. The command to release the bus happens in different places of the slave's interrupt service routine, depending on the transfer phase (address or data byte) and direction (read or write). The CPU has half of the SCL clock cycle period to process the slave's interrupt before stretching occurs. The factors that affect clock stretching include: I^2C bus speed, CPU speed, compiler optimization options, and servicing other interrupts in the device. It is recommended to use an I^2C master that supports clock stretching to communicate with the EZI2C slave component.

Figure 7. Clock Stretching After Address Byte



Wakeup from Sleep Mode

The EZI2C component is able to be a wakeup source from Sleep low power mode, but it must be configured properly.

- The Number of addresses option is set to 1. The address comparison handled by hardware.
- The Enable wakeup from Sleep Mode option must be selected, and the Pin connections must be specified as I2C0 or I2C1.
- The component does not control the pins assignment; therefore, make sure to place SCL and SDA pins at the I2C0 or I2C1 position in the Design-Wide Resources Pins Editor.
- The EZI2C_Sleep() function must be called before entering Sleep mode, and the EZI2C_Wakeup() function must be called after exiting.
- The device clock's configuration must be stored before entering Sleep mode using CyPmSaveClocks(), and it must be restored after exiting Sleep mode using CyPmRestoreClocks() functions.

The wakeup event is the slave primary address match. The logic active in Sleep mode performs address matching and when a matched address is detected the hardware generates an interrupt request. The SCL line remains pulled low after address reception until device is waking up and EZI2C_Wake() is called. The incoming transaction continues after SCL is released (the global interrupts must be enabled to service the component interrupt).

The following code is suggested to enter and exit Sleep mode:

```
/* Prepares EZI2C to wake up from Sleep mode */
EZI2C_Sleep();

/* Switches to the Sleep mode */
CyPmSaveClocks();
CyPmSleep(PM_SLEEP_TIME_NONE, PM_SLEEP_SRC_I2C);
CyPmRestoreClocks();

/* Prepares EZI2C to work in Active mode */
EZI2C_Wakeup();
```

Clock Selection

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

Interrupt Service Routine

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

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)*.

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-function I^2C block and one interrupt are used for this component.

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
2.0	Remove support of obsolete APIs: EZI2C_SlaveSetSleepMode() EZI2C_SlaveSetWakeMode()	These are obsolete functions. EZI2C_Sleep() and EZI2C_Wakeup() must be used instead. Refer to the Wakeup from Sleep Mode section for more information about enter and exit Sleep mode.
	Remove buffer pointer and buffer size clear operation from EZI2C_Init() function.	The buffers can be configured use EZI2C_SetBuffer1() or EZI2C_SetBuffer2() APIs before component is enabled by calling EZI2C_Start(). There is no impact on code that calls EZI2C_Start() before buffer initialization. This was a requirement for previous versions of the component.
	Add protection from the component interruption to the following APIs: EZI2C_GetActivity() EZI2C_SetBuffer1() EZI2C_SetBuffer2()	EZI2C operations executed by the listed functions are atomic.
	Fixed EZI2C_Stop() function to properly reset the I^2C fixed-function block and restore effected registers.	The problem described in the errata of the component version 1.90 is fixed.
	Datasheet updates.	Updated External Electrical Connections section, and placed it closer to the top of the document. Updated Wakeup from Sleep Mode section. Improved description of some APIs (no functional changes). Added Component Debug Window and Clock Stretching sections. Reduced the number of global variables shown in the Global Variables section.
1.90.b	Datasheet updated to add Component Errata section.	Refer to Cypress ID 197653.
1.90.a	Updated the datasheet only.	API sections were out of order. Also updated to comply with latest template.
1.90	Updated MISRA Compliance section.	The component has specific deviations described.
	The type of global variables EZI2C_bufSizeS1, EZI2C_wrProtectS1, EZI2C_bufSizeS2 and EZI2C_wrProtectS2 were changed from uint8 to uint16.	The buffer length equal 256 bytes is supported if the Sub-Address Size is 8 bits.

Version	Description of Changes	Reason for Changes / Impact
	Removed mention about EZI2C_SlaveSetSleepMode() and EZI2C_SlaveSetWakeMode().	These are obsolete functions. EZI2C_Sleep() and EZI2C_Wakeup() have to be used instead of it.
	Added Handle structures section	Documentation enhancement
1.80	Added MISRA Compliance section.	The component was not verified for MISRA compliance.
	Added footnote about non-compliant with the NXP I^2C specification in the some areas.	Documentation enhancement.
	Changed the control flow of the wake up sequence to avoid disabling the I^2C interrupt.	PSoC 5 LP requires an I^2C interrupt to be enabled in order to wake up the device at the event of an address match.
	Fixed control flow behavior when master completes reading beyond the buffer size.	The slave doesn't complete transaction correctly because NAK from master was not checked.
1.70.a	Corrected figure 5.	
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. 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	
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 I^2C 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.

Version	Description of Changes	Reason for Changes / Impact
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 I^2C 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 I^2C-Bus Specification naming and I^2C/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 I^2C 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.
	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.

Version	Description of Changes	Reason for Changes / Impact
	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, 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.