I²C Hardware Block Datasheet I2CHW V 2.20

Copyright © 2008-2013 Cypress Semiconductor Corporation. All Rights Reserved.

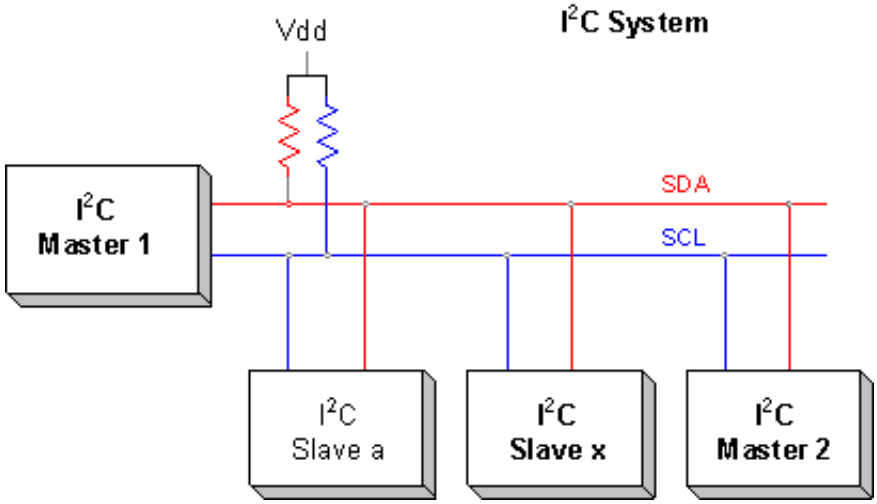
Resources	PSoC® Blocks				API Memory (Bytes)		Pins per senor
	CapSense®	I2C/SPI	Timer	Comparator	Flash	RAM	
CY8C20x66, CY8C20x36, CY8C20336AN, CY8C20436AN, CY8C20636AN, CY8C20xx6AS, CY8C20XX6L, CY8C20x46, CY8C20x96, CY8C20045, CY7C60413, CY7C645xx, CY7C643/4/5xx, CY7C60424, CY7C6053x, CYONS2010, CYONS2011, CYONSFN2051, CYONSFN2053, CYONSFN2061, CYONSFN2151, CYONSFN2161, CYONSFN2162, CYONSFN2010-BFXC, CYONSCN2024-BFXC, CYONSCN2028-BFXC, CYONSCN2020-BFXC, CYONSKN2033-BFXC, CYONSKN2035-BFXC, CYONSKN2030-BFXC, CYONSTN2040, CY8CTST200, CY8CTMA140, CY8CTMG2xx, CY8CTMG30xx, CY8C20xx7/7S, CYRF89x35, CY8C20065, CY8C24x93, CY7C69xxx							
Slave Only		x			279–440	8	2

Features and Overview

- Industry standard Philips I²C bus compatible interface
- Slave only operation
- Only two pins (SDA and SCL) required to interface to I²C bus
- Standard data rate of 100/400 kbits/s, also supports 50 kbits/s
- High level API requires minimal user programming
- 7-bit addressing mode

The I²C Hardware User Module implements an I²C Slave device in firmware. 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 I2CHW User Module supports the standard mode with speeds up to 400 kbits/s and is compatible with other slave devices on the same bus.

Figure 1. I²C Block Diagram



Functional Description

This user module provides support for an I²C hardware resource. It is capable of transferring data at 50/100/400 kbits/s when the CPU clock is configured to run at 12 MHz. It is possible to use slower CPU clocks, but doing so may result in the bus stalling more or less during address or data processing. The I²C specification allows the master to run at clock speeds from 100 kHz down to DC. There are two different selections for serial data (SDA) and serial clock (SCL) providing direct access to the hardware resource. The seven-bit address mode is supported in the supplied APIs.

Detailed descriptions of the I²C bus and the implementation of the resource are available in the device datasheet and on the Internet.

I²C Slave

The I²C resource supports data transfer at a byte-by-byte level. At the end of each address or data transmission/reception, status is reported and a dedicated interrupt may be triggered. Status reporting and interrupt generation depends upon data transfer direction and the condition of the I²C bus as detected by the hardware. Interrupts may be configured to occur on byte-complete and bus-error detection.

Every I²C transaction consists of a Start, Address, R/W direction, data, and a termination. The I²C resource used for this user module is capable of operating only as an I²C Slave. This user module provides an interrupt based buffered transfer mechanism. Communication is initiated from a foreground function call. At the completion of each byte of the message, an interrupt is triggered and the I²C bus is stalled, the interrupt service routine (ISR) provided takes appropriate action on the bus allowing communication to continue, depending upon the initialization performed. A slave device that does not acknowledge an address is not interrupted again until the next address is received. Slave devices must respond to each address either by acknowledging (ACK) or not acknowledging (NAK).

Ignoring differences between a Master and Slave, two general cases exist, that of a receiver and that of a transmitter. For an I²C receiver, an interrupt occurs after the 8th bit of incoming data. At this point, a receiving device must decide to ACK or 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 I²C resource of the ACK or NAK status. The write to the I2C_SCR register paces data flow on the bus by unstalling the

bus, placing the ACK or 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 provides 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 written again to trigger the next portion of the transmission.

When using the buffered read and write routines (bWriteBytes(), bWriteCBytes(), fReadBytes()), it is unnecessary to use any of the buffer initialization functions (InitWrite(), InitRamRead(), InitFlashRead()). These functions are called as part of the function call that initiates the buffered read or write (bWriteBytes(), bWriteCBytes(), fReadBytes()).

Design Considerations

Slave devices maintain an internal count of the buffer space remaining for access by a Master. The variables are named I2CHW_Read_Count and I2CHW_Write_Count (where I2CHW is replaced by the instance name of the user module in PSoC Designer). The variables are global and are accessed from C by including an appropriate 'extern' declaration. Determine the number of bytes read or written by a master by subtracting the current value of the count variable from the initial value of the count variable. The initial size of the count variable is set by using the functions I2CHW_InitWrite, I2CHW_InitRamRead, I2CHW_InitFlashRead in the case of the Slave (only) user module.

Buffers are used to read and write data within the I^2C slave APIs. You must initialize appropriate buffers before enabling the I^2C slave. After initiating a read or a write by the I^2C master, appropriate status bits are set in the I2CHW_Status byte. The foreground process in the slave operates on data deposited in a write buffer or extracted from a read buffer. The slave data transfer interrupt service routine (ISR) does not allow the access of buffers beyond their defined length when the ISR is entered. Reading and writing to buffers is handled in the following manner:

- If the I^2C master attempts to read more data than is contained in a buffer, the last byte is retransmitted until the I^2C master stops reading. (The I^2C protocol does not define a method for the I^2C slave to stop a master from reading)
- When an I^2C master receives and writes data to the I^2C slave and then determines that there is no more available storage, the slave generates a NAK when it receives the last byte. If the I^2C master continues to write data, the slave continues to NAK it. Once the first NAK is generated (data is stored in the last available location), further data is not stored.
- If a buffer is defined with zero length, data written to the I^2C slave is NAKed and not stored. Enabling the ability to read data directly from flash also allows the use of either RAM or flash buffers. If the data transfer ISR uses a read buffer located in flash/ROM or RAM, the supplied APIs should be used to configure it.

Dynamic Reconfiguration

Cypress recommends against incorporating the I2CHW resource into dynamically loaded/unloaded overlays. Place the I2CHW resource only as part of the base configuration. Modify the operation of the I2CHW block based on operational requirements. However, attempting to remove the resource as part of dynamic reconfiguration may result in adverse affects on external I^2C devices.

I²C Addressing

I²C addresses are contained in the upper 7 bits of the first byte of a read or write transaction. This byte is used by the I²C 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 a 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 I²C 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).

DC and AC Electrical Characteristics

Refer to the device datasheet for your PSoC device for the electrical characteristics of the I²C interface.

As the block diagram illustrates, the I²C bus requires external pull-up 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} = 0.4 V for the output stage. This limits the minimum pull up resistor value for a 5 V system to about 1.5 kΩ. The maximum value for R_P depends upon the bus capacitance and the clock speed. For a 5 V 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 NXP web site at www.nxp.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.

Placement

The I2CHW User Module allows two choices for SCL and SDA. One choice has SCL on P1[7] and SDA on P1[5]. The other choice has SCL on P1[1] and SDA on P1[0]. Use the P1[5]/P1[7] pins if at all possible because P1[0]/P1[1] are shared with ISSP and ECO. There are no placement restrictions. Multiple placements of I²C modules is not possible, because the I²C module uses a dedicated PSoC resource block and interrupt.

Parameters and Resources

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

Slave_Addr

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

I2C_Clock

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

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

I2C_Pin

Selects the pins from Port 1 to use for I^{2}C signals. There is no need to select the proper drive mode for these pins because PSoC Designer does this automatically.

Read_Buffer_Types

Selects what types of buffers are supported for data reads. Two selections are available: RAM ONLY or RAM OR FLASH. Selection of RAM ONLY removes code and variables required to support direct flash-ROM reads. Selection of RAM OR FLASH provides code and variable support for reading either RAM buffers or flash-ROM buffers for data to transmit to the master. If RAM OR FLASH is selected, use the I2CHW_InitRamRead() or I2CHW_InitFlashRead() API to select the buffer type.

Communication_Service_Type

This parameter allows the user to select between an interrupt based data processing strategy or a polled strategy. In the interrupt based strategy, a transfer is initiated against a predefined buffer. Data then moves in or out of the buffer very quickly in the background. An ISR is included that handles data movement. When you select the polled data processing strategy, you are in control of when data movement occurs. To implement a polled strategy you periodically call the function I2CHW_Poll() (see the I2CHWslave.h files for the exact instance name). Each time you call the polling function a single byte is transferred. Other I^{2}C functions are used identically. Use the polled communication strategy in a situation where interrupt latency is critically important (and asynchronous communication interrupts may cause problems). Another use is when you need absolute control of when data is transferred. A drawback of polling is that when the I^{2}C state machine is enabled, the bus automatically stalls after each byte until the polling function is called.

Application Programming Interface

The Application Programming Interface (API) firmware provides high level commands that support sending and receiving multibyte transfers. Create read buffers in RAM or flash memory. Write buffers are only setup in RAM memory.

Note

In this, as in all user module APIs, the values of the A and X registers are 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 is used for efficiency and is in force since PSoC Designer version 1.0. The C compiler automatically handles this requirement. Assembly language programmers must ensure their code observes the policy as well. 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.

I2CHW_Start

Description:

Does Nothing. Provided for interface consistency only. Use I2CHW_EnableSlave() to start the slave and I2CHW_EnableInt() to enable interrupts.

C Prototype:

```
void  I2CHW_Start(void);
```

Assembler:

```
lcall  I2CHW_Start
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. This is true for all RAM page pointer registers in the Large Memory Model. When necessary, it is the responsibility of the calling function to preserve the values across calls to fastcall16 functions.

I2CHW_Stop

Description:

Disables the I2CHW by disabling the I^2C interrupt.

C Prototype:

```
void   I2CHW_Stop(void);
```

Assembler:

```
lcall  I2CHW_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

See the note at the beginning of the API section.

I2CHW_EnableInt

Description:

Enables I^2C interrupt allowing start condition detection. Remember to call the global interrupt enable function by using the macro: M8C_EnableGInt.

C Prototype:

```
void   I2CHW_EnableInt(void);
```

Assembler:

```
lcall  I2CHW_EnableInt
```

Parameters:

None

Return Value:

None

Side Effects:

See the note at the beginning of the API section.

I2CHW_DisableInt

Description:

Disables the I^2C slave by disabling the SDA interrupt. Performs the same action as I2CHW_Stop.

C Prototype:

```
void   I2CHW_DisableInt(void);
```

Assembler:

```
lcall I2CHW_DisableInt
```

Parameters:

None

Return Value:

None

Side Effects:

See the note at the beginning of the API section.

I2CHW_EnableSlave

Description:

Starts the I2C Slave function for the I2C HW block by setting the Enable Slave bit in the I2C_CFG register.

C Prototype:

```
void   I2CHW_EnableSlave(void);
```

Assembler:

```
lcall  I2CHW_EnableSlave
```

Parameters:

None

Return Value:

None

Side Effects:

See the note at the beginning of the API section.

I2CHW_DisableSlave

Description:

Disables the I2C Slave function by clearing the Enable Slave bit in the I2C_CFG register.

C Prototype:

```
void   I2CHW_DisableSlave(void);
```

Assembler:

```
lcall  I2CHW_DisableSlave
```

Parameters:

None

Return Value:

None

Side Effects:

See the note at the beginning of the API section.

I2CHW_Poll

Description:

Used when the Communication_ Service_Type parameter is set to "Polled". This function provides a user controlled entry into the I/O processing routine. If Communication_Service_Type parameter is set to "Interrupt", the function does nothing.

Note: Calling I2CHW_Poll releases the I2C bus. This allows a master to possibly read or write data before slave firmware has finished processing the previous request.

C Prototype:

```
void   I2CHW_Poll(void);
```

Assembler:

```
lcall I2CHW_Poll
```

Parameters:

None

Return Value:

None

Side Effects:

One I^2C event is processed each time this routine is called and status variables are updated. An event constitutes either an error condition, an I/O byte, or in certain cases, a stop condition. There are three possible results from calling this routine:

- No action if there is no event.
- Reception or transmission of an address or data byte if one is available.
- Processing of a stop event when an external master completes the write operation.

When a stop state is processed at the end of a write operation, only status variables are updated. If an I^2C byte is pending when a stop state is processed, call the I2CHW_Poll function again to process the byte. The I2CHW_Poll() function has no effect if you set Communication_Service_Type to Interrupt. When a start or restart condition and an address is detected on the bus it stalls until the I2CHW_Poll() function is called. If the address is NAKed, subsequent bytes transferred for that transaction are ignored until another start/restart and address is detected, otherwise the I^2C bus stalls for each data byte until the I2CHW_Poll() function is called.

The I^2C bus is stalled by the I^2C hardware until this function is called if the Communication_Service_Type is set to "Polled". For received data the bus stalls at the end of the byte and before an ACK/NAK is generated by holding the SCL (clock) line low. For transmitted data the bus stalls immediately after the ACK or NAK bit is generated externally.

I2CHW_bReadI2CStatus

Description:

Returns the status bits in the Control/Status register.

C Prototype:

```
BYTE  I2CHW_bReadI2CStatus(void);
```

Assembler:

```
lcall  I2CHW_bReadI2CStatus ; Accumulator contains status
```

Parameters:

None

Return Value:

```
BYTE  I2CHW_RsrcStatus
```

See Table.

Side Effects:

See the note at the beginning of the API section. Currently, only the CUR_PP page pointer register is modified.

Constant	Value	Description
I2CHW_RD_NOERR	01h	Data read by the master, normal ISR exit
I2CHW_RD_ OVERFLOW	02h	More data bytes were read by the master than were available
I2CHW_RD_ COMPLETE	04h	A read was initiated and is complete
I2CHW_READFLASH	08h	The next read is from a flash location
I2CHW_WR_NOERR	10h	Data was written successfully by the master
I2CHW_WR_OVERFLOW	20h	The master wrote too many bytes for the write buffer
I2CHW_WR_COMPLETE	40h	A master write was completed by a new address or stop
I2CHW_ISR_ACTIVE	80h	The I^2C ISR has not yet exited and is active

I2CHW_ClrRdStatus

Description:

Clears the read status bits in the I2CHW_RsrcStatus register. No other bits are affected.

C Prototype:

```
void  I2CHW_ClrRdStatus (void);
```

Assembler:

```
lcall  I2CHW_ClrRdStatus
```

Parameters:

None

Return Value:

None

Side Effects:

See the note at the beginning of the API section.

I2CHW_ClrWrStatus

Description:

Clears the write status bits in the I2CHW_RsrcStatus register. No other bits are affected.

C Prototype:

```
void  I2CHW_ClrWrStatus (void);
```

Assembler:

```
lcall  I2CHW_ClrWrStatus
```

Parameters:

None

Return Value:

None

Side Effects:

See the note at the beginning of the API section. Currently, only the CUR_PP page pointer register is modified.

I2CHW_InitWrite

Description:

Initializes a data buffer pointer for the slave to deposit data and initializes the value of a count byte for the same buffer. Count is initialized to the maximum supplied buffer length. On the next instance of a master write, data is placed at the address defined by this function.

C Prototype:

```
void   I2CHW_InitWrite(BYTE * pWriteBuf, BYTE bBufLen);
```

Assembler:

```
AREA    bss (RAM, REL)
abWriteBuf    blk 10h

AREA    text (ROM,REL)
    push X                      ; save registers
    push A
    add  SP, 3
    mov  X, SP
    dec  X                      ; X points at data SP points at next
      ; empty stack location
    mov  [X], abWriteBuf        ; place the buffer address
      ; (page 0)on the stack at [X]
    mov  [X-2], 10              ; place the count at [x-2]
                                ; don't care what [X-1] is
                                ; the compiler would assign 0 as the
                                ; MSB of the Ramtbl addr
    lcall  I2CHW_InitWrite
    add   SP, -3                ; restore the stack
    pop   A                     ; restore registers
    pop   X
```

Parameters:

pWriteBuf: Pointer to a RAM buffer location.

buf_len: Length of write buffer.

Return Value:

None

Side Effects:

See the note at the beginning of the API section. Currently, only the CUR_PP page pointer register is modified.

I2CHW_InitRamRead

Description:

Initializes a RAM data buffer pointer from which the Master retrieves data, and initializes the value of a count byte for the same buffer. Clears the I2CHW_SlaveStatus flag I2CHW_READFLASH to 0 (Read status bits only), causing the next read to be attempted from a previously set buffer location in RAM.

C Prototype:

```
void   I2CHW_InitRamRead(BYTE * pReadBuf, BYTE bBufLen);
```

Assembler:

```
AREA bss (RAM,REL)
abReadBuf:    blk 10h

AREA text (ROM,REL)
    push  X                        ; save registers
    push  A
add   SP, 3
    mov   X, SP
    dec   X                        ; X points at data SP points at next
                                   ; empty stack location
    mov   [X], abReadBuf           ; place the read buffer address
       ; (page0)on the stack at [X]
    mov   [X-2], 10                ; place the count at [x-2]
                                   ; don't care what [X-1] is
                                   ; the compiler would assign 0 as
                                   ; the MSB of the Ramtbl addr
    lcall  I2CHW_InitRamRead
    add   SP, -3                   ; back up the stack (subtract 3)
    pop   A                        ; restore registers
    pop   X
```

Parameters:

_ReadBuf: Pointer to a RAM buffer location. bBufLen: Length of read buffer.

Return Value:

None

Side Effects:

See the note at the beginning of the API section. Currently, only the CUR_PP page pointer register is modified.

I2CHW_InitFlashRead

Description:

Initializes a flash data buffer pointer for retrieval of data. Sets the I2CHW_SlaveStatus flag I2CHW_READFLASH to 1, causing the next read to be attempted from a previously set buffer location in flash.

C Prototype:

```
void   I2CHW_InitFlashRead(const BYTE * pFlashBuf, WORD wBufLen);
```

Assembler:

```
area table(ROM,ABS)
org 0x1015

abFlashBuf:

      db 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88
      db 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff


area text(ROM,REL)


    push  X                          ; save registers
    push  A
    add   SP, 4
    mov   X, SP
    dec   X                          ; X points at data SP points at next
         ; empty stack location
    mov   [X], <abFlashBuf           ; place the LSB of rom
         ; address on the stack at [X]
    mov   [X-1], >abFlashBuf         ; place the MSB of the rom address
         ; at [x-1] variable
    mov   [X-2], 0x0F                ; place the LSB of length
                                     ; at [x-2]
    mov   [X-3], 0x00                ; place the MSB of length
         ; at [x-3]
    lcall  I2CHW_InitFlashRead
    add   SP, -4                     ; adjust the stack (subtract 4)
    pop   A                          ; restore registers
    pop   X
```

Parameters:

pFlashBuf: Pointer to a flash/ROM buffer location. wBufLen: Length of buffer.

Return Value:

None

Side Effects:

See the note at the beginning of the API section. Currently, only the CUR_PP page pointer register is modified.

I2CHW_EnableHWAddrCheck

Description:

This API is only for the CY8C20xx7/7S device family. It enables the Hardware Address Comparison feature.

C Prototype:

```
void   I2CHW_EnableHWAddrCheck (void);
```

Assembly:

```
lcall I2CHW_EnableHWAddrCheck)
```

Parameters:

None.

Return Value:

None

Side Effects:

Make sure that if the alternative slave address is enabled (flash registers), the device does not respond to it while the Hardware Address Comparison feature is active. This is because the I2C slave responds only to its primary address.

However, the Hardware Address Comparison feature is necessary for AutoNACKing during sleep operation.

Examples of Usage:

```
I2CHW_SetAutoNACK();
while (! (I2CHW_AUTO_NACK_ON & I2CHW_bCheckStatus()));
I2CHW_EnableHWAddrCheck();
SLP_CFG2 |= 0x02; //I2C_ON during sleep
M8C_Sleep;
I2CHW_DisableHWAddrCheck();
```

I2CHW_DisableHWAddrCheck

Description:

This API is only for the CY8C20xx7/7S device family. It disables the Hardware Address Comparison feature.

C Prototype:

```
void I2CHW_DisableHWAddrCheck (void);
```

Assembly:

```
lcall I2CHW_DisableHWAddrCheck
```

Parameters:

None.

Return Value:

None

Side Effects:

Make sure that if the alternative slave address is enabled (flash registers), the device does not respond to it while the Hardware Address Comparison feature is active. This is because the I2C slave responds only for its primary address.

I2CHW_SetAutoNACK

Description:

This API is only for the CY8C20xx7/7S device family. It disables the Hardware Address Comparison feature.

C Prototype:

```
void I2CHW_SetAutoNACK (void);
```

Assembly:

```
lcall I2CHW_SetAutoNACK
```

Parameters:

None.

Return Value:

None

I2CHW_ClearAutoNACK

Description:

This API is only for the CY8C20xx7/7S device family. It disables the Hardware Address Comparison feature.

C Prototype:

```
void I2CHW_ClearAutoNACK (void);
```

Assembly:

```
lcall I2CHW_ClearAutoNACK
```

Parameters:

None.

Return Value:

None

I2CHW_bCheckStatus

Description:

This API is only for the CY8C20xx7/7S device family. It returns the status of I2C data transaction.

C Prototype:

```
BYTE  I2CHW_bCheckStatus (void);
```

Assembly:

```
lcall I2CHW_bCheckStatus
and   A, I2CHW_BUS_BUSY
```

```
jnz   .BusBusy
```

Parameters:

None.

Return Value:

A contains the following flags:

Value Name Definition	Value	Meaning
I2CHW_AUTO_NACK_ON	0X04	Auto NACK feature is activated
I2CHW_LAST_TX_RD	0X20	Last I2C transaction was a read action
I2CHW_LAST_TX_WR	0X40	Last I2C transaction was a write action
I2CHW_BUS_BUSY	0X80	There is I2C traffic on the bus

Sample Firmware Source Code

Here is an example of an implementation of the I2CHW Slave User Module:

```
/*************************************************************/
/* This sample code echoes bytes received from a master */
/* The master sends and requests up to 64 bytes.         */
/*                                                         */
/* The instance name of the I2CHWs User Module is defined */
/* as I2CHW.                                              */
/*                                                         */
/*  NOTE I2CHW does not depend upon the CPU clock         */
/*************************************************************/
#include <m8c.h>
#include <i2CHWCommon.h>

/* setup a 64 byte buffer */
BYTE    abBuffer[64];
BYTE    status;

void EchoData(void)
{
/* Start the slave and wait for the master */
    I2CHW_Start();
    I2CHW_EnableSlave();
/* Enable the global and local interrupts */
    M8C_EnableGInt;
    I2CHW_EnableInt();

    /* Setup the Read and Write Buffer - set to the same buffer */
    I2CHW_InitRamRead(abBuffer,64);
    I2CHW_InitWrite(abBuffer,64);


    /* Echo forever */
```

```
    while(1)
    {
        status = I2CHW_bReadI2CStatus();
        /* Wait to read data from the master */
        if( status & I2CHW_WR_COMPLETE )
        {
        /* Data received - clear the Write status */
        I2CHW_ClrWrStatus();
        /* Reset the pointer for the next read data */
        I2CHW_InitWrite(abBuffer,64);
        }
        /* wait until data is echoed */
        /* want to know if RD_NOERR is SET AND RD_COMPLETE is SET */
        if( status & I2CHW_RD_COMPLETE )
        {
        /* Data echoed - clear the read status */
        I2CHW_ClrRdStatus();
        /* Reset the pointer for the next data to echo */
        I2CHW_InitRamRead(abBuffer,64);
        }
    }
}
void main(void)
{
    EchoData();
}
```

Here is an implementation of the I2CHW User Module configured as a slave written in assembly code:

```
;-----------------------------------------------------------
; this assembly assumes small memory model
;-----------------------------------------------------------
include "m8c.inc"
include "I2CHWCommon.inc"

BUFFERSIZE: equ 64

export abBuffer

area data(REL, CON, RAM)

abBuffer: blk BUFFERSIZE

area text(REL, CON, ROM, CODE)

export _main

_main:
   lcall I2CHW_Start
   lcall I2CHW_EnableSlave
   call I2CHW_EnableInt
   M8C_EnableGInt

.Init:
```

```
   mov A, BUFFERSIZE
   push A
   mov A, >abBuffer
   push A
   mov A, <abBuffer
   push A
   lcall I2CHW_InitWrite
   lcall I2CHW_InitRamRead
   add   SP, -3

.CheckI2CStatus:
   lcall I2CHW_bReadI2CStatus
   push A
   and A, I2CHW_WR_COMPLETE
   jnz .WriteHappened
   pop A
   and A, I2CHW_RD_COMPLETE
   jnz .ReadHappened
   jmp .CheckI2CStatus

.WriteHappened:
   lcall I2CHW_ClrWrStatus
   mov   A, BUFFERSIZE
   push  A
   mov   A, >abBuffer
   push  A
   mov   A, <abBuffer
   push  A
   lcall I2CHW_InitWrite
   add   SP, -4
   jmp   .CheckI2CStatus

.ReadHappened:
   lcall I2CHW_ClrRdStatus
   mov A, BUFFERSIZE
   push A
   mov A, >abBuffer
   push  A
   mov   A, <abBuffer
   push  A
   lcall I2CHW_InitRamRead
   add   SP, -3
   jmp   .CheckI2CStatus

;end _main
```

Configuration Registers

This section describes the PSoC Resource Registers used or modified by the I2CHW User Module.

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

Bit	7	6	5	4	3	2	1	0
Value	Reserved	PinSelect	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 3 valid Clock rates 50- 100- 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 the detection of an 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.

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 only valid 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_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. The received data is read from this register. It may contain an address or data.

Table 4. 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 5. Resource I2C_BP: Bank 0 reg[CB] I2C Base Address Pointer Register

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

The value of this register is modified only at the beginning of every I^2C write transaction. The I^2C master must always supply a value for this register in the first byte of data after the slave's address in a given write transaction. When performing reads the master does not have to set the value of this register. The current value of this register is also used directly for reads.

Table 6. 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, the I2C_BP register gets set to. After each completed data byte of the current I^2C transaction the value of this register is incremented by one. The value of this register always determines the location that read or write data comes from or is written to.

Table 7. 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 I/O 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 start at this address. Firmware ensures that the slave device always has valid data or the data is read before overwritten.

Table 8. 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 or read, the CPU_CP increments automatically.

Table 9. 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. When this register is read, the data at location pointed by CPU current pointer (CPU_CP) is returned. Similarly, when this register is written, the data is transferred to the buffer and written at the location pointed by CPU current pointer (CPU_CP). When this register is read, without initializing the RAM contents either through the I^2C or CPU interface, no valid value is returned.

Version History

Version	Originator	Description
1.1	DHA	Removed the default pin value of the "I2C Pin" parameter, because it corrupts the drive mode of corresponding pins initially set by other user modules. The following changes were done to the Start function: 1. Changed Initial Open-Drain Low drive mode of UM pins to HI-Z analog. 2. Enabled the I^2C block. 3. Gave delay 5 nop instructions. 4. Restored the Initial I^2C pin drive mode. Added ability to update Output Pin through shadow register.
2.00	DHA	Disabled I2CHWSlave Enhanced configuration for new projects.
2.10	DHA	1. Added support for CY8C20xx7/7S device. 2. Added new _SetAutoNACK(), _ClearAutoNACK() and _bCheckStatus () API functions for the CY8C20xx7/7S device
2.20	HPHA	Added I2CHW_EnableHWAddrCheck and I2CHW_DisableHWAddrCheck functions for CY8C20007 family.

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 © 2008-2013 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.