EzI2C Slave Datasheet EzI2Cs V 1.20

Copyright © 2004-2015 Cypress Semiconductor Corporation. All Rights Reserved.

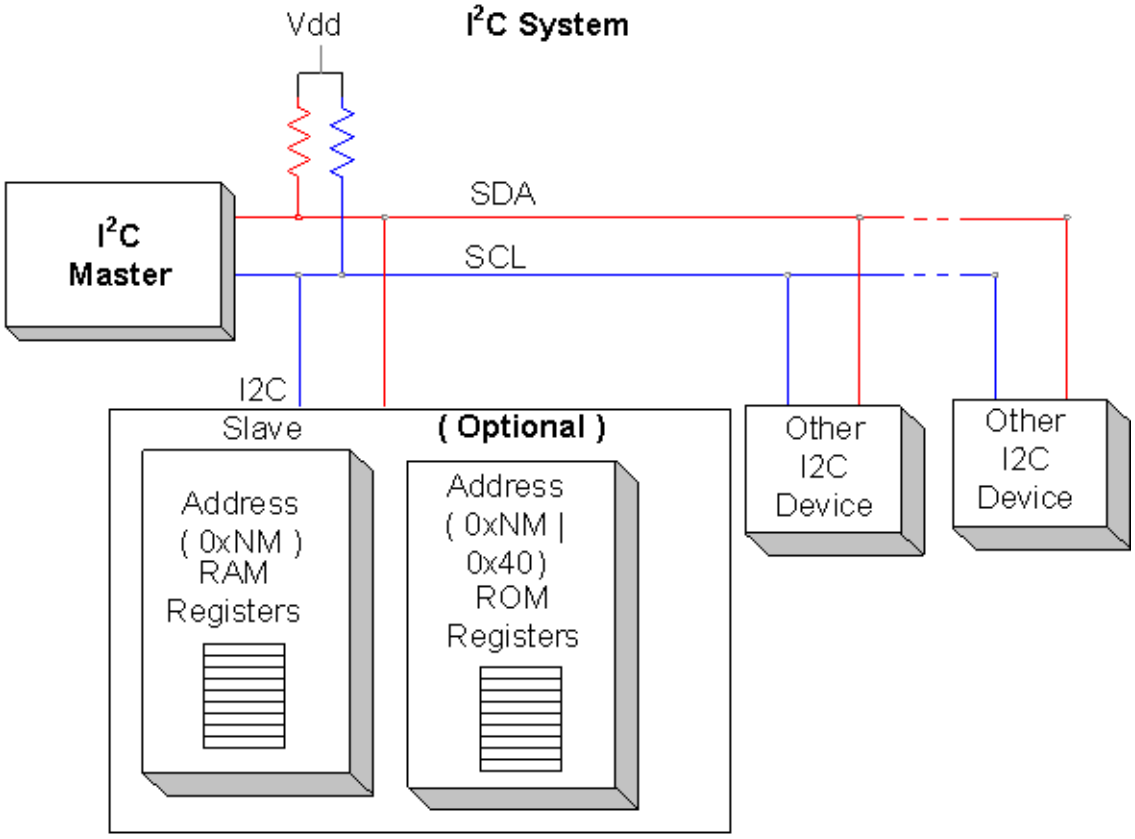
Resources	PSoC® Blocks			API Memory (Bytes)		Pins (per External I/O)
	Digital	Analog CT	Analog SC	flash	RAM	
CY8C20x34, CY8C20x24						
RAM Read/Write Buffers	0	0	0	264	6	2
Additional for ROM Buffer Support	0	0	0	379	6	2

Features and Overview

- Industry standard Philips I^2C bus compatible interface
- Emulates common I^2C EEPROM interface
- Only two pins (SDA and SCL) required to interface to I^2C bus
- Standard data rate of 100/400 kbps
- High level API requires minimal user programming

The EzI2Cs User Module implements an I^2C register-based slave device. The I^2C bus is an industry standard, two wire hardware interface developed by Philips®. The master initiates all communication on the I^2C bus and supplies the clock for all slave devices. The EzI2Cs User Module supports the standard mode with speeds up to 400 kbps. No digital or analog PSoC blocks are consumed with this module. The EzI2Cs User Module is compatible with multiple devices on the same bus.

Figure 1. I²C Block Diagram



Single I2C Slave Device (Address 0xNM)
Shown with RAM registers and optional ROM registers

Functional Description

The EzI2Cs User Module takes a different approach from the I2CHW User Module provided with PSoC Designer. This user module supports only a I²C slave configuration with one or two I²C addresses. The first address is always the RAM allocated area and the optional second address accesses the ROM area. The second address is the RAM area address OR'ed with 0x40. For example, if the user selects an address of 0x05, the RAM area address is 0x05 and the optional second address is 0x45. Both address are right justified. Both the RAM and ROM area may have data structures (bytes, ints, arrays, structures) from 1 to 255 bytes.

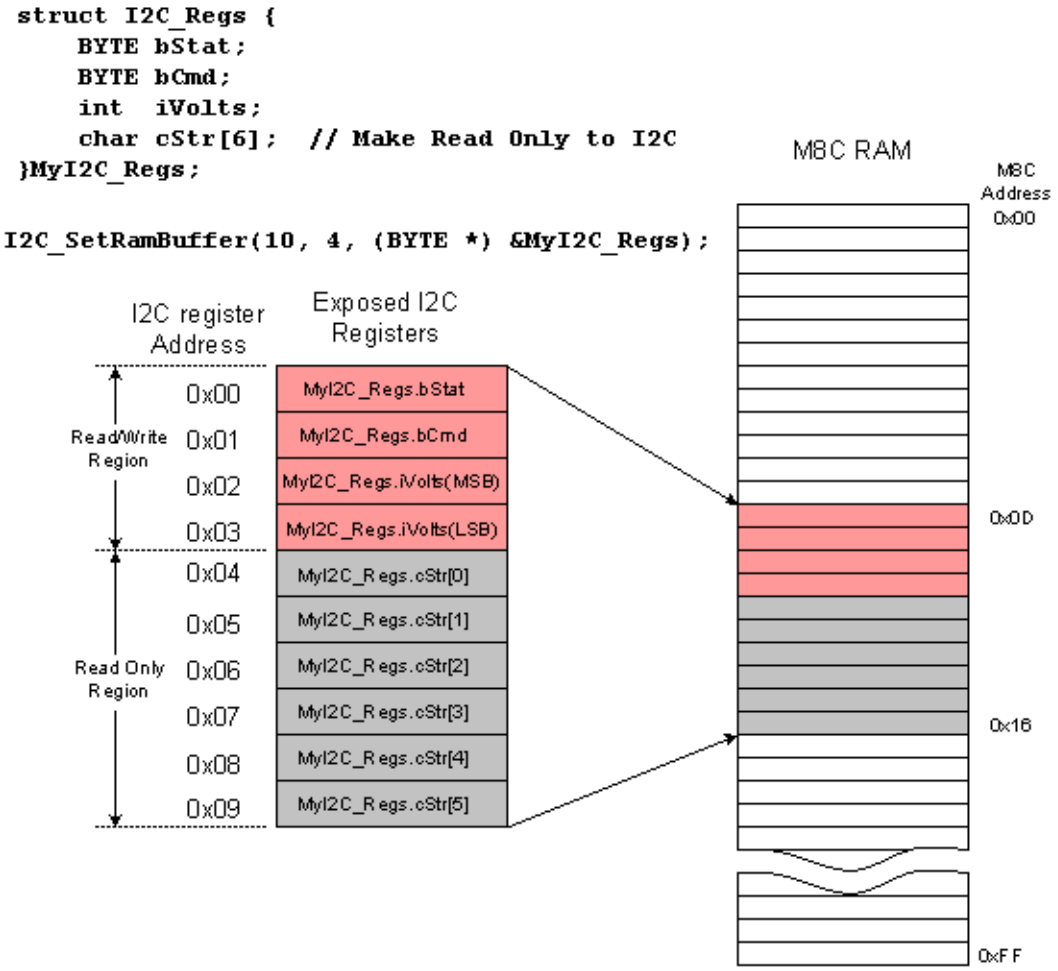
This user module requires that you enable global interrupts since the I²C hardware is interrupt driven. Even though this user module requires interrupts, you do not need to add any code to the ISR (Interrupt Service Routine). The module services all interrupts (data transfers) independent of your code. 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.

RAM Area Interface

To an I²C master the interface looks very similar to a common 256 byte I²C EEPROM. The PSoC API is treated as RAM that can be configured as simple variables, arrays, or structures. In a sense it acts as a shared RAM 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 user module only allows the I²C master to access the specified area of RAM 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. See the following diagram.

Figure 2. RAM Area Interface



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 given earlier, a supplied API is

used to expose the data structure to the I²C interface. The first parameter sets the size of the exposed RAM 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.

```
EzI2Cs_SetRamBuffer(sizeof(MyI2C_Regs), 4,(BYTE *) &MyI2C_Regs);
```

In the following example, 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];
EzI2Cs_SetRamBuffer(15, 8, (BYTE *) theArray);
```

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

```
int iRegister;
EzI2Cs_SetRamBuffer(2, 2, (BYTE *) (&iRegister);
```

ROM Area Interface (Optional)

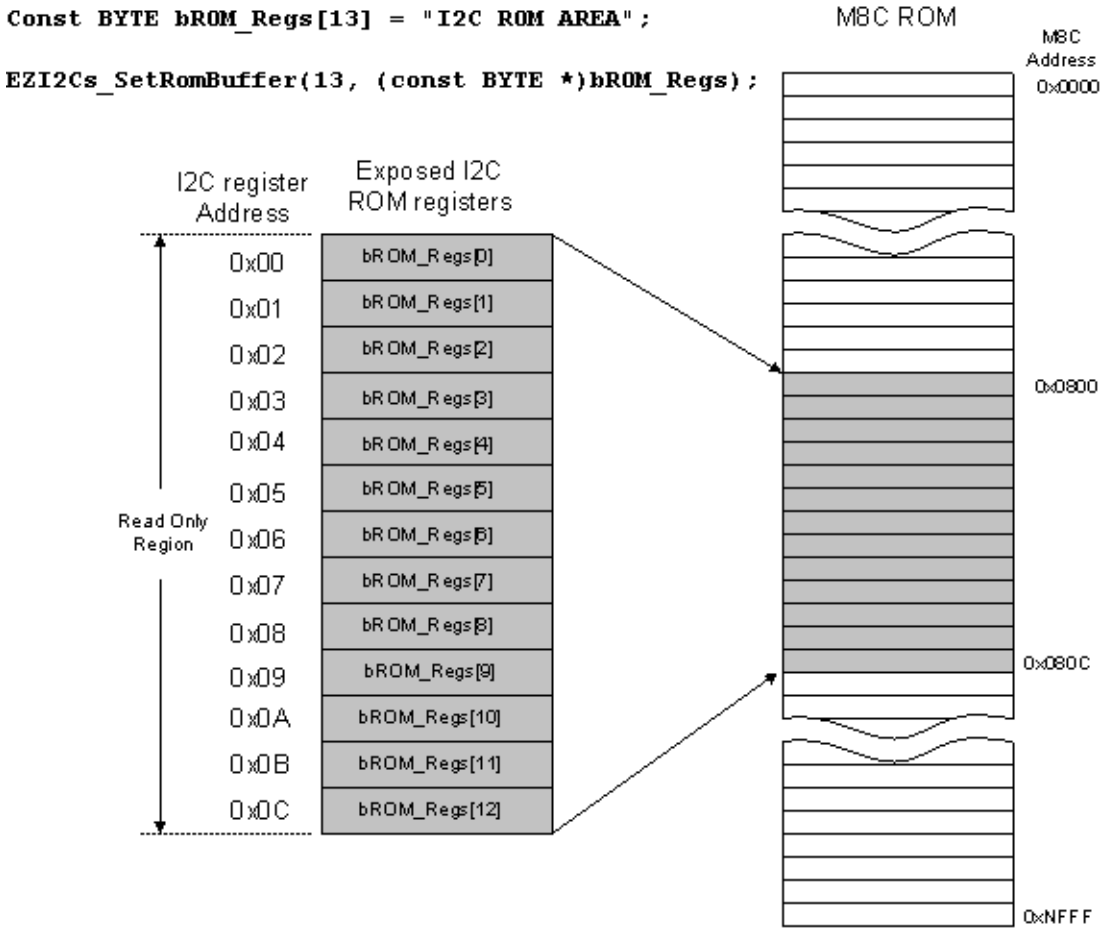
The ROM interface is almost identical to the RAM interface, except that it is read only. As seen in the following example, the SetRomBuffer function is similar to the SetRamBuffer function. In this example the data area is a simple constant string 13 bytes in length counting the terminating NULL (0x00) character.

```
Const BYTE bROM_Regs[13] = "I2C ROM AREA";

EZI2Cs_SetRomBuffer(13, (const BYTE *)bROM_Regs);
```

As mentioned before, the ROM area has it's own I^2C address. This address is simply the RAM area address OR'ed with 0x40. If the main RAM area address is 0x05, the I^2C address to access the ROM area would be 0x45. The following diagram shows how this example may appear in memory.

Figure 3. ROM Area Interface



Interface as Seen by External Master

The EzI2Cs User Module supports basic read and write operations for the RAM area and read only operations for the ROM area. The RAM and ROM area interfaces contain separate data pointers that are set with the first data byte of a write operation. Even the ROM area accepts a single byte write to set the data pointer. When writing one or more RAM 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 third byte (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.

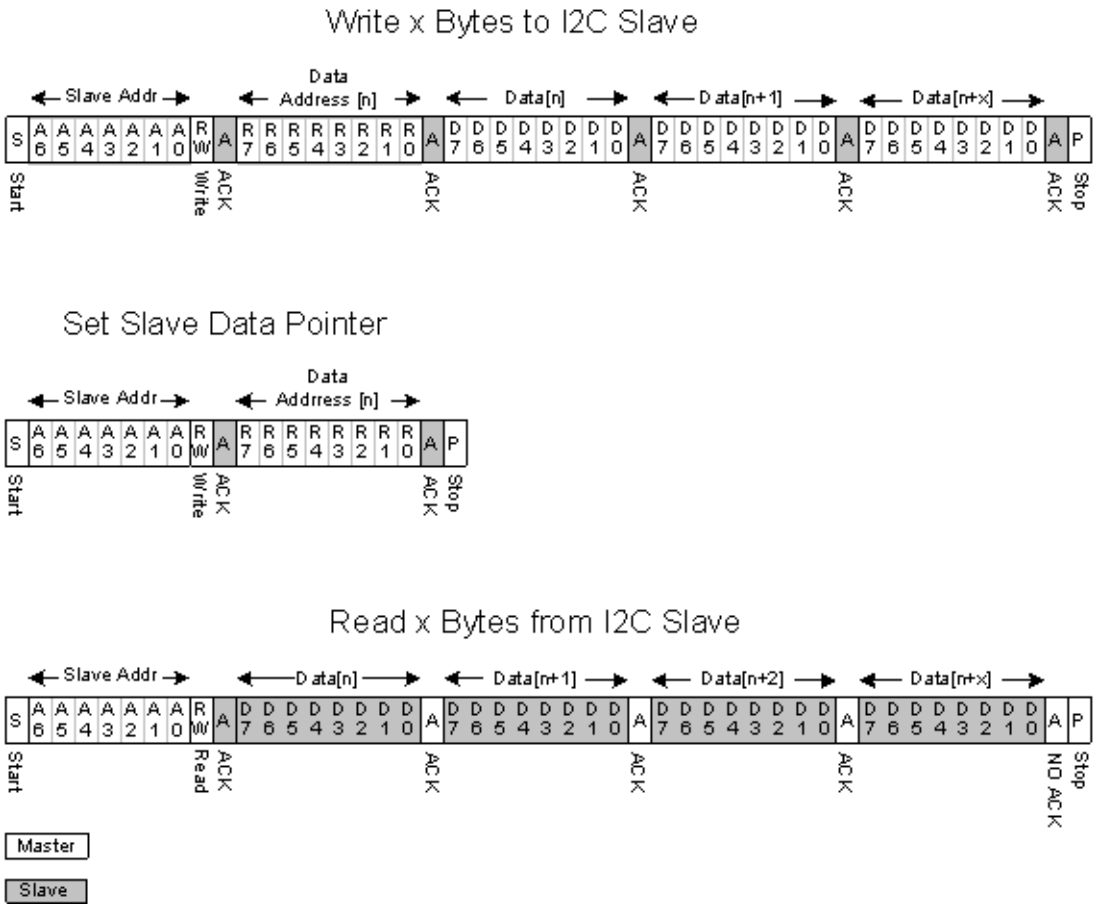
For example, if the data pointer is set to four, a read operation begins to read data at location four and continue sequentially until the end of the data or the host completes the read operation. For example, if the data pointer is set to four, each read operation resets the data pointer to four and 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 I^2C master attempts to write data past the area specified by the SetRamBuffer() 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.

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

Figure 4. Bus Communication



At reset, or power on, the EzI2Cs User Module is configured and APIs are supplied, but the resource must be explicitly turned on using the EzI2Cs_Start() function.

The I^2C resource supports data transfer at a byte-by-byte level. At the end of each address or data transmission/reception, status is reported or a dedicated interrupt may be triggered. Status reporting and interrupt generation depends upon the direction of data transfer and the condition of the I^2C bus as detected by the hardware. Interrupts may be configured to occur on byte-complete, or bus-error detection.

Every I^2C transaction consists of a Start, Address, R/W Direction, Data, and a Termination. For an I^2C slave, an interrupt occurs after the eighth bit of incoming data. At this point a receiving device must decide to acknowledge (ack) or not-acknowledge (nak) the incoming byte whether it is an address or data. The receiving device then writes appropriate control bits to the I2C_SCR register, informing the I^2C resource of the ack/nak status. The write to the I2C_SCR register paces data flow on the bus by installing the bus, placing the ack/nak status on the bus and shifting the next data byte in. For the second case of a transmitter, an interrupt occurs after an external receiving device provided an ack or nak. The I2C_SCR may be read to determine the status of this bit. For a transmitter, data is loaded into the I2C_DR register and the I2C_SCR register is again written to trigger the next portion of the transmission.

Detailed descriptions of the I^2C bus and the implementation here are available in the complete I^2C specification available on the Philips web site, and by referring to the device datasheet supplied with PSoC Designer.

Design Considerations

Set the main oscillator to any clock speed. Data throughput may be affected by processor speed but byte-by-byte data transfer functions at the specified I^2C speed.

Dynamic Reconfiguration

Do not incorporate the EzI2Cs resource into dynamically loaded or unloaded overlays. Place the EzI2Cs resource as part of the base configuration only. Operation of the EzI2Cs block may be modified as operational requirements dictate, but attempting to remove the resource as part of dynamic reconfiguration may result in adverse effects on external I^2C devices. It is also possible to accidentally reconfigure pins selected to disable the I^2C function. Take care to avoid this possibility when switching between configurations.

External Electrical Connections

As the block diagram 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 3 mA at V_{OLmax} = 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 R_P 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 I^2C-Bus Specification", see the Philips web site at www.philips.com.

Note Purchase of I^2C components from Cypress or one of its sublicensed Associated Companies, conveys a license under the Philips I^2C Patent Rights to use these components in an I^2C system, provided that the system conforms to the I^2C Standard Specification as defined by Philips.

Placement

The EzI2Cs User Module does not require any digital or analog PSoC blocks. There are no placement restrictions. Multiple placements of I^2C modules is not possible since the I^2C module uses a dedicated PSoC resource block and interrupt.

Parameters and Resources

Slave_Addr

Selects the 7-bit slave address that is used by the I^2C master to address the slave. Valid selections are from 0x00 to 0x7F if only the RAM registers are used. If the ROM registers are enabled, only addresses between 0x00 and 0x3F are valid, because the ROM register space is the RAM address plus 0x40 (0x40 to 0x7F).

Address_Type

Selects whether the address is static or dynamic. If set to static you cannot change the I^2C address, but saves one byte of RAM. Setting this option to dynamic allows the firmware to change the address at any time. This option is generally for applications where external resistors or dip switches are used to set the LSBs of the I^2C address.

ROM_Registers

The RAM area is always enabled, but you may enable a second address that allows you to access of constants in the ROM area. Choose the enable option for this parameter if the ROM address is desired, if not choose disable.This second address is the base address OR'ed with 0x40.

I2C_Clock

This parameter selects the clock speed used with this slave. Options are 50 kHz, 100 kHz, or 400 kHz. The 400 kHz speed can only be used when IMO (SysClk) is set to 12 MHz.

I2C_Pin

Selects the pins from Port 1 for use with I^2C signals. There is no need to select the proper drive mode for the pins, PSoC Designer does this automatically. This allows you to place the I2C clock and data signals on P1[5] - P1[7] or P1[1] - P1[0].

Application Programming Interface

Note

In this, as in all user module APIs, the values of the A and X register may be altered by calling an API function. It is the responsibility of the calling function to preserve the values of A and X before the call if those values are required after the call. This "registers are volatile" policy was selected for efficiency reasons and has been in force since version 1.0 of PSoC Designer. The C compiler automatically takes care of this requirement. Assembly language programmers must ensure their code observes the policy, too. Though some user module API functions may leave A and X unchanged, there is no guarantee they may do so in the future.

For Large Memory Model devices, it is also the caller's responsibility to preserve any value in the CUR_PP, IDX_PP, MVR_PP, and MVW_PP registers. Even though some of these registers may not be modified now, there is no guarantee that will remain the case in future releases.

API Variables

To guarantee data integrity of variables with length 2 or more bytes check the EzI2Cs_bBusy_Flag variable before changing values of such variables in code, for example:

```
EzI2Cs_DisableInt();
if(!(EzI2Cs_bBusy_Flag == EzI2Cs_I2C_BUSY_RAM_READ))
{
data.wData1++; //safely increment some 2 byte variable
}
```

```
EzI2Cs_ResumeInt();
```

Note that interrupts must be disabled to prevent an I2C read transaction from starting after the flags are checked but before the data is modified. With interrupts disabled, if an I2C transaction does start during this time, clock stretching will be used on the I2C bus until after the interrupt is resumed.

The EzI2Cs_bBusy_Flag variable is updated automatically in ISR and has following pre-defined values:

Symbol Name	Value	Description
I2C_FREE	0x00	No transaction at the current moment
I2C_BUSY_RAM_READ	0x01	RAM read transaction in progress
I2C_BUSY_RAM_WRITE	0x02	RAM write transaction in progress
I2C_BUSY_ROM_READ	0x04	ROM read transaction in progress
I2C_BUSY_ROM_WRITE	0x08	ROM write transaction in progress

Predefined values of EzI2Cs_bBusy_Flag variable

API Functions

EzI2Cs_Start

Description:

Enables the I^2C slave and enable interrupts. Call the required EzI2Cs_SetRamBuffer() function before this function call.

C Prototype:

```
void  EzI2Cs_Start(void);
```

Assembler:

```
lcall EzI2Cs_Start
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

EzI2Cs_SetAddr

Description:

This function sets the address that is recognized by the EzI2Cs User Module. This command is only valid if the Address_Type is set to Dynamic in the parameters. Call this function only after the EzI2Cs_Start() function. This is because the start function sets the default address and this function overwrites that address. This function is optional and only needed if you change the address from the default.

C Prototype:

```
void  EzI2Cs_SetAddr(BYTE bAddr);
```

Assembler:

```
mov    A,0x05             ; Set I2C slave address to 0x05
lcall  EzI2Cs_SetAddr
```

Parameters:

BYTE char: Slave address between 1 and 127.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

EzI2Cs_GetActivity

Description:

This function returns a nonzero value if an I^2C read or write cycle occurred since last time this function was called. The activity flag resets to zero at the end of this function call.

C Prototype:

```
BYTE EzI2Cs_GetActivity(void);
```

Assembler:

```
lcall  EzI2Cs_GetActivity     ; Return value in A
```

Parameters:

None.

Return Value:

Returns a nonzero value if an I^2C read or write has occurred. Zero if no activity has occurred since the last time the function was called. Symbolic constants are provided in C and assembly. Their values are defined as follows:

Symbol	Value	Meaning
EzI2Cs_ANY_ACTIVITY	0x80	Either a write cycle or a read cycle occurred
EzI2Cs_READ_ACTIVITY	0x20	A read cycle occurred
EzI2Cs_WRITE_ACTIVITY	0x10	A write cycle occurred

Side Effects:

The A and X registers may be altered by this function.

EzI2Cs_GetAddr

Description:

This function returns the current address of this I^2C slave. This function is optional and is usually only required if the application must determine the current or default address.

C Prototype:

```
BYTE EzI2Cs_GetAddr(void);
```

Assembler:

```
lcall  EzI2Cs_GetAddr     ; Return value in A
```

Parameters:

None.

Return Value:

The current I^2C address of this I^2C slave

Side Effects:

The A and X registers may be altered by this function.

EzI2Cs_Stop

Description:

Disables the EzI2Cs by disabling the I^2C interrupt and disabling the slave functionality of the I^2C block, to re-enable the slave, call EzI2Cs_Start().

C Prototype:

```
void   EzI2Cs_Stop(void);
```

Assembler:

```
lcall  EzI2Cs_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

EzI2Cs_DisableSlave

Description:

Disables the EzI2Cs without disabling the I^2C interrupt.

C Prototype:

```
void   EzI2Cs_DisableSlave(void);
```

Assembler:

```
lcall  EzI2Cs_DisableSlave
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

EzI2Cs_EnableInt

Description:

Enables I^2C interrupt allowing start condition detection. Remember to call the global interrupt enable function by using the macro: M8C_EnableGInt. This function is not required if the EzI2Cs_Start function is called. By default this function clears pending interrupts. To avoid clearing pending interrupts see the ResumeInt function.

C Prototype:

```
void   EzI2Cs_EnableInt(void);
```

Assembler:

```
lcall  EzI2Cs_EnableInt
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

EzI2Cs_ResumeInt

Description:

Enables I^2C interrupt allowing start condition detection. This function does not clear pending interrupts before enabling interrupts. Remember to call the global interrupt enable function by using the macro: M8C_EnableGInt. This function is not required if the EzI2Cs_Start function is called.

C Prototype:

```
void   EzI2Cs_ResumeInt(void);
```

Assembler:

```
lcall  EzI2Cs_ResumeInt
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

EzI2Cs_DisableInt

Description:

Disables I^2C's slave by disabling the I^2C interrupt.

C Prototype:

```
void   EzI2Cs_DisableInt(void);
```

Assembler:

```
lcall EzI2Cs_DisableInt
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

Side Effects:

The A and X registers may be altered by this function.

EzI2Cs_SetRamBuffer

Description:

This function sets the location and size of the RAM buffer (up to 255 bytes) that are exposed to an I^2C master. This function is required and called before calling EzI2Cs_Start().

C Prototype:

```
void EzI2Cs_SetRamBuffer(BYTE bSize, BYTE bRWboundry, (BYTE *)pAddr);
```

Assembler:

```
lcall  EzI2Cs_SetRamBuffer
```

Parameters:

BYTE bSize: Size of the data structure exposed to an I^2C master. The minimum value for bSize is 1 and the maximum is 255. BYTE bRWboundry: The size of the writable locations starting with the first location. This value should always be less than or equal to bSize for proper operation. BYTE * pAddr: A pointer to the data structure.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function. If bRWboundry is set to a value larger than bSize, the entire bRWboundry size is writable. Setting bSize to the invalid value of 0 may allow the I2C host to write to undefined memory locations.

EzI2Cs_SetRomBuffer

Description:

This function sets the location and size of the ROM buffer (up to 255 bytes) that is exposed to an I^2C master. This function is required before enabling the EzI2Cs_Start function if ROM use is desired.

C Prototype:

```
void EzI2Cs_SetRomBuffer(BYTE bSize,(const BYTE *)pAddr);
```

Assembler:

```
lcall  EzI2Cs_SetRomBuffer
```

Parameters:

BYTE bSize: Size of the data structure exposed to an I^2C master. const BYTE * pAddr: A pointer to the data structure in flash ROM.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

Sample Firmware Source Code

Example C Code

```
//-------------------------------------------------------
// Example code to demonstrate the use of the EzI2Cs UM
//
// This code sets up the EzI2Cs slave to expose both
// a RAM and a ROM data structure.  The RAM structure is
// addressed at I2C address 0x05, and the ROM structure
// is at I2C address 0x45.
//
// * The instance name of the EzI2Cs User Module is "EzI2Cs".
// * The "Address_Type" parameter is set to "Dynamic"
// * The "ROM_Registers" parameter is set to "Enable"
//
// NOTE: to guarantee data integrity of variables with length 2 or more
// bytes check the EzI2Cs_bBusy_Flag variable before changing values of such
// variables.
//-------------------------------------------------------

#include <m8c.h>        // Part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules

struct I2C_Regs {    // Example I2C interface structure
    BYTE bStat;
    BYTE bCmd;
```

```
    int  iVolts;
    char cStr[6];    // Read only string
} MyI2C_Regs;

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

void main(void)
{

    // Set up RAM buffer
   EzI2Cs_SetRamBuffer(sizeof(MyI2C_Regs), 4, (BYTE *) &MyI2C_Regs);


   EzI2Cs_SetRomBuffer(sizeof(DESC), DESC);   // Set up ROM buffer

   M8C_EnableGInt ;                           // Turn on interrupts

   EzI2Cs_Start();                            // Turn on I2C
   EzI2Cs_SetAddr(5);                         // Change address to 5


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

Example ASM Code:

```
;-------------------------------------------------------------------------
; Example code to demonstrate the use of the ExI2Cs UM
;
; This code sets up the EzI2Cs slave to expose both
; a RAM and a ROM data structure.  The RAM structure is
; addressed at I2C address 0x05, and the ROM structure
; is at I2C address 0x45.
;
; * The instance name of the EzI2Cs User Module is "EzI2Cs".
; * The "Address_Type" parameter is set to "Dynamic".
; * The "ROM_Registers" parameter is set to "Enable".
;
; NOTE: to guarantee data integrity of variables with length 2 or more
; bytes check the EzI2Cs_bBusy_Flag variable before changing values of such
; variables.
;-------------------------------------------------------------------------

include "m8c.inc"       ; part specific constants and macros
include "memory.inc"    ; Constants & macros for SMM/LMM and Compiler
include "PSoCAPI.inc"   ; PSoC API definitions for all User Modules

area bss (RAM, REL, CON)        ; Allocate Variables & define constants
RAM_BUF_SIZE:     equ  10       ; Top of ramp value
BUF_RW_SIZE:      equ   5       ; Only first five bytes writable from I2C Master

I2C_RAM_Buf:  blk  RAM_BUF_SIZE ; Reserve RAM for I2C buffer
```

```
area text (ROM, REL, CON)

.LITERAL
I2C_ROM_BUF:                    ; I2C ROM Buffer
   DB  11h, 22h, 33h, 44h, 55h, 66h, 77h, 88h
.ENDLITERAL
ROM_BUF_SIZE:     equ   8

export _main

_main:
    ; Set RAM Buffer
    mov    A,>I2C_RAM_Buf       ; Save MSB of RAM buffer address
    push   A
    mov    A,<I2C_RAM_Buf       ; Save LSB of RAM buffer address
    push   A
    mov    A,BUF_RW_SIZE        ; Save RW size param
    push   A
    mov    A,RAM_BUF_SIZE       ; Save I2C buffer size
    push   A
    call   EzI2Cs_SetRamBuffer
    ADD    SP,-4                ; Reset Stack

  ; Set ROM Buffer
    mov    A,>I2C_ROM_BUF       ; Save MSB of ROM buffer address
    push   A
    mov    A,<I2C_ROM_BUF       ; Save LSB of ROM buffer address
    push   A
    mov    A,ROM_BUF_SIZE       ; Save ROM buffer size
    push   A
    call   EzI2Cs_SetRomBuffer
    add    SP,-3                ; Reset Stack

  M8C_EnableGInt              ; Enable Global Interrupts

  call   EzI2Cs_Start         ; Start and enable IRQ
    mov    A,5                  ; Set address to 5
    call   EzI2Cs_SetAddr

.Loop:

    ;; User Code goes here

  jmp    .Loop
```

Configuration Registers

This section describes the PSoC Resource Registers used or modified by the EzI2Cs User Module. The use of these registers is not required when using the EzI2Cs User Module, but is provided as a reference.

Table 1. Resource I2C_CFG: Bank 0 reg[D6] Configuration Register

Bit	7	6	5	4	3	2	1	0
Value	Reserved	PinSelect	Bus Error IE	Stop IE	Clock Rate[1]	Clock Rate[0]	0	Enable Slave

Pin Select: selects either SCL and SDA as P1[5] - P1[7] or P1[1] - P1[0].

Bus Error Interrupt Enable: enable I^2C interrupt generation on a Bus Error.

Stop Error Interrupt Enable: enable an I^2C interrupt on an I^2C Stop condition.

Clock Rate[1,0]: select among three valid Clock rates 50, 100, 400kbps.

Enable Slave: enable the I^2C HW block as a bus Slave.

Table 2. Resource I2C_SCR: Bank 0 reg[D7] Status Control Register

Bit	7	6	5	4	3	2	1	0
Value	Bus Error	NA	Stop Status	ACK out	Address	Transmit	Last Recd Bit (LRB)	Byte Complete

Bus Error: indicates the detection of a Bus Error condition.

Stop Status: indicates the detection of an I^2C stop condition.

ACK out: direct the I^2C block to Acknowledge (1) or Not Acknowledge (0) a received byte.

Address: received or transmitted byte is an address.

Last Received Bit (LRB): the value of last received bit (bit 9) in a transmit sequence, status of Ack/Nak from destination device.

Byte Complete: 8 data bits were received. For Receive Mode, the bus is stalled waiting for an Ack/Nak. For Transmit Mode Ack Nak was also received (see LRB) and the bus is stalled for the next action.

Table 3. Resource I2C_DR: Bank 0 reg[D8] Data Register

Bit	7	6	5	4	3	2	1	0
Value	Data							

Received or Transmitted data. To transmit data, you must load this register before a write to the I2C_SCR register. Received data is read from this register and may contain an address or data.

Version History

Version	Originator	Description
1.1	DHA	Removed the default pin value of the "I^{2}C Pin" parameter, because it corrupts the drive mode of the corresponding pins initially set by other user modules. Updated to prevent SCL getting stuck to low. The following changes were done to the Start function: 1. Changed Initial Open-Drain Low drive mode of user module pins to HI-Z analog. 2. Enabled the I^{2}C block. 3. Gave delay 5 nop instructions. 4. Restored the Initial I^{2}C pin drive mode. Made EzI2C_StopSlave API public.
1.20	DHA	Added user code sections in the EzI2Cs UM ISR file.
1.20.b	DHA	1. Updated the EzI2Cs_Stop() and EzI2Cs_DisableInt() API function descriptions. 2. Updated predefined values of the EzI2Cs_bBusy_Flag variable and corrected the example code of the EzI2Cs_bBusy_Flag usage. 3. Updated comments for the EzI2Cs_I2C_BUSY_ROM_READ and EzI2Cs_I2C_BUSY_ROM_WRITE constants definitions.

Note PSoC Designer 5.1 introduces a Version History in all user module datasheets. This section documents high level descriptions of the differences between the current and previous user module versions.

Copyright © 2004-2015 Cypress Semiconductor Corporation. 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 Designer™ and Programmable System-on-Chip™ are trademarks and PSoC® is a registered trademark 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.