I²C Hardware Block Datasheet I2CSBUF V 1.00

Copyright © 2012-2013 Cypress Semiconductor Corporation. All Rights Reserved.

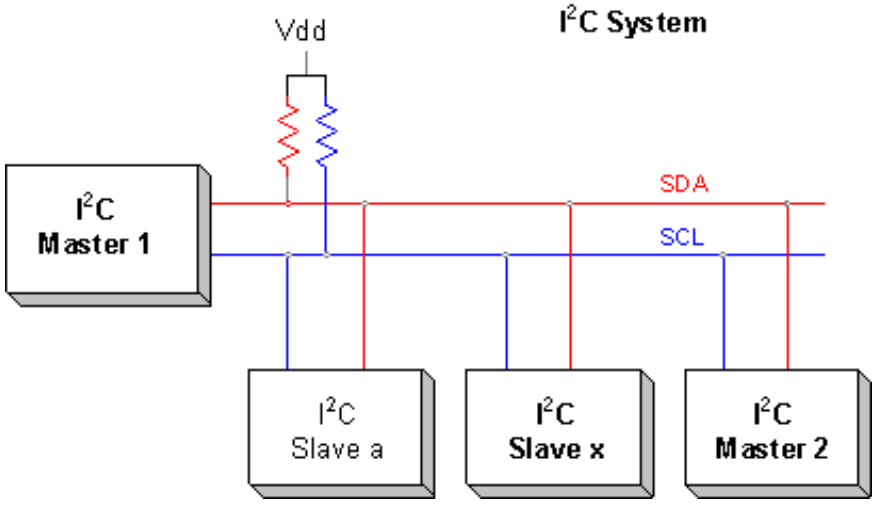
Resources	PSoC® Blocks				API Memory (Bytes)		Pins per Sensor
	CapSense®	I2C/SPI	Timer	Comparator	Flash	RAM	
CY8C20xx7/S, CY8C20055, CY8C24093, CY8C24393, CY8C24293, CY8C24693							
Slave Only	–	1	–	–	213	1	2

Features and Overview

- Industry standard Philips I²C bus compatible interface
- Slave-only operation
- Only two pins (SDA and SCL) are required to interface to I²C bus
- Standard data rate of 50, 100, and 400 kbps
- No clock stretching
- 32-byte hardware data buffer

The I2CSBUF User Module implements an I²C register based slave device. The I²C bus is an industry standard, two wire hardware interface developed by Philips® (now NXP). The master initiates all communication on the I²C bus and supplies the clock for all slave devices. The I2CSBUF user module supports the standard mode with speeds of as much as 400 kbps. The I2CSBUF user module is compatible with multiple devices on the same bus.

Figure 1. I²C Block Diagram



Functional Description

The I2CSBUF User Module employs an easy-to-use data buffer accessible externally as an I2C slave. The buffer is accessed with simple write and read transactions carried out by the I2C master. Because the buffer is implemented in hardware, no clock stretching by the I2C slave is required.

At the end of each address or data transmission/reception, a status is reported in PSoC registers and a dedicated interrupt may be triggered. Status reporting and interrupt generation depend on data transfer direction and the condition of the I2C bus, as detected by the hardware. Interrupts may be configured to occur on bytecomplete, address match and stop condition detection.

Every I2C transaction consists of a start, address, R/W direction, data, and a termination. The I2C resource used for this module can operate only as an I2C slave. Communication is initiated from a foreground function call.

The I2C interface allows the master to perform writes and reads in the following way. To start, the master sets the base data pointer. It does that by transmitting the slave address and a write bit, then the desired data pointer. This sets the user module's data pointer and ensures that it will read from or write to the hardware buffer starting at that location.

To read from the buffer, the master then transmits the slave's address, followed by a read bit. It then ACKs each slave transaction until it has read as many bytes as it wants, and then NAKs, and initiates a stop condition. An example for read transmission is provided below ('W' indicates write,'R' indicates read, 05 is slave address, 'DP' indicates data pointer, 'X' indicates read a byte and 'P' indicates stop condition):

W 05 DP P

R 05 x x x P

To write to the buffer, initiate write traffic and set the base data pointer (I2C_BP). Then, the host transmits the data bytes to be written to I2C buffer and initiates a stop condition at the end of data transfer.

An example for write transmission is provided below ('W' indicates write, 'R' indicates read, 05 is slave address, 'DP' indicates data pointer, 'X' indicates read a byte and 'P' indicates stop condition)

W 05 DP <Write Byte 1> <Write Byte 2> ... <Write Byte N> P

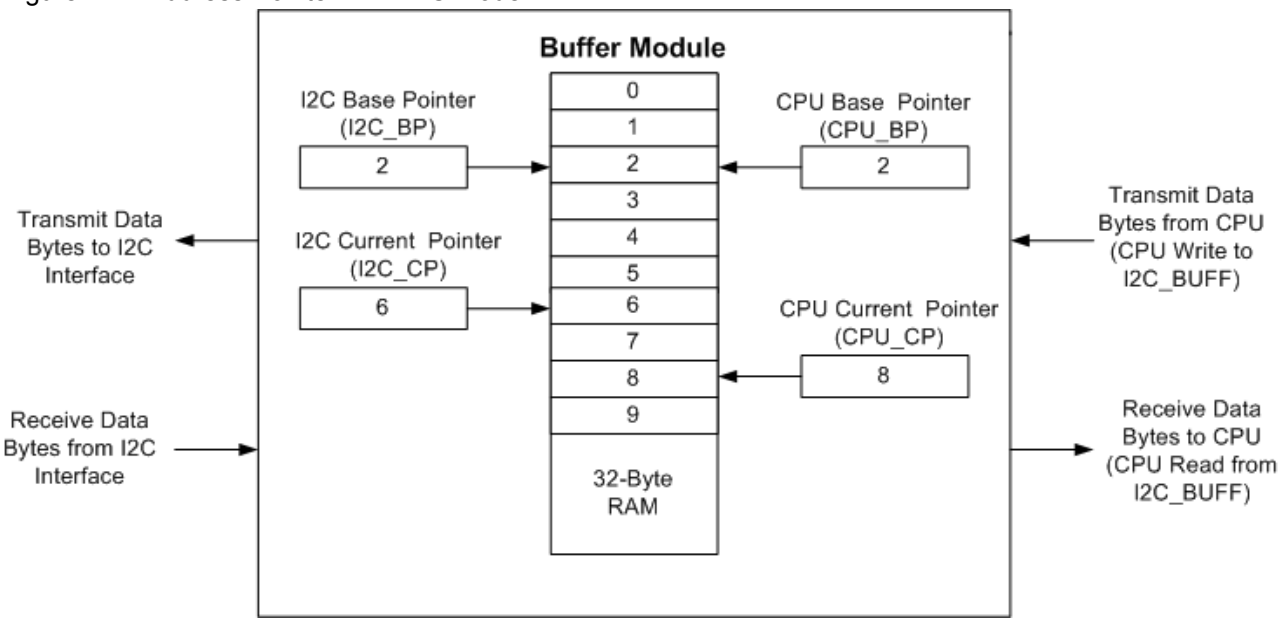
When writing one or more RAM bytes, the first data byte is always the base datapointer.There is another data pointer named current data pointer which increments for each byte read or written (I2C_CP), but it is reset to the base data pointer upon every address match event or write to base data pointer. The byte after the base data pointer is written into the location indicated by the current data pointer. The third byte (second data byte) is written to the current data pointer plus one, and so on. A new read operation begin store a data at the location indicated by the base data pointer (current base pointer reset to base pointer on address match event).

For example, if the data pointer is set to four, each read operation (new read operation with slave address) resets the data pointer to four and reads sequentially from that location until the end of the data buffer or until the host completes the read operation. This holds true whether a single read operation or multiple ones are performed. The base data pointer is not changed until you initiate a new write operation.

The following diagram illustrates how to configure the address pointers in I2CSBUF user module. In this example, the external master sent a start, slave address, and a data bye of 2 to initialize both the base

address pointer (I2C_BP register) and the current address pointer (I2C_CP register). Then, 4 bytes were written, OR a start or restart was sent with the device address, and 4 bytes were read. On the CPU side, a 2 was written to the CPU base address register (CPU_BP register), and 6 subsequent bytes were read from or written to the I2C_BUFF register by the CPU.

Figure 2. Address Pointer in EZI2C Mode



Dynamic Reconfiguration

Cypress strongly recommends against incorporating the I2CSBUF resource into dynamically loaded/unloaded overlays. Place the I2CSBUF resource as part of the base configuration only. Modify the operation of the I2CSBUF block as operational requirements dictate, but trying to remove the resource as part of dynamic reconfiguration may adversely affect external I^2C devices.

I^2C Addressing

I^2C addresses are contained in the upper 7 bits of the first byte of a read or write transaction. This byte is used by the I^2C master to address the slave. Valid selections are from 0-127(dec). The LSb of the byte contains the R/~W bit. If this bit is 0, the address is written to; if the LSb is 1, then the addressed slave has data read from it.

Internally, the user module takes the input address, shifts it, and combines it with a read/write bit to construct a complete address byte.

Example

An address of 0x48 is passed as a parameter or defined as a slave address. A separate parameter is passed containing read/write information. An I^2C master sends a byte (8-bits) of 0x90 to write data to the slave and the byte 0x91 to read data from the slave.

Since the slave module accepts decimal based numeric input for its address parameter, type the 7-bit address in decimal (decimal 72).

Simultaneous Access of I2C Buffer by CPU and I2C Master

Do not access the I2C hardware buffer external master and CPU simultaneously as this can lead to data corruption. A semaphore mechanism is provided to prevent the simultaneous access of buffer by CPU and master. When the CPU needs to access the buffer (read or write), it needs to check whether the buffer is being accessed by the host. This can be checked in the 'I2CSBUF_BUS_BUSY' flag using 'I2CSBUF_bCheckStatus' API. The bus busy flag is set on the address match event and is reset on stop condition or repeated at start condition. CPU should set the auto NACK mode if the bus is free and before accessing the buffer, this prevents host access to the buffer. If the host tries to access the buffer at the same time, the I2C block will provide the NACK signal. After CPU access to the buffer is completed, auto NACK should be cleared and buffer must be released for host access. The section “Sample Firmware Source Code” provides a code snippet for using the semaphore mechanism efficiently.

Sleep and I2C Operation

CY8C20xx7/7S family of device supports wake up on I2C slave address match event. This feature helps to wake the device up from sleep mode when host is tried to access CY8C20xx7/7S device. To make this feature work, power to I2Cblock should be turned on by setting I2C_ON bit in SLP_CFG2 register and auto NACK feature should be set before entering into sleep. Upon the address match device will NACK address and also wake up from sleep (only if address is matched). Once device woken up from sleep, auto NACK will be automatically disabled. Example code is provided in section “Example C Code: wake up from sleep using I2C address match event”. Not setting Auto NACK for I2C block before entering sleep will corrupt traffic on I2C bus

DC and AC Electrical Characteristics

Refer to the device datasheet for your PSoC device for the electrical characteristics of the I^2C interface.

As the block diagram illustrates, the I^2C bus requires external pull-u.p resistors. The pull up resistors (R_P) are determined by the supply voltage, clock speed, and bus capacitance. The minimum sink current for any device (master or slave) is 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 on the bus capacitance and the 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 NXP web site at www.nxp.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 I2CSBUF User Module allows two choices for SCL and SDA. One choice has SCL on P1[7] and SDA on P1[5]. The other has SCL on P1[1] and SDA on P1[0]. If possible, use the P1[5]/P1[7] pins, because P1[0]/P1[1] are shared with ISSP and ECO. Multiple placements of I^2C modules are not possible, because the I^2C module uses a dedicated PSoC resource block and interrupt.

Parameters and Resources

All buffers are named according to their use by the I^2C master. For example a buffer with ‘Read’ in the name or description would be read by the I^2C master.

Slave_Addr

Slave_Addr selects the 7-bit slave address used by the I^2C master to address the slave. Valid selections are from 0 to 127 (decimal).

I2C Clock

Specifies the desired clock speed at which the I^2C interface runs. There are three possible clock rates:

- 50K Standard
- 100K Standard
- 400K Fast

I2C Pins

Selects the pins from Port 1 to use for I^2C signals. PSoC Designer selects the drive mode automatically. As a result, you can route the I^2C clock and data signals to P1[5] - P1[7] or P1[1] - P1[0].

Application Programming Interface

The API functions are provided as part of the user module to allow you to work with the user module at a higher level. This section specifies the interface to each function along with related constants supplied by the include files.

Each time a user module is placed, it is assigned an instance name. By default, PSoC Designer assigns I2CSBUF_1 to the first instance of this user module in a given project. This name can be changed to any unique value that follows the syntactic rules for identifiers. The assigned instance name becomes the prefix of every global function name, variable and constant symbol. In the following descriptions the instance name has been shortened to I2CSBUF for simplicity.

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. The calling function is responsible for preserving 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 meets that requirement. Assembly language programmers also must ensure their code observes the policy. 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, the caller is responsible for preserving 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.

I2CSBUF_Start

Description:

Starts the operation of the I2C hardware buffer slave.

C Prototype:

```
void  I2CSBUF_Start(void)
```

Assembler:

```
lcall I2CSBUF_Start
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

I2CSBUF_Stop

Description:

Stops the operation of the I^2C hardware buffer slave.

C Prototype:

```
void  I2CSBUF_Stop(void)
```

Assembler:

```
lcall I2CSBUF_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

I2CSBUF_EnableInt

Description:

Enables I^2C interrupt based on argument passed to API function.

C Prototype:

```
void  I2CSBUF_EnableInt(BYTE bIntMask)
```

Assembler:

```
mov   A, bIntMask
lcall I2CSBUF_EnableInt
```

Parameters:

bIntMask - set of flags of desired interrupt enabling. The symbolic names given in C and assembly, and their associated values, are listed in the following table:

Symbolic Name	Value
I2CSBUF_INT_ADDR	0x01
I2CSBUF_INT_STOP	0x02
I2CSBUF_INT_ADDR_STOP	0x03

Symbolic Name	Value
I2CSBUF_INT_BC	0x04
I2CSBUF_INT_ADDR_BC	0x05
I2CSBUF_INT_STOP_BC	0x06
I2CSBUF_INT_ADDR_STOP_BC	0x07

Return Value:

None

Side Effects:

See the note at the beginning of the API section.

I2CSBUF_DisableInt

Description:

Disables the I^2C interrupt based on argument passed to API function.

C Prototype:

```
void  I2CSBUF_DisableInt(BYTE bIntMask)
```

Assembler:

```
mov   A, bIntMask
lcall I2CSBUF_DisableInt
```

Parameters:

bIntMask - set of flags of desired interrupt enabling. The symbolic names given in C and assembly, and their associated values, are listed in the following table:

Symbolic Name	Value
I2CSBUF_INT_ADDR	0x01
I2CSBUF_INT_STOP	0x02
I2CSBUF_INT_ADDR_STOP	0x03
I2CSBUF_INT_BC	0x04
I2CSBUF_INT_ADDR_BC	0x05
I2CSBUF_INT_STOP_BC	0x06
I2CSBUF_INT_ADDR_STOP_BC	0x07

Return Value:

None

Side Effects:

See the note at the beginning of the API section.

I2CSBUF_bGetAddress

Description:

Returns the 7-bit I^{2}C address of the user module.

C Prototype:

```
BYTE  I2CSBUF_bGetAddress(void)
```

Assembler:

```
lcall I2CSBUF_bGetAddress
```

Parameters:

None

Return Value:

The 7 bit I^{2}C slave address value.

Side Effects:

See Note ** at the beginning of the API section.

I2CSBUF_SetAddress

Description:

Sets the 7 bit I^{2}C address of the user module to a user-defined value.

C Prototype:

```
void  I2CSBUF_SetAddress(BYTE bAddress)
```

Assembler:

```
mov   A, bAddress
lcall I2CSBUF_SetAddress
```

Parameters:

bAddress - the 7-bit I^{2}C slave address value

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

I2CSBUF_SetAutoNACK

Description:

Sets the respective bit to engage the auto NACK (also referred as force NACK) feature in I^{2}C slave block. Auto NACK may not be activated immediately and will be active only upon the next byte boundary.

C Prototype:

```
void  I2CSBUF_SetAutoNACK(void)
```

Assembler:

```
lcall I2CSBUF_SetAutoNACK
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

I2CSBUF_ClearAutoNACK

Description:

Clears the respective bit to disable auto NACK feature in I^2C slave block.

C Prototype:

```
void  I2CSBUF_ClearAutoNACK(void)
```

Assembler:

```
lcall I2CSBUF_ClearAutoNACK
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

I2CSBUF_WriteBuffer

Description:

Allows the user to write provided values to a certain range of the I^2C HW buffer. Make inputs the starting index of the buffer to write to, the length of data to be written, and a pointer to a location in memory to write from.

C Prototype:

```
void  I2CSBUF_WriteBuffer(BYTE bBufIndex, BYTE * pWriteData, BYTE bDataLen)
```

Assembler:

```
mov   A, bDataLen
push  A
mov   A, >pWriteData
push  A
mov   A, <pWriteData
push  A
mov   A, bBufIndex
push  A
lcall I2CSBUF_WriteBuffer
add   SP, -4
```

Parameters:

bBufIndex – starting index of the buffer to write to.

pWriteData – pointer to a location in memory to write from.

bDataLen – length of data to be written.

Return Value:

The number of bytes successfully written to hardware buffer.

Side Effects:

See Note ** at the beginning of the API section.

I2CSBUF_bReadBuffer

Description:

Allows you to read a certain range of values out of the I^2C HW buffer. Inputs should be: the starting index of the buffer to read from, length of data to be read, and a pointer to a location in memory to read to.

C Prototype:

```
BYTE  I2CSBUF_bReadBuffer(BYTE bBufIndex, BYTE *pReadData, BYTE bDataLen)
```

Assembler:

```
mov   A, bDataLen
push  A
mov   A, >pReadData
push  A
mov   A, <pReadData
push  A
mov   A, bBufIndex
push  A
lcall I2CSBUF_bReadBuffer
add   SP, -4
```

Parameters:

bBufIndex – starting index of the buffer to read from.

pReadData – pointer to a location in memory to read to.

bDataLen – length of data to be written.

Return Value:

The number of bytes successfully read from the hardware buffer into application memory.

Side Effects:

See Note ** at the beginning of the API section.

I2CSBUF_WriteBufferByte

Description:

Allows you to write a single byte to the hardware buffer. Inputs should be index of the buffer to write to, and a single byte to write to that location.

C Prototype:

```
void  I2CSBUF_WriteBufferByte(BYTE bBufIndex, BYTE bData)
```

Assembler:

```
mov   A, bBufIndex
mov   X, bData
lcall I2CSBUF_WriteBufferByte
```

Parameters:

bBufIndex – index of the buffer to write to.

bData – data byte to write to.

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

I2CSBUF_bReadBufferByte

Description:

Allows you to read a single byte from the hardware buffer. Input should be: the index of the buffer to write to. Should return the byte stored in that location.

C Prototype:

```
BYTE  I2CSBUF_bReadBufferByte(BYTE BufIndex)
```

Assembler:

```
mov   A, bBufIndex
lcall I2CSBUF_bReadBufferByte
```

Parameters:

bBufIndex – index of the buffer to read from.

Return Value:

The data byte read from.

Side Effects:

See Note ** at the beginning of the API section.

I2CSBUF_bCheckStatus

Description:

Returns a value that indicates status of I2C data transaction.

C Prototype:

```
BYTE  I2CSBUF_bCheckStatus(void)
```

Assembler:

```
lcall I2CSBUF_bCheckStatus
```

Parameters:

None

Return Value:

The status of I2C data transaction: four bits in one byte return value indicates the below information as using I2C_XSTAT register. Symbolic names used for the bits are mentioned below.

Symbolic Name	Value	Description
I2CSBUF_BUS_BUSY	0x80	This bit is set only if there is ongoing I2C traffic on bus.
I2CSBUF_AUTO_NACK_ON	0x04	This bit is set only if auto NACK feature is activated by I2C block.
I2CSBUF_LAST_TX_RD	0x20	This bit is set if last I2C transaction by host is a read operation.
I2CSBUF_LAST_TX_WR	0x40	This bit is set if last I2C transaction by host is a write operation.

Note: If the I2CSBUF_AUTO_NACK_ON bit is set then all other status bits are cleared automatically.

Side Effects:

See Note ** at the beginning of the API section.

I2CSBUF_ClearWriteStatus

Description:

Clears the write status bit in the I2C block. If the I2CSBUF_AUTO_NACK_ON bit is set (I2CSBUF_SetAutoNACK() was recently called) then the Write Status is cleared automatically.

C Prototype:

```
void  I2CSBUF_ClearWriteStatus(void)
```

Assembler:

```
lcall I2CSBUF_ClearWriteStatus
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

I2CSBUF_ClearReadStatus

Description:

Clears the read status bit in the I2C block. If the I2CSBUF_AUTO_NACK_ON bit is set (I2CSBUF_SetAutoNACK() was recently called) then the Read Status is cleared automatically.

C Prototype:

```
void  I2CSBUF_ClearReadStatus(void)
```

Assembler:

```
lcall I2CSBUF_ClearReadStatus
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

Sample Firmware Source Code

Here is an example of an implementation of the I2CSBUF Slave User Module:

```
//*************************************************************
// Example code to demonstrate the use of the I2CSBUF UM
//
// This code sets up the I2CSBUF slave to expose 32 byte RAM buffer.
//
// * The instance name of the I2CSBUF User Module is "I2CSBUF".
//
//*************************************************************
#include <m8c.h>     // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules

BYTE baI2CSBUFReadBuffer[8], baI2CSBUFWriteBuffer[] = {1,2,3,4,5,6,7,8};

void main(void)
{
    I2CSBUF_Start();
    I2CSBUF_DisableInt(I2CSBUF_INT_ADDR_STOP_BC);

    while(1)
    {
        // Check if last transaction was written from host
        if ((I2CSBUF_bCheckStatus() & I2CSBUF_LAST_TX_WR) && !(I2CSBUF_bCheckStatus()
        & I2CSBUF_BUS_BUSY))
        {
            // Engage the auto NACK from block to prevent the host access to buffer
            I2CSBUF_SetAutoNACK();

            // Check if the auto NACK is active, before accessing the buffer
            if (I2CSBUF_bCheckStatus() & I2CSBUF_AUTO_NACK_ON)
            {
                // Read out data from hardware buffer to RAM
                I2CSBUF_bReadBuffer(0, baI2CSBUFReadBuffer, 8);

                // Add user code here that acts upon data written by master,
                //for example:
                baI2CSBUFWriteBuffer[0] = baI2CSBUFReadBuffer[0] + 1;
                baI2CSBUFWriteBuffer[1] = baI2CSBUFReadBuffer[1] + 1;
                baI2CSBUFWriteBuffer[2] = baI2CSBUFReadBuffer[2] + 1;

                // Write data back to buffer to be read by master
                I2CSBUF_WriteBuffer(0, baI2CSBUFWriteBuffer, 8);
```

```
            }

            // Clear auto NACK to allow the host access to buffer
            I2CSBUF_ClearAutoNACK();
        }
    }
}
```

Here is an example of an implementation to enable wake up sleep using I2C address match event.

```
//************************************************************
// Example code to demonstrate the use of the I2CSBUF UM with sleep mode
//
// This code sets up the I2CSBUF slave to expose 32 byte RAM buffer
// and demonstrate waking device up from I2C address match event
//
// * The instance name of the I2CSBUF User Module is "I2CSBUF"
//************************************************************

#include <m8c.h> // part specific constants and macros
#include "PSoCAPI.h" // PSoC API definitions for all user modules

void main (void)
{
    I2CSBUF_Start(); // Start I2C block
    I2CSBUF_EnableInt(0); //I2CSBUF_INT_ADDR_STOP_BC);
    // Enable I2C block interrupt for wake up from sleep

    M8C_EnableGInt; // Enable global interrupt

    SLP_CFG2 |= SLP_CFG2_I2C_ON; // Set bit to power i2C block during sleep.

    I2CSBUF_WriteBufferByte(0, 0); //Clear buffer[0]
    I2CSBUF_WriteBufferByte(1, 0); //Clear buffer[1]

    while(1) // infinite loop
    {
        // Check if last transaction was written from host
        if ((I2CSBUF_bCheckStatus() & I2CSBUF_LAST_TX_WR) && !(I2CSBUF_bCheckStatus()
        & I2CSBUF_BUS_BUSY))
        {
            if(I2CSBUF_bReadBufferByte(0)) //Check buffer[0]
            {
                I2CSBUF_WriteBufferByte(0, 0); //Clear buffer[0];

                I2CSBUF_SetAutoNACK(); // Set auto NACK and check if auto NACK is
                //engaged

                if (I2CSBUF_bCheckStatus() & I2CSBUF_AUTO_NACK_ON) M8C_Sleep;
                // Check if auto NACK is engaged

                I2CSBUF_WriteBufferByte(1, (I2CSBUF_bReadBufferByte(1) + 1));
                //Increment buffer[1]
            }
        }
```

```
    }
}
```

Here is an example of an implementation of the I2CSBUF Slave User Module written in assembly code:

```
;-------------------------------------------------------------------------
; Example code to demonstrate the use of the I2CSBUF UM
;
; This code sets up the I2CSBUF slave to expose 32 byte RAM buffer.
;
; * The instance name of the I2CSBUF User Module is "I2CSBUF".
;-------------------------------------------------------------------------

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

export _main

AREA I2CSBUF_RAM (RAM, REL, CON)

baI2CSBUFReadBuffer: blk 8
baI2CSBUFWriteBuffer: blk 8

AREA text (ROM, REL, CON)

_main:

    ; M8C_EnableGInt ; Uncomment this line to enable Global Interrupts
    lcall I2CSBUF_Start
    mov   A, I2CSBUF_INT_ADDR_STOP_BC
    lcall I2CSBUF_DisableInt

.mainloop:
    lcall I2CSBUF_bCheckStatus
    and   A, I2CSBUF_LAST_TX_WR
    jz    .mainloop
    lcall I2CSBUF_bCheckStatus
    and   A, I2CSBUF_BUS_BUSY
    jnz   .mainloop
    lcall I2CSBUF_SetAutoNACK
    lcall I2CSBUF_bCheckStatus
    and   A, I2CSBUF_AUTO_NACK_ON
    jz    .auto_nack_off

    mov   A, 8
    push  A
    mov   A, >baI2CSBUFReadBuffer
    push  A
    mov   A, <baI2CSBUFReadBuffer
    push  A
    mov   A, 0
```

```
    push  A
    lcall I2CSBUF_bReadBuffer
    add   SP, -4

    RAM_SETPAGE_CUR >baI2CSBUFReadBuffer

    mov   A, [<baI2CSBUFReadBuffer]
    inc   A
    mov   [<baI2CSBUFWriteBuffer], A
    mov   A, [<baI2CSBUFReadBuffer+1]
    inc   A
    mov   [<baI2CSBUFWriteBuffer+1], A
    mov   A, [<baI2CSBUFReadBuffer+2]
    inc   A
    mov   [<baI2CSBUFWriteBuffer+2], A

    mov   A, 8
    push  A
    mov   A, >baI2CSBUFWriteBuffer
    push  A
    mov   A, <baI2CSBUFWriteBuffer
    push  A
    mov   A, 0
    push  A
    lcall I2CSBUF_WriteBuffer
    add   SP, -4

.auto_nack_off:
    lcall I2CSBUF_ClearAutoNACK
    jmp   .mainloop
.terminate:
    jmp .terminate
```

Configuration Registers

This section describes the PSoC Resource Registers used or modified by the I2CSBUF User Module.

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

Bit	7	6	5	4	3	2	1	0
Value	Reserved	Pin Select	Reserved	Stop IE	Clock Rate[1:0]		Reserved	Enable

Pin Select: When cleared, SDA is on P1[5] and SCL is on P1[7]. When set, SDA is on P1[0] and SCL is on P1[1].

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, and 400 kbps.

00b = 100 kHz

01b = 400 kHz

10b = 50k

11b = reserved

Enable: Enable the I^2C HW block.

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

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

Bus Error: Indicates a bus error has been detected.

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.

Transmit: This bit sets the direction of the shifter for a subsequent byte transfer. The shifter is always shifting in data from the I^2C bus, but a write of '1' enables the output of the shifter to drive the SDA output line. Since a write to this register initiates the next transfer, data must be written to the data register before writing this bit. In receive mode (0), the previously received data must have been read from the data register before this write. Firmware derives this direction from the RW bit in the received slave address. This direction control is valid only for data transfers. The direction of address bytes is determined by the hardware.

Last Received Bit (LRB): 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 and waiting for the next action.

Table 3. Resource I2C_ADDR: Bank 0 reg[CA] I2C Address Register

Bit	7	6	5	4	3	2	1	0
Value	Reserved	Slave Address						

These seven bits hold the slave's own device address.

Table 4. Resource I2C_BP: Bank 0 reg[CB] I2C Base Address Pointer Register

Bit	7	6	5	4	3	2	1	0
Value	EZ_RD_IE	EZ_WR_IE	CLK_STRETCH_EN	I^2C Base Pointer				

The I^2C Base Address Pointer Register contains the base address value of the RAM data buffer and additional control bits.

EZ_RD_IE: Interrupt enable for EZ_RD_STATUS.

EZ_WR_IE: Interrupt enable for EZ_WR_STATUS.

CLK_STRETCH_EN: This bit configures the clock stretch mode during sleep-to-wakeup transition.

Table 5. Resource I2C_CP: Bank 0 reg[CC] I2C Current Address Pointer Register

Bit	7	6	5	4	3	2	1	0
Value	Reserved	Reserved	Reserved	I^2C Current Pointer				

This register gets set at the same time, and with the same value, as the I2C_BP register. After each completed data byte of the current I2C transaction, the value of this register is incremented by one. The value always determines the location from which read or write data comes or is written to.

Table 6. Resource CPU_BP: Bank 0 reg[CD] CPU Base Address Pointer Register

Bit	7	6	5	4	3	2	1	0
Value	Reserved	Reserved	Reserved	CPU Base Pointer				

This register value is completely controlled by IO writes by the CPU. When this register is written, the current address pointer, CPU_CP, is also updated with the same value. The first read or write from or to the I2C_ BUF register starts at this address. Firmware ensures that the slave device always has valid data or that the data is read before it is overwritten.

Table 7. Resource CPU_CP: Bank 0 reg[CE] CPU Current Address Pointer Register

Bit	7	6	5	4	3	2	1	0
Value	Reserved	Reserved	Reserved	CPU Current Pointer				

This register is set at the same time, and with the same value as the CPU_BP register. Whenever I2C_BUF register is written to or read, the CPU_CP increments automatically.

Table 8. Resource I2C_BUF: Bank 0 reg[CF] I2C Data Buffer Register

Bit	7	6	5	4	3	2	1	0
Value	Data Buffer							

The I^2C Data Buffer Register (I2C_BUF) is the CPU read and write interface to the data buffer. Whenever this register is read, the data at the location pointed to by CPU current pointer (CPU_CP) is returned. Similarly, whenever this register is written, the data is transferred to the buffer and written at the location indicated by the CPU current pointer (CPU_CP). Whenever this register is read, without initializing the RAM contents either through the I^2C or CPU interface, no valid value is returned.

Table 9. Resource I2C_XCFG: Bank 0 I2C Extended Control Register

Bit	7	6	5	4	3	2	1	0
Value	CSR_CLK_EN	Reserved	FORCE_NACK_MODE	FORCE_NACK	No BC Int	Reserved	Buffer Mode	HW Addr EN

The I^2C Extended Control Register (I2C_XCFG) configures enhanced features. When all bits are left in the default reset state of '0', the block operates in compatibility mode. Bits 0 through 7 (except bit 2) are RW.

Table 10. Resource I2C_XSTAT: Bank 0 I2C Extended Status Register

Bit	7	6	5	4	3	2	1	0
Value	CSR_CLK_EN	Reserved	FORCE_NACK_MODE	FORCE_NACK	No BC Int	Reserved	Buffer Mode	HW Addr EN

The I^2Extended Status Register (I2C_XSTAT) reads enhanced feature status. All bits are read only.

Version History

Version	Originator	Description
1.00	DHA	Initial version.
1.00.b	DHA	Updated RAM and ROM usage in the user module datasheet.

Note PSoC Designer 5.1 introduces a version history in all user module datasheets to document high level descriptions of the differences between the current and previous user module versions.

Copyright © 2012-2013 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.