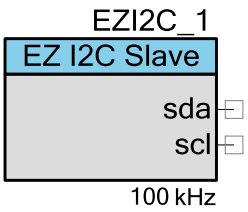
EZ I2C Slave

1.20

Features



- Industry standard Philips I²C bus compatible interface
- Emulates common I²C EEPROM interface
- Only two pins (SDA and SCL) required to interface to I2C bus
- Standard data rate of 50/100/400 kbps
- High level API requires minimal user programming
- Support one or two address decoding

General Description

The EZ I²C 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 EZ I²C Slave supports the standard mode with speeds up to 400 kbps and is compatible with multiple devices on the same bus.

The EZ I²C 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 Start() function is executed, there is little need to for the user to interact with the API.

When to use a EZ I²C Slave

This component is best used when a shared memory model between the I²C Slave and I²C Master is desired. The EZ I²C Slave buffer/s may be defined as any variable, array, or structure in the user's code without any thought of the I²C protocol. The I²C master may view any of the variables in this buffer and modify the variables defined by the SetBuffer1/2 function.

Input/Output Connections

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

SDA – In/Out

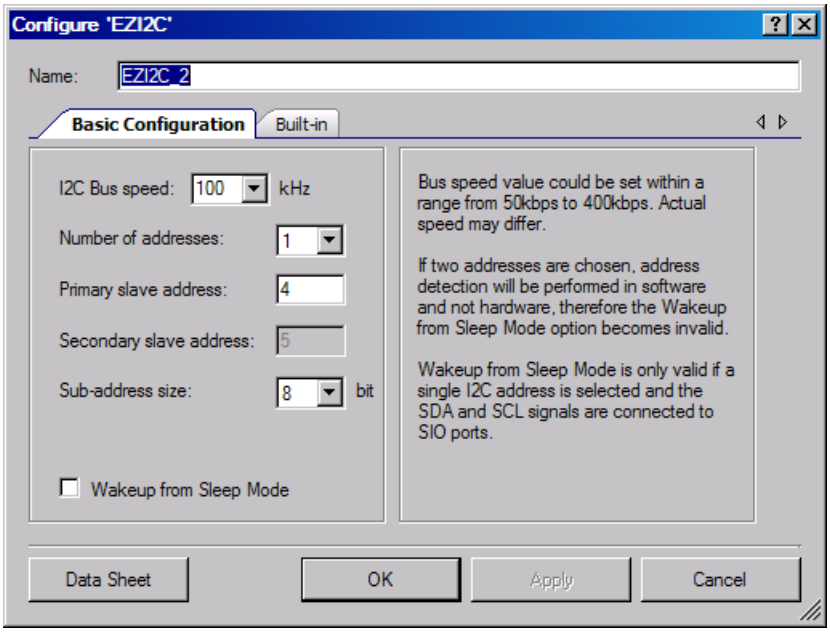
This is the I²C data signal. It is a bi-directional data signal used to transmit or receive all bus data.

SCL – In/Out

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

Parameters and Setup

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



The EZ I²C component provides the following parameters.

I2C Bus Speed

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

Number of Addresses

This option determines if 1 (default) or 2 independent I^2C slave addresses are recognized. If two addresses are recognized, address detection will be performed in software and not hardware, therefore the EnableWakeup option becomes invalid.

Primary Slave Address

This is the primary I^2C slave address (default is 4).

Secondary Slave Address

This is the secondary I^2C slave address (default is 5). This second address is only valid when the parameter "Number of Addresses" is set to 2.

Sub-address Size

This option determines what range of data can be accessed. A sub-address of 8 (default) or 16 bits may be selected. If an address size of 8 bits is used, the master may only access data offsets between 0 and 254. 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 address.

Wakeup from Sleep Mode

This option enables the system to be awakened from sleep when an address match occurs (default is true). This option is only valid if a single I^2C address is selected and the SDA and SCL signals are connected to SIO ports.

Clock Selection

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

Resources

The fixed I^2C block is used for this component.

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
void EZI2C_Start(void)	Start responding to I²C traffic. (Enables interrupt)
void EZI2C_Stop(void)	Stop responding to I²C traffic (Disables interrupt)
void EZI2C_EnableInt(void)	Enable interrupt, Start does this automatically.
void EZI2C_DisableInt(void)	Disable interrupt, Stop does this automatically.
void EZI2C_SetAddress1(uint8 addr)	Set the I²C primary address that it should respond.
uint8 EZI2C_GetAddress1(void)	Return the I²C address for the primary device.
void EZI2C_SetBuffer1(uint16 bufSize, uint16 rwBoundry, void * dataPtr);	Set the buffer pointer for the primary address for both reads and writes.
uint8 EZI2C_GetActivity(void)	Check status on device activity.

Optional Second Address API

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

Function	Description
void EZI2C_SetAddress2(uint8 addr)	Set the I²C secondary address that it should respond.
uint8 EZI2C_GetAddress2(void)	Return the I²C address for the secondary device.
void EZI2C_SetBuffer2(uint16 bufSize, uint16 rwBoundry, void * dataPtr);	Set the buffer pointer for the secondary address for both reads and writes.

Optional Sleep/Wake modes

These functions are only available if a single address is used and the SCL and SDA signals are routed to the SIO ports.

Function	Description
void EZI2C_SlaveSetSleepMode(void)	
void EZI2C_SlaveSetWakeMode(void)	

void EZI2C_Start(void)

Description: This function initializes the I²C hardware and enables the I²C interrupt.

Parameters: None

Return Value: None

Side Effects: Enables I²C interrupt.

void EZI2C_Stop(void)

Description: Disables I²C hardware and disables I²C interrupt.

Parameters: None

Return Value: None

Side Effects: None

void EZI2C_EnableInt(void)

Description: Enables I²C interrupt. Normally this function is not required since the Start function enables the interrupt.

Parameters: None

Return Value: None

Side Effects: None

void EZI2C_DisableInt(void)

Description: Disable I²C interrupts. Normally this function is not required since the Stop function disables the interrupt. If the I²C interrupt is disabled while the I²C master is still running, it may cause the I²C bus to lock up.

Parameters: None

Return Value: None

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

void EZI2C_SetAddress1(uint8 address)

Description: Sets the I²C slave address for the primary device. This value may be any value between 0 and 127.

Parameters: (uint8) address: I²C slave address for the primary device.

Return Value: None

Side Effects: Should be called after EZI2C_Start().

uint8 EZI2C_GetAddress1(void)

Description: Returns the I²C slave address for the primary device.

Parameters: None

Return Value: (uint8) 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: (uint16) bufSize: Size of the buffer exposed to the I²C master.

(uint16) rwBoundry: Bytes from offset 0 to (rwBoundry-1) are both readable and writable by the I²C master. Data located at offset rwBoundry and above are read only.

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

Return Value: None

Side Effects: None

uint8 EZI2C_GetActivity(void)

Description: This function returns status bits that are set depending on I²C bus activity.

Parameters: None

Return Value: (uint8) Status of I²C activity.

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

Side Effects: None

void EZI2C_SetAddress2(uint8 address)

Description: Sets the I²C slave address for the second device. 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: (uint8) address: I²C slave address for the second device.

Return Value: None

Side Effects: Should be called after EZI2C_Start().

uint8 EZI2C_GetAddress2(void)

Description: Returns the I²C slave address for the second device. This function is only provided if two I²C addresses have been selected in the user parameters.

Parameters: None

Return Value: (uint8) 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 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: (uint16) bufSize: Size of the buffer exposed to the I²C master.

(uint16) rwBoundry: Bytes from offset 0 to (rwBoundry-1) are both read and writable by the I²C master. Data located at offset rwBoundry and above are read only.

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

Return Value: None

Side Effects: None

void EZI2C_SlaveSetSleepMode(void)

Description: Disables the run time EZ I²C and enables the sleep Slave I²C. Should be called just prior to entering sleep. This function is only provided if a single I²C address is used.

Parameters: None

Return Value: None

Side Effects: None

void EZI2C_SlaveSetWakeMode(void)

Description: Disables the sleep EZ I²C slave and re-enables the run time I²C. Should be called just after awaking from sleep. Must preserve address to continue. This function is only provided if a single I²C address is used.

Parameters: None

Return Value: None

Side Effects: None

Sample Firmware Source Code

The following is a C language example demonstrating the basic functionality of the EZ I²C component. This example assumes the component has been placed in a design with the default name "EZI2C_1".

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

```
/*******************************************************
*  Example code to demonstrate the use of the EZ I2C
*
*  This example enables two Slave addresses. The buffer for
*  the first is set to the structure MyI2C_Regs and the
*  buffer for the second address is set to the constant
*  string DESC. The slave addresses for buffer1 and buffer2
*  are set to 6 and 7 respectively.
*  Parameter Settings:
*  BusSpeed_kHz:  400
*  EnableWakeup:  false
*  I2C_Address1:  4  (Does not matter since program resets to 6)
*  I2C_Address2:  5  (Does not matter since program resets to 7)
*  I2C_Addresses: 2
*  Sub_Address_Size:  Width_8_bits
```

```
*****************************************************************/

#include <device.h> /* Part specific constants and macros */

struct I2C_Regs     /* Example I2C interface structure  */
{
    uint8 stat;     /* R/W variable */
    uint8 cmd;      /* R/W variable */
    int16 volts;    /* R/W variable */
    char cStr[6];   /* Read only string  */
}
MyI2C_Regs;

const char DESC[] = "Hello I2C Master";

void main()
{
   CYGlobalIntEnable;   /* Enable global interrupts */
   EZI2C_1_Start();                                      /* Turn on I2C    */
                                                         /* Set up Buffer1 */
   EZI2C_1_SetBuffer1(sizeof(MyI2C_Regs), 4, (void *) &MyI2C_Regs);
   EZI2C_1_SetBuffer2(sizeof(DESC), 10, (void *)&DESC); /* Set up buffer2 */

   EZI2C_1_SetAddress1(6);                    /* Change address1 to 6 */
   EZI2C_1_SetAddress2(7);                    /* Change address2 to 7 */

   while(1) {
      /* Place user code here to update and read structure data. */
   }
}
```

Functional Description

This component supports only an I²C slave configuration with one or two I²C addresses. Either address may be defined as RAM, or FLASH data space. The addresses are right justified.

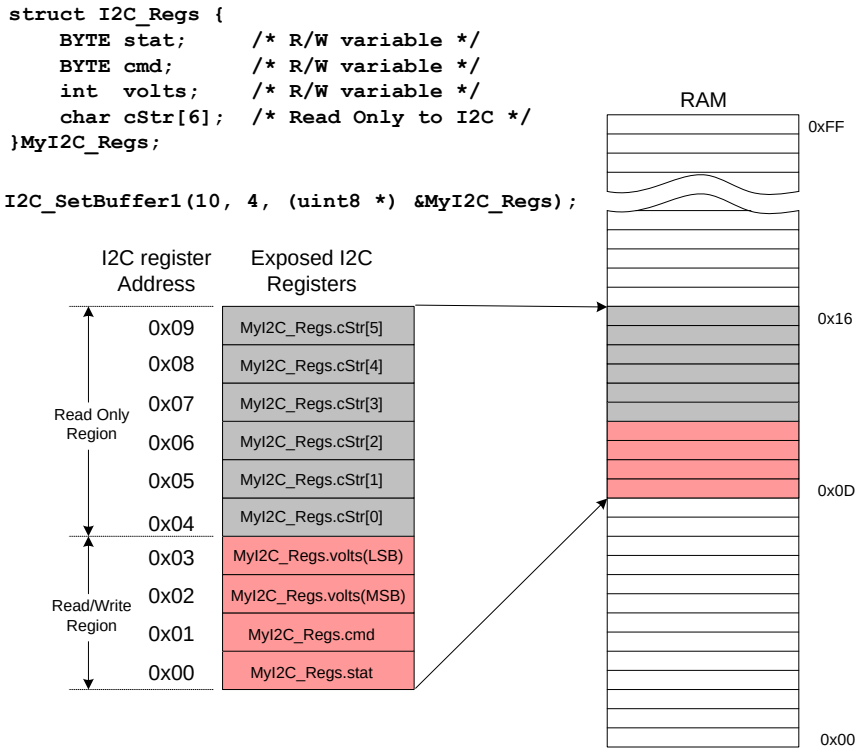
This component requires that you enable global interrupts since the I²C hardware is interrupt driven. Even though this component requires interrupts, you do not need to add any code to the ISR (Interrupt Service Routine). The module services all interrupts (data transfers) independent of your code. The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

If required, you can create a higher level interface between a master and this 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 EZ I²C API is treated as RAM or FLASH that can be configured as simple variables, arrays, or structures. In a sense it acts as a shared memory interface between your program and an I²C master on the I²C

bus. The API allows the user to expose any data structure to an I²C Master. The component only allows the I²C master to access the specified area of memory and prevents any reads or writes outside that area. The data exposed to the I²C interface can be a single variable, an array of values, or a structure. All that is required is a pointer to the start of the variable or data structure when initialized. The interface to the internal processor or I²C master is identical for both slave addresses. See the following diagram.



For example, you could create this structure.

```
struct I2C_Regs {  /*Example I2C interface structure */
    BYTE bStat;
    BYTE bCmd;
    int  iVolts;
    char cStr[6]; / * Read only string */
} MyI2C_Regs;
```

This structure may contain any group of variables with any name as long as it is contiguous in memory and referenced by a pointer. The interface (I²C Master) only sees it 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. The first parameter sets the size of the exposed memory to the I²C interface, in this case it is the entire structure. The second parameter sets the boundary between the read/write and read only areas by setting the number of bytes in the read/write area. The read/write area is first, followed by the read only area. In this case, only the first 4 bytes may be written to, but all bytes may be read by the I²C master. The third parameter is a pointer to the data.

```
EZI2C_SetBuffer1(sizeof(MyI2C_Regs), 4,(BYTE *) &MyI2C_Regs);
```

In the example below a 15 byte array is created and exposed to the I²C interface. The first 8 bytes of the array are read/write, and the remaining 7 bytes are read only.

```
char theArray[15];
EZI2C_SetBuffer2(15, 8, (BYTE *) theArray);
```

The example below is a very simple example where only a single integer (2 bytes) is exposed. Both bytes are readable and writable by the I²C master.

```
uint16 myVar;
EZI2C_SetBuffer1(2, 2, (BYTE *) (&myVar);
```

Interface as Seen by External Master

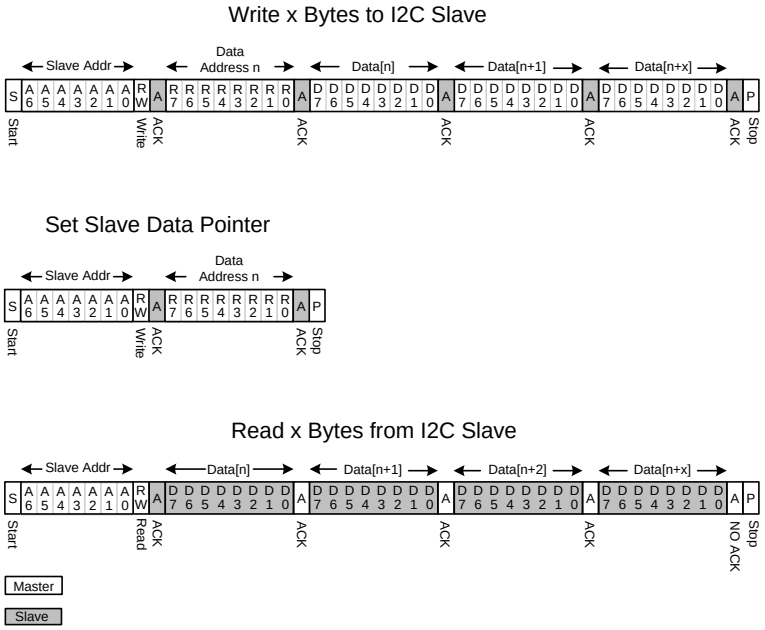
The EZ I²C Slave component supports basic read and write operations for the RAM area and read only operations for the FLASH area. The two buffer area interfaces contain separate data pointers that are set with the first one or two data bytes of a write operation, depending on the Sub_Address_Size parameter. For the rest of this discussion, we will concentrate on an 8-bit Sub_Address_Size.

When writing one or more bytes, the first data byte is always the data pointer. The byte after the data pointer is written into the location pointed to by the data pointer byte. The second data byte is written to the data pointer plus one and so on. This data pointer increments for each byte read or written, but is reset to the first value written at the beginning of each new read operation. A new read operation begins to read data at the location pointed to by the data pointer.

For example, if the data pointer is set to four, a read operation begins to read data at location four and continue sequentially until the end of the data or the host completes the read operation. For example, if the data pointer is set to four, each read operation resets the data pointer to four and reads sequentially from that location. This is true whether a single or multiple read operations are performed. The data pointer is not changed until a new write operation is initiated.

If the I²C master attempts to write data past the area specified by the SetBuffer1() function, 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.

The following diagram illustrates the bus communication for an 8-bit data write, data pointer write, and a data read operation. Remember that a data write operation always rewrites the data pointer.



At reset, or power on, the EZ I²C 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 here are available in the complete I²C specification available on the Philips web site, and by referring to the device data sheet supplied with PSoC Creator.

External Electrical Connections

As the block diagram illustrates, the I²C bus requires external pull up resistors. The pull up resistors (RP) are determined by the supply voltage, clock speed, and bus capacitance. Make the minimum sink current for any device (master or slave) no less than 3 mA at VOLmax = 0.4V for the output stage. This limits the minimum pull up resistor value for a 5V system to about 1.5 kΩ. The maximum value for RP depends upon the bus capacitance and clock speed. For a 5V system with a bus capacitance of 150 pF, the pull up resistors are no larger than 6 kΩ. For more information on "The I²C -Bus Specification", see the Philips web site at www.philips.com.

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

Interrupt Service Routine

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

DC and AC Electrical Characteristics

5.0V/3.3V DC and AC Electrical Characteristics

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

Component Changes

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

Version	Description of Changes	
1.20.e	Minor datasheet edit.	
1.20.d	Minor datasheet edit.	
1.20.c	Added silicon revision incompatibility message.	
1.20.b	Minor datasheet edit.	
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	The Configure dialog was updated.	Digital Port was changed to Pins component in the schematic.

© Cypress Semiconductor Corporation, 2008-2016. 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, 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.