AN50987

Getting Started with I²C in PSoC® 1

Author: Todd Dust and M. Ganesh Raja
Associated Project: Yes
Associated Part Family: CY8C21x23, 21x34, 21x45, 22x45,23x33, 24x23A, 24x33, 24x94, 27x43, 28xxx, 29x66
Software Version: PSoC® Designer™5.4

AN50987 gives an overview of the I²C standard and explains how a PSoC® 1 device handles I²C communications. After reading this application note, you should have an understanding of how I²C works, how to implement it in PSoC 1, and how to choose the correct user module for a design. Example projects demonstrate how to configure PSoC 1 as an I²C master/slave to communicate with other I²C devices on the bus.

Contents

Introduction 2
I²C Basics 2
The Physical Layer 2
The Protocol Layer 2
I²C in PSoC 1 4
Hardware 5
I²C User Modules 6
Firmware 8
Slave Operation 8
Master Operation 9
Multimaster Slave Operation 10
I²Cm 11
Special I²C Considerations 12
I²C Addressing 12
Pull-up Resistors 12
I²C and ISSP Programming Conflicts 12
Pin Glitches at Power-up 13
Clock Speeds 13
Clock Stretching and Interrupt Latency 14
Hot Swapping 14
Glitch Filtering 14
I²C and Sleep 14
I²C and Dynamic Reconfiguration 15
Dynamic Slave Addressing in I²CHW UM 15
The SCL Line Gets Stuck LOW 15
Summary 16
Appendix A 17
Hardware Registers 17
Firmware Requirements 17
Arbitration 18
Appendix B 20
EzI2Cs_ADC_LED_DAC Example Project 20
I²CHW Slave Example Project 24
I²CHW Master Example Project 25
Migrating Example Project 26
Worldwide Sales and Design Support 28

Introduction

Inter-integrated circuit (IIC or I²C) is a common chip-to-chip serial communications standard developed by the Philips semiconductor division (now NXP). I²C provides a simple way for ICs to communicate on the same printed circuit board (PCB). I²C consists of a simple physical layer that requires only two pins and minimal external components. One advantage of I²C over a communication standard such as SPI is that it has a built-in communication protocol that allows easy, error-free communication between devices.

The Cypress PSoC 1 device offers several choices for implementing I²C in a design. These choices come in the form of user modules (UMs) that are found in the PSoC Designer™ integrated development environment (IDE). This application note first describes the basics of I²C to help you understand how the PSoC 1 device handles I²C. If you already have an understanding of I²C basics, skip to I2C in PSoC 1. If you are looking for help troubleshooting your I²C design, skip to the Special I2C Considerations section.

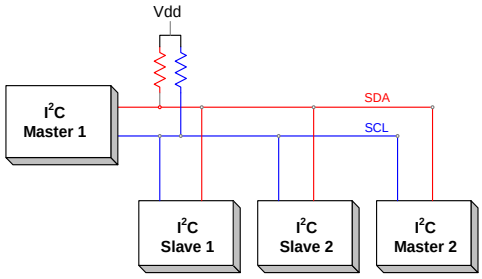
This application note assumes that you are familiar with the PSoC 1 device and PSoC Designer IDE.

I²C Basics

The Physical Layer

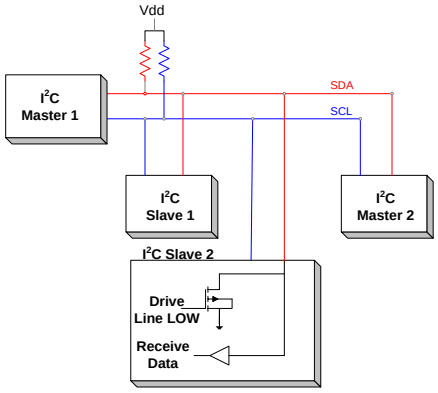
Figure 1 shows how I²C devices connect to the I²C bus. The I²C bus consists of two physical lines: serial data (SDA) and serial clock (SCL). All devices on the bus must connect to these two physical lines. The only external hardware required are pull-up resistors to V_{DD} (high rail) on SDA and SCL. For more information, see the Pull-up Resistors section.

Figure 1. Typical I²C Bus



SDA and SCL are bidirectional; the master can communicate data to the slave and vice versa. SDA and SCL have an open-drain drive mode and drive LOW. A device can drive the line to a logic LOW, or it can be high-impedance. Therefore, no device can drive the lines HIGH. This configuration prevents power-to-ground shorts on the bus; see Figure 2. The pull-up resistors on SDA and SCL produce a HIGH logic level.

Figure 2. Open Drain Drives LOW Pin Configuration



Devices on the I²C bus have a master-slave relationship. The master initiates all data transfers on the bus and generates all clock signals. Each slave has a unique address; the master must first address a specific slave and receive an acknowledgment before the data transfer can begin. Multiple masters and multiple slaves can exist together on the same bus. The I²C bus operates at a variety of frequencies; typical frequencies are 100 kHz and 400 kHz. The I²C specification allows higher frequencies of 1 MHz and 3.4 MHz.

The I²C bus operates at different voltage levels depending on the individual devices themselves. To determine if two devices can successfully communicate, review their respective datasheets to ensure that the logic levels are compatible.

The I²C specification offers a solution for bridging between buses of different voltage levels, in the event that the voltage and logic levels of two devices are not compatible.

The Protocol Layer

Understanding the protocol layer of I²C is the next major step in mastering this digital communication technique.

Each I²C transaction consists of the following elements: start (or repeated start), address, data, and stop.

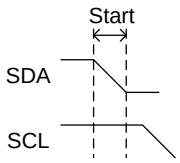
Start or Repeated Start

The master device controls the clock line and therefore initiates all communication on the bus. To gain control of the bus and initiate a transaction, the master first sends a start condition; see Figure 3.

A start condition signals to every device on the bus that a master has taken control and is ready to send an address. The bus is considered busy, so other masters must wait for the bus to be freed by a stop condition to initiate a transfer.

Start = HIGH to LOW transition on SDA when SCL is HIGH

Figure 3. Start Condition



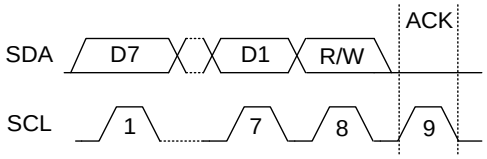
A repeated start condition is physically identical to a start condition. It gives a signal that the master has maintained control of the bus and that the bus is not free.

Address

The address is the first data byte sent by the master after the start condition. Each slave device has a unique address. Most I²C addresses are seven bits long. A read/write (R/W) bit completes the 8-bit byte and indicates the direction of communication for the rest of the transaction; see Figure 4. A read signifies that the master wants to read data from the slave. A write indicates that the master wants to write data to the slave. All address and data bytes are sent in the order of most significant bit (MSB).

For example, if the 7-bit address is 0x20, the full 8-bit byte for a read will be 0x40; the full 8-bit byte for a write will be 0x41.

Figure 4. 7-Bit Address



ACK/NAK

After the master transmits the address, it waits for an acknowledgment (ACK) from the slave. The ACK is an additional status bit at the end of each data byte; thus, each I²C transaction is nine bits long.

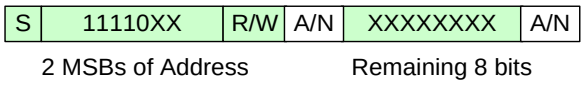
Each slave on the bus is responsible for reading the incoming address and comparing it with its own internal address. If the address matches, the slave must send out an ACK on the ninth clock cycle. If the address does not match, the slave does not acknowledge (NAK).

- ACK = 0 (LOW logic level)
- NAK = 1 (HIGH logic level)

10-Bit Address

The I²C specification also allows for 10-bit addressing; see Figure 5. To accomplish this, the master must send two address bytes to the slave. The first byte, which the slave must acknowledge, contains the sequence 11110 followed by the two MSBs of the address followed by the R/W bit. Next, the master sends the remaining eight bits of the address. Finally, the slave sends an ACK/NAK.

Figure 5. 10-Bit Address

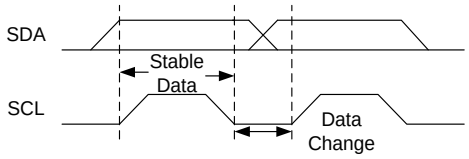


Data

After the master has sent the address and the slave has acknowledged it, data can be transferred eight bits at a time. Both the master and the slave can transmit and receive, so together they control the SDA line.

Data on the SDA line must be stable before the rising edge of SCL. Data on SDA can change only when SCL is LOW; see Figure 6.

Figure 6. SDA Data Change



If the master is performing a write operation, it writes out eight bits of data on SDA and provides eight clock cycles on SCL. After the master sends eight bits of data, the slave must send an ACK or NAK on the ninth clock cycle; see Figure 7.

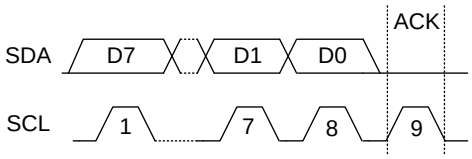
- ACK = Slave has room for more data
- NAK = Slave cannot take any more data

The alternate condition is that the master wants to read data from the slave. In this case, the master provides the eight clock cycles but the slave transmits eight data bits on SDA. After the eight data bits are transferred, the master must send the ACK or NAK.

- ACK = Master wants to read more data
- NAK = Master is done reading

Note The slave does not have a way to inform the master that it has no data to send.

Figure 7. Eight Data Bits Followed by an ACK Bit

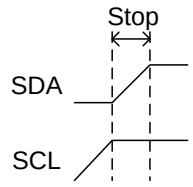


Stop

After all bytes are sent, the master sends a stop condition. A stop indicates that the current transaction is complete and the bus is free; see Figure 8.

Stop = LOW to HIGH transition on SDA when SCL is HIGH

Figure 8. Stop Condition



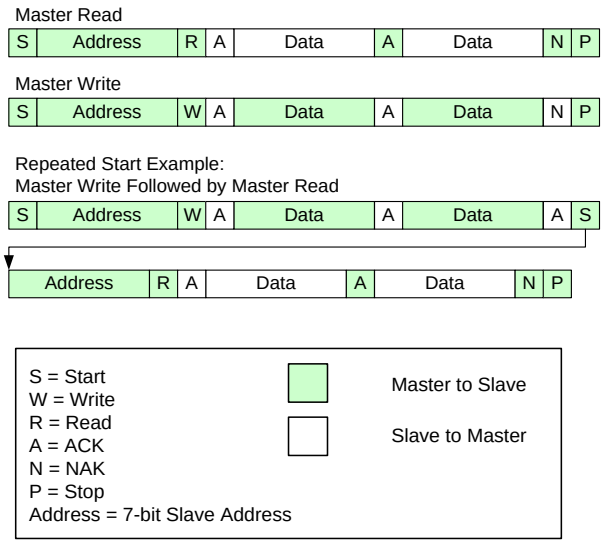
The master can send a repeated start condition instead of a stop. A repeated start condition is physically the same as a start. The repeated start allows the master to maintain control of the bus when it wants to address a slave device to start a new transaction or change the direction of data flow.

A repeated start is needed when a master wants to write data to a specific slave and then turn around and read data from that same slave. Using a repeated start allows the master to maintain control of the bus. If the master uses a stop, other masters on the bus can gain control.

Putting It All Together

Figure 9 shows a complete transaction for a master read of two bytes and a master write of two bytes. It also shows a repeated start that first writes two bytes and then reads two bytes.

Figure 9. Complete Transaction

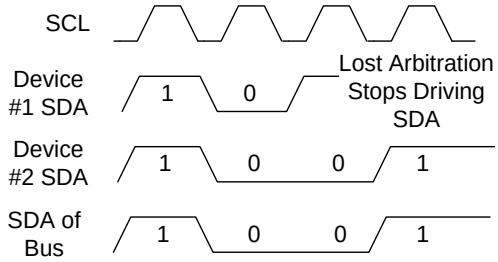


Arbitration

The I²C protocol allows multiple masters to communicate on the same bus. Both masters may start communicating at the same time. To prevent loss of data, each I²C master must check the I²C bus to ensure that the data on the bus matches what it intended to put on the bus. If the data does not match, the I²C master that loses arbitration must stop driving the SDA line and must wait until the bus is free before sending data again.

As Figure 10 shows, the first master that leaves SDA HIGH while another tries to drive it LOW loses arbitration and must wait.

Figure 10. Arbitration



For more information on I²C and its protocol, see the I²C Specification from Philips (NXP).

I²C in PSoC 1

In PSoC 1, a dedicated I²C hardware block handles I²C transactions, removing much of the processing burden from the CPU and freeing the CPU for important real-time tasks. The basic structure of the block is similar for most PSoC 1 devices; see Table 1 for differences among the part families.

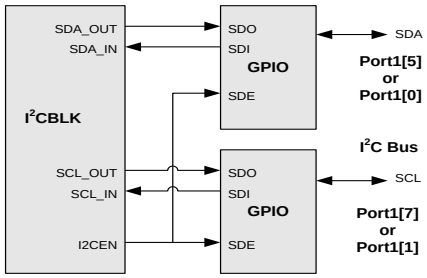
Table 1. I²C Hardware Block Differences

Device	Master	Slave	HW Addr Match	Two I²C Blocks
20x34	No	**Yes**	No	No
20xx6A	No	**Yes**	**Yes**	No
21x23	**Yes**	**Yes**	No	No
21x34	**Yes**	**Yes**	No	No
22xxx/21x45	**Yes**	**Yes**	No	No
23x33	**Yes**	**Yes**	No	No
24x23A	**Yes**	**Yes**	No	No
24x94	**Yes**	**Yes**	No	No
27x43	**Yes**	**Yes**	No	No
28xxx	**Yes**	**Yes**	**Yes**	**Yes**
29x66	**Yes**	**Yes**	No	No

Hardware

As Figure 11 shows, the hardware block allows Ports 1.5, 1.0, 1.7, and 1.1 to connect to the I²C bus. Cypress recommends avoiding Ports 1.1 and 1.0 for I²C; these lines are used for programming.

Figure 11. I²C Hardware Block



The hardware block is a simple block that handles all of the status and timing requirements of the I²C transaction. This block generates the I²C clock when it is in master mode. It also shifts I²C data in and out of PSoC 1 and reports the status of I²C transactions and errors.

Interrupts and Transaction Queuing

The block is capable of receiving or transmitting only one byte of data at a time. At each byte boundary, the block will generate an interrupt. The CPU must service the interrupt and provide more data to the block, or it must read the data that the block received. The CPU need not service the block immediately, because the block holds the SCL LOW until the CPU releases it; this process is referred to as clock stretching. For more information, see the Clock Stretching and Interrupt Latency section.

The block is capable of queuing only one transaction at a time. Multiple starts cannot be queued in the block. Therefore, user code must ensure that the current I²C transaction is complete before another transaction can be initiated.

The block automatically detects and reports arbitration conditions in multimaster environments. In the case of an arbitration event, the block reports to the CPU that it has lost arbitration. User code checks to see if arbitration has been lost. If so, then the code retries the transfer.

The I²C hardware block will generate an interrupt on three conditions: a bus error, a stop, or a byte complete.

Bus errors occur when there is a misplaced start or stop on the bus. When this occurs, all devices on the bus must stop their current transfer and return to an idle state.

If enabled, the stop interrupt occurs every time there is a stop condition on the bus.

The byte complete interrupt is triggered at different points depending on the direction of data flow; see Table 2. This table applies for both address and data transfers.

Table 2. Byte Complete Interrupt

Mode	Master	Slave
Transmitter	After 8 bits of data + ACK/NAK	After 8 bits of data
Receiver	After 8 bits of data	After 8 bits of data + ACK/NAK

For more information on the functionality of the I²C hardware block, see Appendix A and the I²C sections of the PSoC Technical Reference Manual.

Hardware Blocks in CY8C28xxx

Part family CY8C28xxx offers two separate I²C hardware blocks, allowing hardware connections to more than one I²C bus at a time. In addition, each block provides hardware address matching. The hardware interrupts the CPU only on an address match. However, it does not wake PSoC 1 out of a sleep state. After the address, the hardware interrupts the CPU according to the conditions in Table 2.

Each of the hardware blocks in part family CY8C28xxx allows for additional I²C pin connections either on Ports 1.2 and 1.6 or on Ports 3.0 and 3.2.

Having two I²C hardware blocks allows for many powerful applications. Table 3 lists some of the unique applications that can be achieved.

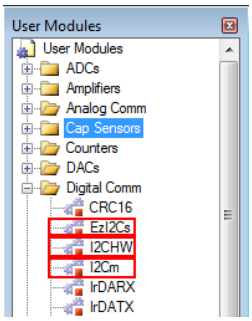
Table 3. Two I²C Configurations with Common Use Cases

Configuration	Common Uses
Two I²C slaves	Multi-I²C bus systems I²C shared memory device
One I²C slave, one I²C master or multimaster	I²C bus switch I²C hot-swap controller I²C buffer Debugging
Two I²C masters or multimasters	Increase bandwidth

I²C User Modules

Cypress provides a set of preconfigured and precoded I²C user modules (UMs), located in PSoC Designer in the UM Catalog; see Figure 12.

Figure 12. I²C UMs in PSoC Designer



When these UMs are placed in a design, you will not see them appear in the chip view within PSoC Designer, because the UMs either use the dedicated hardware block or are a software master. They use digital blocks only with the 28xxx family; in that case, the chip view shows both I²C hardware blocks visually so you know which block you have used and how many remain; see Figure 13.

Figure 13. Two I²C Blocks in 28xxx



I²C UMs, which include EzI2Cs and I²CHW, provide a level of abstraction over the I²C hardware block. This section briefly describes each UM and explains when it will be needed in a design. For more information, see the specific UM datasheet.

EzI2Cs

EzI2Cs operates exclusively as a slave; there is no master version. If a design requires master operation, use I²CHW or I²Cm.

EzI2Cs, as the name implies, is an easy-to-implement I²C slave interface. It is a firmware layer on top of the I²C hardware block. It implements many of the firmware requirements mentioned in the hardware block description found in Appendix A.

EzI2Cs is unique; it requires minimal user knowledge of how the I²C bus works. It allows you to set up a data structure in user code and expose that structure to the I²C master. All I²C transactions happen in the background through interrupts. You need not worry about any of the I²C functionality after the UM is started. The application code just needs to update the data structure with data for the master to read and then check and process data that the master writes.

This UM is best suited for situations when PSoC needs to stream data continuously to a master device and when you want to write minimal I²C code. For example, the UM is popular in CapSense® applications. PSoC reads the state of CapSense buttons and the master device communicates continuously with PSoC to see if a button has been pressed. In this situation, expose the CapSense variables to the EzI2Cs and the master can easily read the state of the CapSense buttons.

EzI2Cs is patterned after an I²C-based memory such as EEPROM. It uses subaddressing to write or read to specific data locations within the exposed I²C data structure. For example, consider an I²C data structure that appears as follows:

```
struct MyI2C_Regs {
    BYTE bStat;
    BYTE bCmd;
    int  iVolts;
    char cStr[6];
} MyI2C_Regs
```

The above data will be exposed to the master with the following memory map:

0x00	MyI2C_Regs.bStat
0x01	MyI2C_Regs.bCmd
0x02	MyI2C_Regs.iVolts(MSB)
0x03	MyI2C_Regs.iVolts(LSB)
0x04	MyI2C_Regs.cStr[0]
0x05	MyI2C_Regs.cStr[1]
0x06	MyI2C_Regs.cStr[2]
0x07	MyI2C_Regs.cStr[3]
0x08	MyI2C_Regs.cStr[4]
0x09	MyI2C_Regs.cStr[5]

If the master wants to write to iVolts, it first sends the slave address, followed by a subaddress (0x02), which indicates the offset of iVolts in the structure. To calculate the offset, count the number of bytes before iVolts; in this example, the subaddress is '2.' To extend the example, a subaddress of '0' will write to bStat, a subaddress of '1' will write to bCmd, and a subaddress of '4' will write to the first element of the cStr array. Figure 14 shows an example of a master writing data to iVolts in an EzI2Cs UM with a slave address of 0x04.

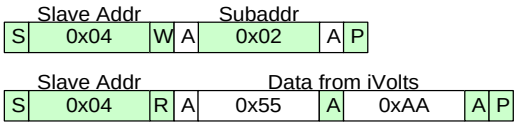
Figure 14. EzI2Cs Example: Write to iVolts

	Slave Addr			Subaddr		Data to iVolts				
S	0x04	W	A	0x02	A	0x55	A	0xAA	A	P

If the master wants to read iVolts, it first needs to address the PSoC slave and write the subaddress of '2.' It should then address the slave again and read out two bytes.

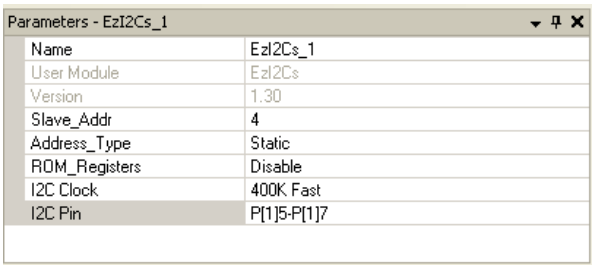
Each subsequent read will start at iVolts until a new subaddress is written. Figure 15 shows an example of a master reading the data in iVolts.

Figure 15. EzI2Cs Example: Read from iVolts



Let's take a look at how EzI2Cs can be implemented in a project. Figure 16 shows the parameters for the project.

Figure 16. EzI2Cs UM Parameters



Slave_Addr: This parameter sets the address of the EzI2Cs slave. If the ROM_Registers parameter is disabled, the Slave_Addr is a 7-bit address in the range 0 to 127. If the ROM_Registers parameter is enabled, the Slave_Addr is a 6-bit address in the range 0 to 63.

Address_Type: When the address type is set to static, the address of the EzI2Cs slave is fixed to the value set in the Slave_Addr parameter. If the address type is set to dynamic, the address can be changed in firmware using the EzI2Cs_SetAddr function. This is useful in applications in which more than one PSoC 1 EzI2Cs slave is in a bus and the address of the slaves can be set by configuring GPIO pins.

ROM_Registers: By enabling the ROM_Registers parameter, it is possible to expose data stored in a ROM array to an I²C master. When this parameter is enabled, the EzI2Cs is exposed to a master with two addresses. If the master wants to access the ROM memory space, it uses a 7-bit address with the seventh bit set. To access the RAM memory space, it uses a 7-bit address with the seventh bit cleared. For example, if the Slave_Addr is set to 0x04, the master addresses the slave with an address of 0x44 to access the ROM registers and an address of 0x04 to access the RAM registers. This is why the Slave_Addr should be a 6-bit address when the ROM registers are enabled.

I2C_Clock: This parameter sets the maximum speed at which the slave can operate. Remember that the maximum speed is based on a SYSCLK of 24 MHz. If the SYSCLK is set to 6 MHz or 12 MHz (SLIMO enabled) in the Global Resources, the maximum clock speed will reduce by the same factor. For example, if the I2C_Clock parameter is set to 400 kHz and the SYSCLK is set to 6 MHz, the actual speed of the slave will be 100 kHz. See the Clock Speeds section for more details.

I2C_Pins: This parameter selects the pins used for the I²C clock and data.

After the UM parameters are configured, follow these steps to make the EzI2Cs work in firmware.

1. Create a data structure to expose to the master. For example:

```
structMyI2C_Regs {
    BYTE bStat;
    BYTE bCmd;
int  iVolts;
char cStr[6];
} MyI2C_Regs
```

2. Start the EzI2Cs by calling the EzI2Cs_Start function.
3. Enable interrupt by calling the EzI2Cs_EnableInt function.
4. Expose the data structure to the master by using the EzI2Cs_RamSetBuffer function.

```
EzI2Cs_SetRamBuffer(sizeof(MyI2C_Regs),
2, (char*)&MyI2C_Regs);
```

   The first parameter sets the size of the buffer. The second parameter sets the write boundary and the third parameter initializes the pointer to the data structure.

   The EzI2Cs allows defining read/write permissions on the data structure through the SetRamBuffer function. For example, in the code given in step 1, the application sets read/write permission to bStat and bCmd variables and makes all other variables read-only by setting the write boundary to '2.' In this case, if the master tries to write to these read-only variables, the slave will generate a NAK and ignore the data written by the master.

5. In the main loop, keep updating the data structure with process data for the master to read and keep looking for fresh data from the master.

Data coherency is an important consideration when using the EzI2Cs UM. In the example, iVolts is a two-byte variable. It is possible that the CPU started to update this variable but only had time to write to one byte before the I²C master read both bytes. As a result, the master read incorrect data. For example, suppose that the variable iVolts has a value of 0x01FF. The CPU would have to update this with a new value of 0x0200. The CPU writes 0x02 to the MSB first and 0x00 to the least significant bit (LSB). If the I²C read occurs just after the CPU has written to the MSB, but before writing to the LSB, the master will read 0x2FF.

To combat this, it is best to use flags or semaphores between the master and slave to indicate when data is ready to be read or written. The EzI2Cs datasheet gives

more details on other ways to ensure that the data is coherent.

This application note includes a simple project that demonstrates how to use the EzI2Cs UM. For details on how to use the project, see Appendix B.

For a detailed description of how this UM works and how to use it, see the EzI2Cs datasheet.

EzI2Cs Summary and Important Notes
EzI2Cs is an easy-to-implement slave-only UM. EzI2Cs exposes a register structure to a master with clearly defined read/write permissions. When ROM registers are enabled, the slave will have two addresses, one for RAM space (seventh bit cleared) and one for ROM space (seventh bit set). For this reason, the slave address is limited to a 6-bit value when ROM registers are enabled. When dealing with multiple byte values in EzI2Cs, take care to ensure data coherency by using flags or semaphores between the master and slave.

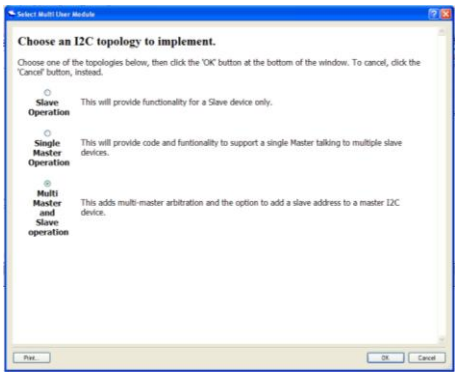
I²CHW

This UM is a firmware layer on top of the I²C hardware block and implements all the firmware tasks described in Appendix A. You can use this flexible UM as a slave, a master, or a multimaster slave.

I²CHW, unlike EzI2Cs, requires more code interaction. Check status bits to see if an I²C transaction occurred, reinitialize buffers when a transaction is complete, and clear status bits. Also, check the firmware for error conditions on a transaction.

Here's how to implement I²CHW in a project. When you double-click, the I²CHW topology selection window opens. See Figure 17.

Figure 17. I²CHW Topology Window



Slave Operation: Select this option if you need slave operation.

Single Master Operation: Select this option if you need a simple operation in a single master environment.

In the master operation, three parameters are available: Read_Buffer_Type, I2C_Clock, and I2C_Pin. Configuration of these parameters is the same as that described for the slave operation below.

Multimaster and Slave Operation: Select this option if you need the I²CHW as a master and slave in a multimaster environment.

Depending on the option you select, the UM parameter window will populate.

Slave_Addr: This is the 7-bit slave address. Unlike EzI2Cs, I²CHW does not offer dynamic addressing or separate address for ROM buffers. In multimaster and slave operation, Slave_Addr is the address of the device when it is addressed as a slave by another master on the bus.

Read_Buffer_Type: When you select "RAM only," the application will set up the read buffer only in RAM. When you select "RAM or flash," the application exposes either a RAM buffer or a flash buffer to the master.

Communication Service Type: When you select "Interrupt," all I²C transactions are automatically handled inside the interrupt. When you select "Polled," I²C transactions are handled only when the foreground application calls the I2CHW_Poll function. The hardware keeps the I²C clock stretched until the application calls I2CHW_Poll. Cypress recommends the "Interrupt" option.

I2C_Clock: This parameter selects the speed of the slave. As explained in the clock parameter of the EzI2Cs slave, the speed is based on a SYSCLK of 24 MHz.

I2C_Pin: This parameter selects the pins for the I²C. Options are P1[0]-P1[1] and P1[5]-P1[7] for all devices except for CY8C28xxx. For CY8C28xxx, P1[2]-P1[6] and P3[0]-P3[2] options are also available.

Firmware

After the topology and parameters are configured, follow these steps to get the I²CHW working in firmware.

Slave Operation

1. Declare a read buffer in RAM (or flash) for the master to read from. For example:

```
BYTE ReadBuffer[16];
```

   The buffer can also be a structure such as:

```
struct ReadBuffer {
    BYTE bStatus;
    int iVolts;
}ReadBuffer;
```

2. Declare a write buffer in RAM for the master to write to. For example:

```
BYTE WriteBuffer[16];
```

   The buffer can also be a structure such as:

```
struct WriteBuffer {
    BYTE bCmd;
    int iDACCounts;
}WriteBuffer;
```

3. Enable global interrupts by calling the M8C_EnableGInt macro.

   ```
   M8C_EnableGInt ;
   ```

   It is important to enable global interrupts, because all of the I²C operations take place in the background inside the I²C ISR.

4. Call the I2CHW_Start, I2CHW_EnableSlave, and I2CHW_EnableInt functions to start the slave.

   ```
   I2CHW_Start();
   I2CHW_EnableSlave();
   I2CHW_EnableInt();
   ```

5. Initialize the read buffer using I2CHW_InitRamRead.

   ```
   I2CHW_InitRamRead(ReadBuffer,
   sizeof(ReadBuffer));
   ```

   This function initializes a pointer to the read buffer and sets the size of the buffer. Whenever the master tries to read from the slave, data from the buffer is transmitted to the master automatically. If the master tries to read more bytes than the size of the buffer, the last byte of the buffer is transmitted repeatedly.

6. Initialize the write buffer using the I2CHW_InitWrite function.

   ```
   I2CHW_InitWrite(WriteBuffer,
   sizeof(WriteBuffer));
   ```

   This initializes a pointer to the write buffer and the buffer size. When the master writes data to the slave, the data is automatically deposited in the buffer. If the master tries to write a greater number of bytes than the buffer size, the slave will not acknowledge the extra bytes and those extra bytes will be discarded.

7. Call the function I2CHW_bReadI2CStatus and check for the I2CHW_RD_COMPLETE flag. If this flag is set, it means that the master has completed a read transaction. Clear the status flag and reinitialize the read buffer.

   ```
   // Check if a read operation is over
   if (I2CHW_bReadI2CStatus()
   I2CHW_RD_COMPLETE)
   {
   // Prepare fresh data for Master

   // Clear the flag
   I2CHW_ClrRdStatus();

   // Re-initialize the buffer
   I2CHW_InitRamRead(ReadBuffer,
   sizeof(ReadBuffer));
   }
   ```

   It is important to check for the RD_COMPLETE flag and reinitialize the buffer. Otherwise, on further reads from the master, the last byte from the buffer will be transmitted repeatedly.

8. Call the function I2CHW_bReadI2CStatus and check for the I2CHW_WR_COMPLETE flag. If this flag is set, it means that the master has completed a write transaction. Process the data, clear the flag, and reinitialize the write buffer.

   ```
   // Check if a write operation is over
   if (I2CHW_bReadI2CStatus() &
   I2CHW_WR_COMPLETE)
   {
   // Process data from Master

   // Clear the flag
   I2CHW_ClrWrStatus();

   // Re-initialize the buffer
   I2CHW_InitWrite(WriteBuffer,
   sizeof(WriteBuffer));
   }
   ```

   It is important to reinitialize the write buffer; otherwise, the slave will not acknowledge any further data from the master.

A simple project in this application note demonstrates how to use I²CHW as a slave that echoes the data written by an I²C master. For details on the project, see Appendix B.

Master Operation

1. Declare a read buffer in RAM (or flash) where data read from the slave will be stored. For example:

   ```
   BYTE ReadBuffer[16];
   ```

   The buffer can also be a structure similar to the one discussed in the slave operation section above.

2. Declare a write buffer in RAM for data that will be written to the I²C slave. For example:

   ```
   BYTE WriteBuffer[16];
   ```

3. Enable global interrupts by calling the M8C_EnableGInt macro.

   ```
   M8C_EnableGInt;
   ```

   It is important to enable global interrupts, because all of the I²C operations take place in the background inside the I²C ISR.

4. Call the I2CHW_Start, I2CHW_EnableMstr, and I2CHW_EnableInt functions to start the master.

   ```
   I2CHW_Start();
   I2CHW_EnableMstr();
   I2CHW_EnableInt();
   ```

5. To write to a slave, use the I2CHW_bWriteBytes function.

```
I2CHW_bWriteBytes(0x50, WriteBuffer, 16,
I2CHW_CompleteXfer);
```

The first parameter is the slave address; use the 7-bit slave address. The bWriteBytes function automatically adds the read/write bit to the 7-bit address before transmitting the address on the I²C bus. The second parameter is the pointer to the buffer that has the data for the slave. The third parameter is the number of bytes to be written to the slave. The fourth parameter is the transaction type and can have three different values: I2CHW_CompleteXfer, I2CHW_NoStop, and I2CHW_RepStart.

When you use I2CHW_CompleteXfer, one full I²C transaction is completed, which includes the start bit, the address byte, the data bytes, and the stop bit. When you use I2CHW_NoStop, the stop is not generated after writing the data. A transaction with a NoStop can be followed by a transaction with the I2CHW_RepStart. Usually, Cypress recommends I2CHW_CompleteXfer for most I2C transactions.

The I2CHW_bWriteByte function initializes a pointer to the buffer, sets a count value, and initiates the start bit in the I²C hardware. After this, all the operations take place inside the ISR. The I²C hardware generates an interrupt on every byte complete; the ISR takes care of incrementing the pointer to the buffer, transmitting the next byte to the slave. When all the bytes are transferred, the ISR generates a stop.

6. Use the I2CHW_bReadI2CStatus function to check that the write operation is complete, and then clear the flag. For example:

```
while(!(I2CHW_bReadI2CStatus() &
    I2CHW_WR_COMPLETE));
    I2CHW_ClrWrStatus();
```

The above code waits until the WR_COMPLETE flag is set. You can use an “if” condition instead of a “while”; the processor can do other things when the I²C transaction takes place in the background.

7. Use the I2CHW_fReadBytes function to initiate a read from the slave. Use the I2CHW_bReadI2CStatus function and check for the I2CHW_RD_COMPLETE flag; then clear the read status.

```
I2CHW_fReadBytes(0x50, ReadBuffer, 16,
    I2CHW_CompleteXfer);
while(!(I2CHW_bReadI2CStatus() &
    I2CHW_RD_COMPLETE));
    I2CHW_ClrRdStatus();
```

Multimaster Slave Operation

In the multimaster slave mode, the I²CHW can act as both master and slave in a multimaster environment.

1. Call the I2CHW_Start function.
2. Call I2CHW_EnableMstr and I2CHW_EnableSlave functions to enable both master and slave modes and the I2CHW_EnableInt function to enable interrupt.

```
I2CHW_Start();
I2CHW_EnableMstr();
I2CHW_EnableSlave();
I2CHW_EnableInt();
```

3. The slave mode operation is similar to the single slave operation, but the function names are different.
   - Allocate read and write buffers; call these functions to initialize: I2CHW_InitSlaveRamRead and I2CHW_InitSlaveWrite.
   - Call I2CHW_bReadSlaveStatus.
   - Check for I2CHW_RD_COMPLETE and I2CHW_WR_COMPLETE flags. If these flags are set, reinitialize the buffers.
4. The firmware for the master side of the operations has to deal with some special cases. In a multimaster environment, when a master (let’s call this Master 1) tries to initiate a transaction to a slave, there are three different scenarios depending on the activities of other master(s). (Assuming that there is another master in the bus, let’s call it Master 2.)
   a. The bus is free and Master 1 initiates and completes a read or write transaction.
   b. Master 2 had already initiated a transaction with another slave on the bus; because of this, the bus is busy. Master 1 must wait until the bus becomes free and then initiate the transaction.
   c. Master 1 initiates a read or write transaction at the same time Master 2 initiates a transaction. In this situation, arbitration occurs. If Master 1 wins the arbitration, it completes the transaction. If Master 1 loses arbitration, it must retry the transaction after Master 2 completes its transaction. Appendix A gives details about arbitration.

   I²CHW provides separate functions for the master for handling the multimaster environment.
5. To write to a slave, call I2CHW_bWriteBytesNoStall. To read from a slave, call I2CHW_fReadBytesNoStall. These functions check if the bus is busy before initiating the transaction. If the bus is busy, these functions return 0xFF. The firmware should check the return value and retry the transaction if the return value is 0xFF. For example, the following code loops until the I2CHW_bWriteBytesNoStall returns a value other than 0xFF.

```
while(I2CHW_bWriteBytesNoStall(0x50,
WriteBuffer, 16, I2CHW_CompleteXfer) ==
0xFF);
```

You can use an “if” condition to check the return value instead of using a “while” loop, which is blocking. If the return value is 0xFF, you can retry the transaction after a fixed time interval.

6. When a read or write operation is started without a bus busy error, the firmware should next check if the

operation is completed or if the master lost arbitration. Following is an example code that does this.

```
while((!(I2CHW_bReadMasterStatus() &
I2CHW_WR_COMPLETE)) &&
(!(I2CHW_bReadBusStatus() &
I2CHW_LOST_ARB)));
if (I2CHW_bReadMasterStatus() &
I2CHW_WR_COMPLETE)
{
    // Write is completed successfully.
    // Clear flag
    I2CHW_ClrMasterWrStatus();
}
if (I2CHW_bReadBusStatus() &
I2CHW_LOST_ARB)
{
    //  Master  lost  arbitration.  Retry
later
}
```

The first line loops until either the write is completed or the LOST_ARB error flag is set. If the “while” loop terminates, the code checks to determine which condition caused the loop to terminate. If the WR_COMPLETE flag is set, then clear the flag. If the LOST_ARB flag is set, then retry the transaction later.

This application note includes a simple project that demonstrates how to use the I²CHW UM to read a standard EEPROM. For details, see Appendix B.

For more information, see the I²CHW UM datasheet and the I²C section of the Technical Reference Manual.

I²CHW Summary and Important Notes

I²CHW can operate in single master, single slave, or multimaster slave modes.

In slave mode, when a master completes a read or write, the buffers must be reinitialized for the next transaction to take place.

In the master mode in a multimaster environment, the firmware must check the return value of the bWriteBytesNoStall and fReadBytesNoStall functions to know if the bus is busy, and must retry the transaction.

In the master mode in a multimaster environment: When a read or write transaction is initiated, the firmware checks if the transaction has completed successfully or the master has lost arbitration to another master. If the master lost arbitration, the transaction must be retried.

I²Cm

I²Cm, another UM in PSoC 1, implements an I²C master through software manipulation of GPIO port pins. This UM does not use the I²C hardware block. There is no I²C software slave user module.

The advantage of this UM over I²CHW is that it can be used on any pair of pins instead of just P1.5, P1.7 or P1.0, P1.1. One drawback is that it requires more CPU overhead and does not support multimaster operation. It requires 100 percent of CPU during I²C transactions. A second disadvantage is that it is limited to bus frequencies of ≤100kHz.

I²Cm is best for applications in which multiple masters are needed in a single chip or when the pins that connect to the I²C hardware are not available. Cypress recommends using I²CHW for master operations if only one master is needed and if P1.5, P1.7 or P1.0, P1.1 are available.

Following are the steps to get this UM running in a project.

1. Start the UM using the I²Cm_Start function.
2. To write data from a RAM buffer (or a ROM buffer) to a slave, use the I2Cm_bWriteBytes function. This function accepts the slave address, pointer to the source data in RAM (or ROM), and the number of bytes to be transferred.
3. To read data from a slave, use the I2Cm_fReadBytes function. This function accepts the address of the slave, pointer to destination buffer where the read data must be saved, and the number of bytes to be read.

This UM works by manipulating the drive mode (PRTxDMx) and data registers (PRTxDR) of the port on which it is operating. For this reason, take care when manipulating data registers in user code; otherwise, I²C traffic is affected adversely. The best way to avoid issues is to use shadow registers when writing to the drive mode and data registers associated with the I²Cm pins.

When I²Cm is placed in the project, PSoC Designer automatically creates shadow registers for the data register and the two drive mode registers PRTxDM0 and PRTxDM1. Because the UM does not affect PRTxDM2, a shadow register is not created for this register.

These shadow registers are defined in *psocconfig.asm* and *psocgpioint.h* files. For example, if I²Cm is placed in Port0, the following variables are defined in the *psocconfig.asm* file.

```
; write only register shadows
_Port_0_Data_SHADE:
Port_0_Data_SHADE:          BLK     1
_Port_0_DriveMode_0_SHADE:
Port_0_DriveMode_0_SHADE:     BLK     1
_Port_0_DriveMode_1_SHADE:
Port_0_DriveMode_1_SHADE:     BLK     1
```

The following definitions are placed in *psocgpioint.h*.

```
extern BYTE Port_0_Data_SHADE;
extern BYTE Port_0_DriveMode_0_SHADE;
extern BYTE Port_0_DriveMode_1_SHADE;
```

The application code uses these shadow registers to write to the data and drive mode registers. For example, if the application wants to set P0[0], use the following code:

```
// Write to PRT0DR through shadow register
Port_0_Data_SHADE |= 0x01;
```

```
PRT0DR = Port_0_Data_SHADE;
```

To set P0[0] to strong mode using the PRT0DMx registers:

```
// Write to PRT0DM0 through shadow register
Port_0_DriveMode_0_SHADE |= 0x01;
PRT0DM0 = Port_0_DriveMode_0_SHADE;

// Write to PRT0DM1 through shadow register
Port_0_DriveMode_1_SHADE &= ~0x01;
PRT0DM1 = Port_0_DriveMode_1_SHADE;

// Write to PRT0DM2 directly
PRT0DM2 &= ~0x01;
```

For more information on the need to use shadow registers, refer to the Shadow Registers Database. For more information on this user module, see the I²Cm datasheet.

I²Cm Summary and Important Notes
I²Cm is a software implementation of I²C master and does not occupy the I²C hardware resource. I²Cm uses all the CPU resources while reading or writing to a slave. If the application code has to write to the data or drive mode registers of the port in which I²Cm is placed, take care to use the shadow registers to avoid problems with the I²C interface.

Special I²C Considerations

I²C offers an easy way for devices to communicate with one another. As with any design, problems may occur during the design process. This section seeks to answer common questions and prevent problems that may occur when designing with I²C. It includes the following topics:

- 7-bit and 10-bit addressing
- Pull-up resistor consideration
- Sharing I²C and ISSP pins
- Glitches during power-up
- SYSCLK versus I²C clock speeds
- Clock stretching and interrupt latency
- Hot swapping
- Glitch filtering
- I²C and sleep
- I²C and dynamic reconfiguration
- Dynamic addressing in I2CHW

I²C Addressing

There are two common ways used to describe the I²C slave address. One uses the seven bits of address and not the read/write bit (for example, 0x42). This is how Cypress's UMs treat the I²C slave address.

The other way is to include the read/write bit as part of the address (for example, 0x84/0x85).

Make sure that you know which addressing method the devices you are working with use.

At this time, Cypress does not support 10-bit slave addresses.

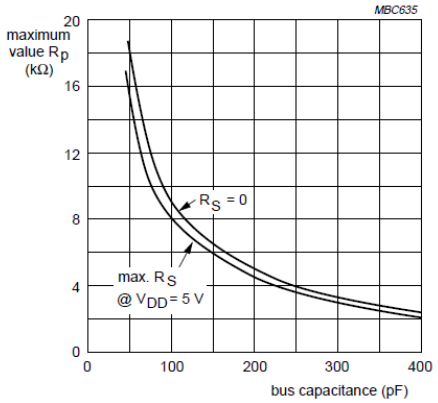
Pull-up Resistors

Another common design consideration with I²C is the value of the pull-up resistors. The selected value of the resistors depends on the communication frequency and the bus capacitance. A larger bus capacitance and pull-up resistor size causes longer rise time on the clock and data lines. The I²C specification provides a maximum rise time.

If the rise time on the bus exceeds the maximum, I²C communication does not occur properly. The I²C specification offers the graph shown in Figure 18 to determine the value of the pull-up resistors. R_S is a series resistor; the I²C specification defines the maximum R_S.

Pull-up resistors between 2.2 k and 4 k work for most systems.

Figure 18. Pull-up Resistor Value



I²C and ISSP Programming Conflicts

One common consideration when using I²C with PSoC is the choice of which pins to use. When using the hardware block, there are two pairs of fixed pins available for I²C: P1.5, P1.7 and P1.0, P1.1. (In the 28xxx family, you can also use P1.2 and P1.6 or P3.0 and P3.2.) You can also use pins P1.0, P1.1 to program PSoC. Pull-up resistors on these lines can cause in-system programming failures.

During programming, P1.0 uses the resistive pull-down drive mode to force a logic level LOW on the line. The pull-down resistor is approximately 5.6kΩ. This internal pull-down resistor and the external I²C pull-up resistor create a

voltage divider. The voltage divider applies an indeterminate or HIGH logic level on the line, which the programmer does not recognize as the desired LOW logic level. This causes programming to fail. See the DC programming specifications and GPIO logic levels in the device specific datasheets to determine which voltage levels are appropriate.

The best solution is to use P1.5, P1.7. Only use P1.0, P1.1 when it is necessary. Another option is to change the drive mode in PSoC to pull up for P1.0, P1.1, instead of using external pull-up resistors. The internal pull-up resistors are approximately 5.6 kΩ.

Pin Glitches at Power-up

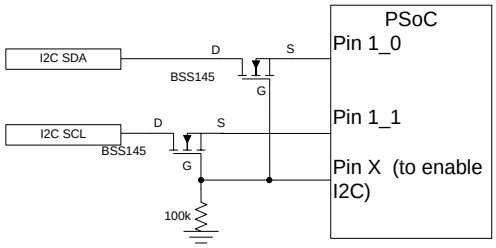
As already stated, P1.0 and P1.1 are used for programming, which means that these pins behave differently at power-up than other pins.

After the part is released from reset, P1.0 drives out a strong HIGH. This will cause issues if one of the other I²C devices on the bus is driving the line LOW. P1.1 drives out a resistive LOW. As stated earlier, this will create a voltage divider and an intermediate voltage on the bus, which other devices may not recognize. After a set amount of time, P1.0 transitions to a resistive LOW, causing an intermediate voltage on that line.

Be aware of this behavior and how it will affect I²C devices on the bus, if other I²C devices are powered and connected to P1.0 and P1.1 when you start PSoC.

To avoid this issue, use P1.5 and P1.7 for I²C communications. Or take steps to minimize the impact of the pin behavior on P1.0 and P1.1 during reset. This can be as simple as keeping all other devices in reset until PSoC has started up. Or implement a more complicated solution such as gating the output of PSoC with a transistor or logic gates. See Figure 19.

Figure 19. Isolating PSoC



Notes

- It is important to place the N channel FETs used in this block so that the source of the FET is attached to the PSoC GPIO pin.
- The FETs must be N channel and must have a VGS_{TH} rating of ≤ PSoC V_{dd}. For example, if the PSoC V_{dd} = 3.3 V, choose an N channel FET with a rating of VGS_{TH} ≤ 3.3 V.
- Set pin X to strong and write a '1' to it to connect to the I²C bus.

Clock Speeds

As stated earlier, the I²C hardware is responsible for generating the clock on SCL when it is in master mode. The available I²C clock frequencies in PSoC 1 are 50 kHz, 100 kHz, and 400 kHz. These frequencies are based on hardwired clock dividers of the system clock (SYSCLK).

These clock dividers produce sampling clocks that oversample the I²C lines 16 or 32 times. Table 4 lists the internal sample rate.

Table 4. Internal Sampling Rates

Clock Rate	SYSCLK Pre-Scaler	Internal Sampling Clock Frequency (SYSCLK=24 MHz)	Samples Per Bit
50 kHz	/16	1.5 MHz	32
100 kHz	/4	1.5 MHz	16
400 kHz	/16	6 MHz	16

The internal sampling clocks assume that SYSCLK is at 24 MHz. If SYSCLK is slower than 24 MHz, the I²C clocks are slower. For example, if SYSCLK is 12 MHz, then the available speeds are 25 kHz, 50 kHz, and 200 kHz. If SYSCLK is 6 MHz, the available I²C speeds are 12.5 kHz, 25 kHz, and 100 kHz. (See Table 5.)

Table 5. Actual Frequency versus IMO and UM Setting

UM Setting	IMO (SYSCLK) Setting		
	24 MHz	**12 MHz**	**6 MHz**
400 kHz	400 kHz	200 kHz	100 kHz
100 kHz	100 kHz	50 kHz	25 kHz
50 kHz	50 kHz	25 kHz	12.5 kHz

Note SYSCLK is separate from CPU_Clock.

When operating in slave mode, the same rules apply for the highest clock speed that the I²C block can read. The oversample clock is used to monitor the I²C lines. Setting the speed in slave mode indicates how often the clock and data lines are oversampled by the hardware.

If the I²C clock frequency is 400 kHz and the hardware is configured for 100 kHz, the hardware will not receive data properly. One common mistake is to set the I²C slave clock frequency in PSoC Designer to 100 kHz and SYSCLK to 6 MHz. The assumption that the slave will operate on the 100 kHz bus is incorrect. Because SYSCLK is at 6 MHz, the slave can only operate on a 25-kHz bus. Therefore, to operate on a 100-kHz bus with a SYSCLK of 6 MHz, choose 400 kHz as the I²C speed in the user module properties.

The frequency of SYSCLK depends on the supply voltage for PSoC. If the supply voltage is above 3 V, then most devices have a 24-MHz SYSCLK. If the supply voltage is below 3 V, then many devices have a 6-MHz SYSCLK. Several devices have a 12-MHz SYSCLK option. Further, the value of SYSCLK can be changed within PSoC Designer. When using I²C, do not change the frequency of SYSCLK or the CPU_CLK; this can cause glitches on the I²C lines. For more information on clocks, see the clock section of the Technical Reference Manual.

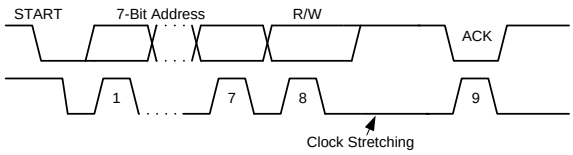
When using an external clock, be sure to use the correct I²C clock speed.

Clock Stretching and Interrupt Latency

Clock stretching is the process in which a slave device holds the clock line LOW, thus stalling further communication on the bus by the master. The slave device typically stretches the clock so it can process information it is receiving from the master or prepare more data to send to the master. This stretching can be done at any point of the transaction. PSoC stretches the clock after the byte complete interrupt. See Figure 20.

The I²C specification indicates that this is an optional feature; not all I²C devices need to support clock stretching. However, all Cypress PSoC 1 I²C slave UMs stretch the clock. If you use a master that does not support clock stretching, the bus can lock up and fail to reset.

Figure 20. Clock Stretching Example

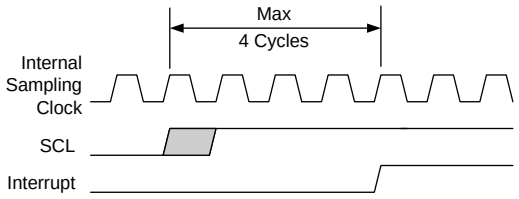


The time the PSoC slave spends stretching the clock depends on whether the master is reading or writing to the slave and whether other interrupts occur in the system. It also depends on the CPU speed.

PSoC 1 devices will stretch the clock a majority of the time for I²C bus speeds of 100 kHz or greater. The ISR code in the EzI2Cs and I2CHW UMs contain 150 to 300 CPU instruction cycles. With a 24-MHz CPU clock, the ISR takes approximately 6 to 13 µs to execute. The SCL line is released at the end of the ISR code.

The ISR is triggered three or four internal sample clock periods after the rising edge of SCL; see Figure 21. This is due to an internal glitch filter on SCL; see Figure 22. Refer to Table 4 for the frequency of the sampling clock. For 100 kHz, the sample clock is 1.6 MHz, which means that the ISR will fire ~2.5 µs after the rising edge of SCL. The nominal period of a 100-kHz clock is 10 µs. If it takes 2.5 µs for the ISR to be triggered and the ISR takes 6 to 13 µs, the clock will be stretched in most cases.

Figure 21. Byte Complete Interrupt Timing



Transmit: Ninth positive edge SCL
Receive: Eighth positive edge SCL

Using this information, you can determine if the clock will be stretched and for how long. However, if there are other interrupts in the system, the time spent in those interrupts must be considered.

To minimize clock stretching in PSoC, the first step is to run the CPU at 24 MHz. Next, the I²C clock speed must be 100 kHz or lower. Last, having minimal interrupts reduces clock stretching.

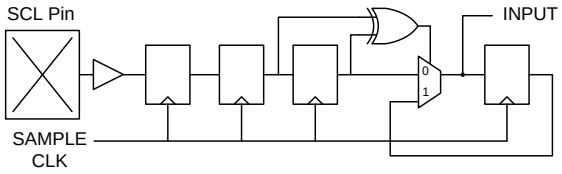
Hot Swapping

Hot swapping is the process of attaching powered devices to an unpowered PSoC. I²C in PSoC is not designed to be hot swappable. There are several factors to consider when hot swapping PSoC. The first issue is back powering. If PSoC is not powered but one of the external pins is at a high voltage level, then there is a possibility of back powering the PSoC device. This is not a desirable situation, because PSoC may execute in unexpected ways and the I²C lines may be adversely affected.

Glitch Filtering

The input of the SCL has a glitch filter, which Figure 22 shows.

Figure 22. SCL Glitch Filter



The input is double-synchronized to the sample clock, and then it is glitch-filtered. The raw input and the delayed input must match for a signal to pass through. Because the sample clock is either 1.5 MHz or 6 MHz, glitches less than 666 ns and 166 ns are suppressed.

I²C and Sleep

When using I²C in designs that enter and exit sleep mode, you need to take special design considerations into account.

First, it is important that all I²C transactions are complete before the I²C enters sleep mode. Otherwise, when the part wakes up, the I²C block can erroneously interpret data

as an address or vice-versa. After confirming that all I²C traffic has stopped, follow these steps:

- Configure the I²C pins in a HIGHZ drive mode.
- Disable the I²C block. This is generally done by calling the stop() API of the UM.
- Clear any pending I²C interrupts.

After these conditions are met, PSoC can be put to sleep. When the part wakes from sleep, follow these steps to ensure proper I²C operation after sleep:

- Ensure that no I²C activity is occurring on the bus.
- Call the appropriate start API.
- Configure I²C pins for open-drain drives LOW.
- Enable interrupts.

If you follow these steps, you can avoid most errors when using I²C in conjunction with sleep.

I²C and Dynamic Reconfiguration

I²C UMs should never be loaded or unloaded through dynamic reconfiguration. They should always be present. If they are loaded and unloaded, I²C errors will occur.

The I²C UMs should be located either in the base configuration or in a separate overlay that is always loaded.

Dynamic Slave Addressing in I²CHW UM

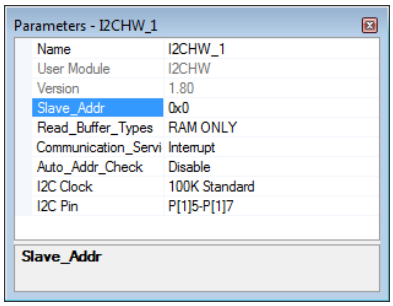
The EzI2Cs UM allows you to programmatically change the I²C slave address on the fly. However, this functionality is not available in the I²CHW UM; to achieve the same functionality, do the following:

In the I²CHW UM, the address from the master is processed inside the *I2CHW_1int.asm* file. The following code does the address matching.

```
mov A, reg[I2CHW_1_DR]
and F, 0xF9
rrc A
xor A, I2CHW_1_SLAVE_ADDR
```

When the address is received from the master, it has a 7-bit address and the read/write bit. The 'rrc A' drops the read/write bit and compares the 7-bit address with the constant I2CHW_1_SLAVE_ADDR, which the device editor creates according to the Slave_Addr set in the UM parameters for the I²CHW. This constant is located in the *I2CHW_1.inc* file.

Figure 23. I²CHW UM Slave Address



Follow these steps to replace the constant with a RAM variable:

1. Create a RAM variable to hold the dynamic slave address. In the *I2CHW_1int.asm* file, just below the variable allocations, is a custom user code area. Under the custom declaration area, add this code:

   ```
   export _I2CSlaveAddress
   export I2CSlaveAddress
   ```

   Under the variable allocation area, add the following:

   ```
   Area InterruptRAM(ram)
   _I2CSlaveAddress:
   I2CSlaveAddress:   BLK   1
   ```

   The variable with the '_' allows you to modify this in C code.

2. Modify the code that does the address comparison.

   ```
   mov A, reg[I2CHW_1_DR]
   and F, 0xF9
   rrc A
   xor A, [I2CSlaveAddress]
   ```

   Because this code is inside a custom user code area, changes to the UM library file are preserved during application generation. Note that if you rename the UM, changes will be lost and must be made again.

3. Add a reference to the I²C slave address variable that is defined in the *I2CHW_1int.asm* file by adding the following code in *main.c* (or any other C file).

   ```
   extern BYTE I2CSlaveAddress;
   ```

The SCL Line Gets Stuck LOW

A common issue with I²C is the SCL getting stuck LOW. Follow these guidelines for debugging and fixing this issue with PSoC 1 devices:

1. Are you using EzI2Cs with PSoC Designer 5.0 SP5? If yes, there is a known bug with this version of the UM that causes the SCL to be stuck LOW. Update to the latest version of PSoC Designer to fix this issue. See the article I²C Clock Permanently Stuck to Logical LOW.
2. Are you using PSoC Designer version prior to PD5.0? If yes, then there are glitches on P1.5, P1.7 at startup.

This issue has been resolved in the latest version of PSoC Designer. For more information, see the article Glitch on the I²C Lines During Power-Up.

3. Does the master device support clock stretching? If no, then the communication between the slave and master will become unsynchronized and undesirable behavior may occur, including the SCL being stuck LOW. If the master does not support clock stretching, there is no guarantee of PSoC functioning properly on the bus.
4. Does the CPU clock or SYSCLK change dynamically during code execution? If yes, this can cause glitches on the SCL and SDA. To avoid this problem, do not change clock speeds in code.
5. Are you enabling and disabling I²C interrupts in your code? If yes, make sure you are using the ResumeInt API and not the EnableInt API, which clears the interrupt. If there is a pending I²C interrupt and you clear it, SCL will be stuck LOW forever.
6. Can there be other I²C traffic on the bus when PSoC starts up? If yes, then there is a known bug that has been fixed in the latest version of PSoC Designer. If you do not have the latest version and cannot update, use the following workaround: In the chip editor, set the drive mode of the I²C pins to analog High-Z. In the main code, enable the I²C UM. Then set the drive mode of the I²C pins to open-drain drive LOW.
7. Is sleep being used in the project? If yes, follow the steps in the sleep section to avoid any issues.

The latest version of PSoC Designer has fixes and workarounds for a majority of the issues discussed above. Cypress always recommends using the latest version of PSoC Designer.

Summary

I²C is a simple two-wire chip-to-chip digital communication protocol. The protocol is master oriented but allows bidirectional communication on just two communication lines.

The Cypress PSoC offers several user modules for the implementation of I²C in a design that includes slave, master, and multimaster configurations. I²C communication in PSoC is easy and reliable if you follow the considerations that Cypress recommends.

About the Authors

Name: Todd Dust

Title: Application Engineer

Background: BSEE Seattle Pacific University

Name: M. Ganesh Raaja

Title: Application Engineer Principal

Background: Ganesh earned his Diploma in Electronics and Communications Engineering at Motilal Nehru Govt. Polytechnic in Pondicherry, India. He has about 20 years of experience in analog circuit design and microcontrollers. He also writes the blog PSoC Hacker on the Cypress website.

Appendix A

This appendix is for users who are curious about the detailed operation of the I²C hardware block in PSoC.

Hardware Registers

Several registers control the I²C hardware block and report its status.

I2C_CFG Register: Controls the configuration of the I²C hardware block. It controls the clock speed and the pins being used, and it indicates if the slave or master functionality is enabled. It also enables support for interrupts on stop conditions and bus errors.

I2C_SCR Register: Returns status flags from the I²C hardware block. It reports if a full byte (byte complete) is sent or received. This register indicates if a bus error has occurred or if arbitration is lost. It also determines if the last transaction was an address.

I2C_DR Register: This register holds the value of data sent or received. It shifts data in only if the data is an address, if the hardware block is configured as a slave and is addressed, or if the master has initiated a read.

I2C_MSCR Register: This register controls the master portion of the I²C transaction. It holds the bit that allows generating a start. The enable master bit in the I2C_CFG register must be set for this register to be available; otherwise, this register is held in reset.

Firmware Requirements

This section describes the operation of the hardware block for different configurations. Note that wherever the text mentions a requirement of firmware, all PSoC I²C user modules run this code.

Start Generation

If the start gen bit is set in the master status and control register, the hardware block generates a start condition and sends out the address in the data register. However, the hardware recognizes if another device has taken control of the bus. If an external start condition is detected, the hardware queues the current start until the bus is free. The start bit is not cleared until the hardware has successfully sent out a start condition or the firmware has cleared it.

Only one start can be queued at a time. If the master attempts to send two starts while the bus is busy, the hardware sends only the last address. As stated earlier, the user code must ensure that this situation does not occur.

Master Operation

In master mode, after the start and address are sent, the hardware waits until the ACK/NAK bit from the slave is received. On receiving the ACK/NAK bit, the hardware interrupts the CPU. Firmware then determines if the slave acknowledged or did not acknowledge the address. It does so by reading the last received bit (LRB) in the I2C_SCR register. If the bit is a zero, then the slave has acknowledged; if the bit is a one, then the slave has not acknowledged. Take appropriate action depending on this condition.

Firmware is also responsible for setting the direction of communication. This is done by setting the transmit bit in the SCR register. If a zero is written to the bit, the block is placed in receive mode. In this mode, the hardware interrupts the CPU when it receives eight data bits. After this interrupt, the firmware must determine if an ACK/NAK needs to be sent.

If a '1' is written to the transmit bit, the block is placed in transmit mode. In this mode, the hardware interrupts after the ACK/NAK is received from the slave device. The firmware again handles these cases.

When the SCR register is written, the master will start to generate clocks to either send or receive more data.

Slave Operation

When the hardware detects a start condition in slave mode, it shifts the next eight bits of data into the I2C_DR register. On receipt of the eighth bit, the hardware block interrupts the CPU and causes the address bit in the I2C_SCR register to go HIGH.

When the interrupt is posted, the hardware holds the clock line LOW. It is then the firmware's responsibility to read the incoming address and acknowledge whether the address is its own. The firmware must set the ACK bit in the SCR register appropriately. The firmware should also read the read/write bit of the address. If the bit is a '1,' then the transmit bit in the SCR register is changed to a '1.' After the CPU writes to the SCR register, the hardware releases the clock line, which allows the transaction to continue.

If the slave is configured as a transmitter, firmware must load valid data into the DR register before writing to the SCR register and releasing the bus. After the bus is released, the hardware shifts data out to the SDA line on the clock edges provided by the master. The hardware waits until it receives an ACK/NAK from the master and then interrupts the CPU. The firmware then must load new data or do nothing.

If the slave is configured as a receiver, the hardware interrupts after the eighth bit of data is received. The firmware then decides if it can receive more data. It must then set the ACK bit appropriately.

Stop Condition

When a transaction is completed in master mode, the hardware block generates a stop condition to indicate that the bus is free.

In slave mode, the reception of a stop puts the hardware block into an idle mode until it receives a new start condition.

Interrupt Sources

In the earlier examples, the hardware block interrupts on a byte-complete condition. The occurrence of this interrupt depends on the direction of communication.

- Transmit: Byte Complete Interrupt = Ninth bit
- Receive: Byte Complete Interrupt = Eighth bit

The hardware block allows two additional interrupt sources. Setting the appropriate bit in the I2C_CFG register enables these interrupts. The hardware is capable of interrupting on a stop condition. This is useful when operating in slave mode; the interrupt alerts the firmware that the current transaction is complete. The next available interrupt is on a bus error, which is a misplaced start or stop on the bus. If the hardware detects the interrupt, it stops its current activities and posts the interrupt; the firmware then decides what to do with this situation.

Arbitration

The hardware block can also indicate if the master loses arbitration. This is necessary when operating in multimaster mode. Arbitration occurs when two masters start writing at the same time. The hardware monitors the SDA line. If the master attempts to leave the bus HIGH but another master pulls it LOW, the master signals that it has lost arbitration. After the arbitration condition, the hardware does not control the SDA line but continues to clock the SCL line. On a subsequent byte-complete interrupt, it is the firmware's responsibility to view the lost arbitration bit to see if arbitration was lost during the previous transfer.

Basic I²C Flow

Figure 24 and Figure 25 show the basic flow for I²C transactions. This is the basic flow needed to implement I²C in PSoC 1 successfully. The provided UMs follow this flow with added overhead.

Figure 24. Flow Diagram for a Successful Slave Transmitter/Receiver

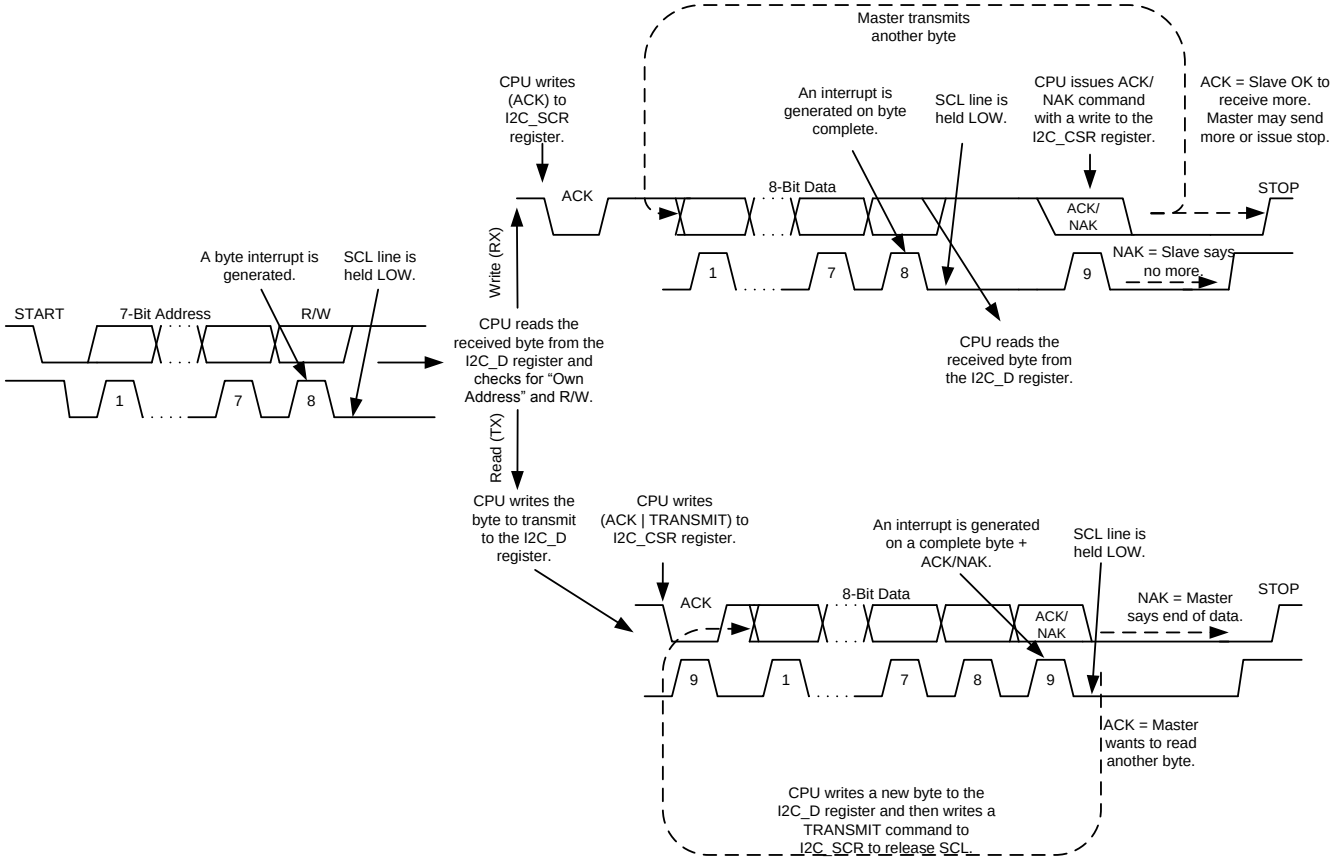
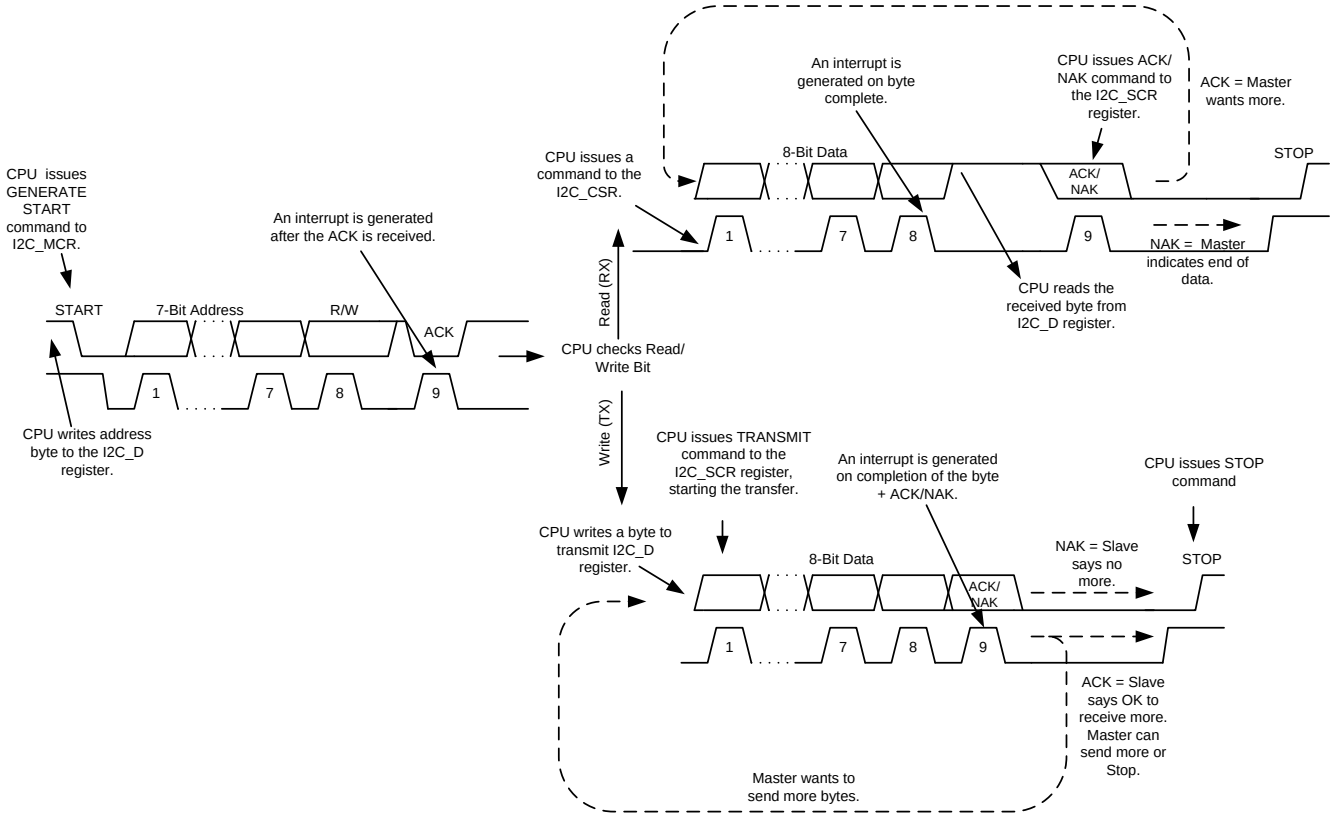


Figure 25. Flow Diagram for a Successful Master Transmitter/Receiver



Appendix B

This appendix describes various example projects that demonstrate the use of the I²C UMs.

EzI2Cs_ADC_LED_DAC Example Project

This project demonstrates how to configure the EzI2Cs UM. The project configures PSoC as an I²C slave and may be considered as an implementation of an I²C-controlled analog peripheral device and port expander.

This project creates the following data structure:

```
struct I2CRegs
{
    BYTE LEDValue;/* Updates LEDs */
    BYTE DACValue;/* Updates DAC */
    BYTE ADCValue;/* Reads ADC value */
}I2CRegs;
```

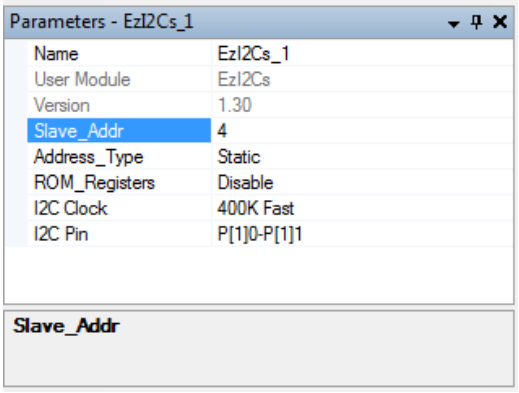
The structure is exposed to the I²C master through the following API call:

```
EzI2Cs_1_SetRamBuffer(sizeof(I2CRegs),    2,
(BYTE *) &I2CRegs);
```

The first parameter sets the size of the structure. The second parameter sets the number of parameters that have read-write permission. Parameters beyond this boundary are read-only. In this structure, LEDValue and DACValue are read-write and ADCValue is read-only. The final parameter is the pointer to the structure itself.

Configure the EzI2Cs UM in the chip view as shown in Figure 26.

Figure 26. EzI2Cs UM Configuration



Set the slave address to 0x04; set the address type to static; disable the ROM registers; set the clock speed to 400 kHz, fast mode; and use P1[0] and P1[1] as I²C pins. P1[0] and P1[1] are used in such a way that the ISSP port on the CY3210 PSoC 1 evaluation board can also be used to connect the I²C master.

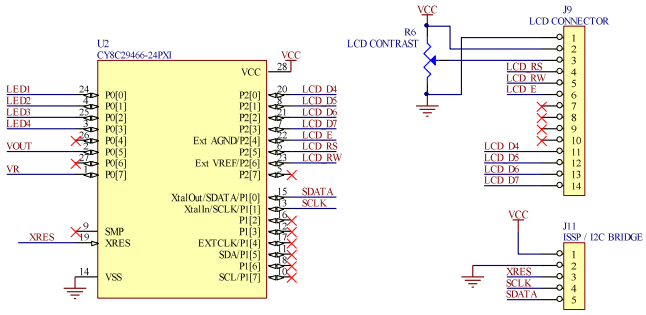
In firmware, an ADC is used to read the voltage on P0.7. You can simulate different voltages by connecting a potentiometer to this pin. The value read by the ADC is written to the variable ADCValue. A master device can read this variable.

In firmware, a local copy for LEDValue and DACValue are created. The values of LEDValue and DACValue in the I²C register structure are continuously compared with the local copies of these parameters. When the master writes a different DAC value or LED value, then the LED ports (P0[0] to P0[3]) and DAC are updated with the new values. The DAC output is available on P0[5].

Testing the Project

Figure 27 shows the setup for project testing. Use a CY3210 PSoC 1 evaluation board to wire the setup.

Figure 27. Schematic for EzI2Cs_ADC_LED_DAC_Project



P0[0] to P0[3] are connected to the LED1 to LED4 signals on J5. P0[7] is connected to the variable resistor VR. A digital multimeter is connected to P0[5] to monitor the DAC output. The LCD is connected to the LCD connector J9.

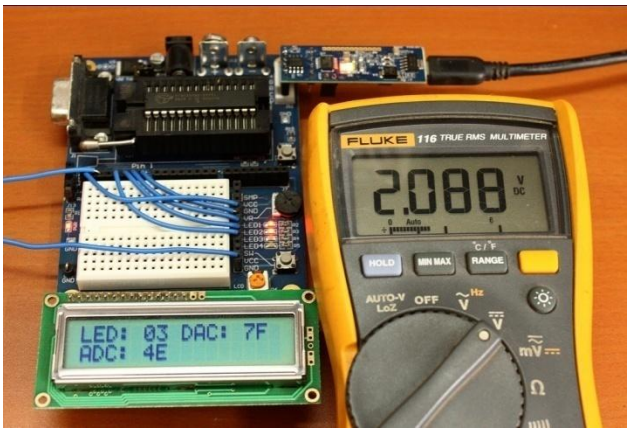
A CY3217 MiniProg1 or CY8CKIT-002 MiniProg3 can be used to program the device using the ISSP header on the CY3210 board. See Table 6.

A CY3240 USB-I²C Bridge PSoC Development Kit or MiniProg3 may be used as I²C master. Figure 28 shows the CY3210 board with the project in action. The CY3240 I²C-USB bridge is used here.

Table 6. Setup on the CY3210 Evaluation Board

PSoC 1 Pins	CY3210 Connections	Description
P0[0] to P0[3]	LED1 to LED4	Connect P0[3:0] to the 4 LEDs
P0[7]	VR	Potentiometer input
P0[5]	-	Connect multimeter
-	ISSP header (J11)	Connect MiniProg1 or MiniProg3 for programming
-	ISSP header (J11)	Connect Cy3240/MiniProg3 for I2C communication to PC

Figure 28. Test Setup for EzI2Cs Example Project



There are two methods for testing the project. You can use Bridge Control Panel software or an external MCU.

Using Bridge Control Panel Software to Test

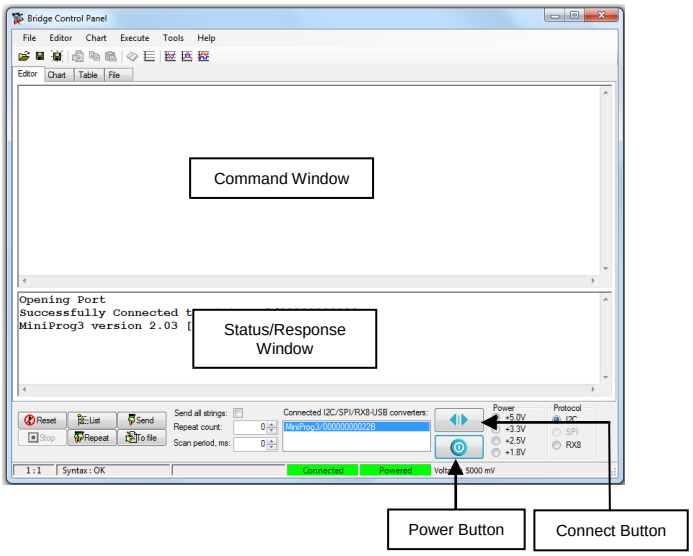
The Cypress Bridge Control Panel software acts as a graphical front end to communicate with PSoC I²C slave devices. It is useful for testing, tuning, and debugging programs that have an I²C slave interface. You can install the Bridge Control Panel with PSoC Designer and PSoC Programmer™.

After programming the device, connect the CY3240 or MiniProg3 to the ISSP connector. The CY3240 and the MiniProg3 have the pull-up resistors required for the SDA and SCL lines of the I²C.

The following steps show how to use the Bridge Control Panel to read the value of ADC and to control the LED and DAC outputs of the EzI2Cs_ADC_LED_DAC project.

1. Open Bridge Control Panel from the Windows start menu. It is located in the Cypress folder.
2. Select the MiniProg3 or the CY3240 from the device list and click the Connect button.
3. Next, click the Power button to supply power to the CY3210 test setup. (See Figure 29.)

Figure 29. Bridge Control Panel



4. To write to the LED and DAC parameters, type the following command in the command window of the Bridge Control Panel:

   ```
   w 04 00 03 80 p
   ```

   'w' is the write command. '04' is the slave address. '00' is the subaddress where the values have to be written. '03' is the value for LED. For a value of 03, LED1 and LED2 will be turned on. '80' is the DAC output. The output of the DAC is about 2.09 V for this value.

   When you enter this command in the command window and press Enter, the I²C master transmits the command to the EzI2Cs slave. Observe the state of LED and the output on P0[5] in the results window. (See Figure 30.)

   ```
   w 04+ 00+ 03+ 80+ p
   ```

   A '+' sign after each byte indicates that the EzI2Cs slave acknowledged the byte. A '–' sign indicates that the byte was not acknowledged.

   For example, try the following command:

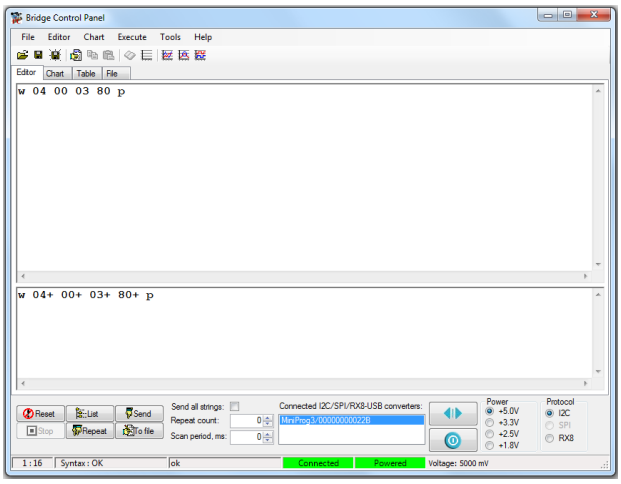
   ```
   w 04 00 03 80 55 p
   ```

   The response would be:

   ```
   w 04+ 00+ 03+ 80+ 55- p
   ```

   Because we have set the third byte in the EzI2Cs register structure as read-only, the slave will not acknowledge (NAK) the write to this register.

Figure 30. Write LED and DAC Values



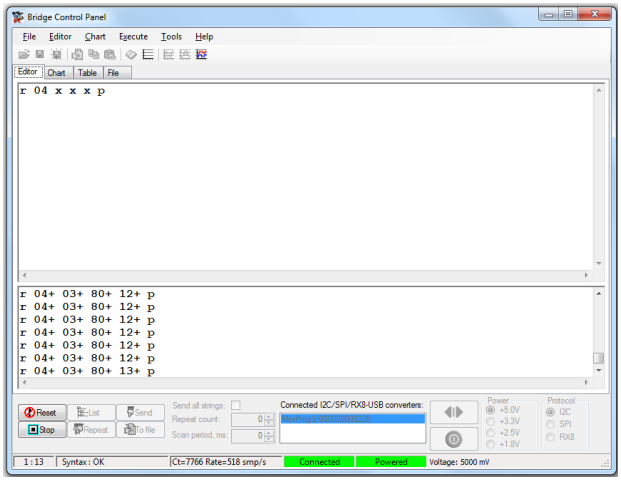
5. To read the ADC result, type the following command in the command window:

   r 04 x x x p

   'r' is the read command. '04' is the slave address. The three '*x*'s indicate how many bytes to read from the slave.

   Now click the Repeat button in the Bridge Control Panel and observe the result in the results window. (See Figure 31.)

Figure 31. Read All of the I²C Registers



   The last byte in the response is the ADC result. The second and third bytes show the values of the LED and DAC parameters respectively.

6. If you want to read only the ADC value and not the LED and DAC values (see Figure 32), first execute the following command:

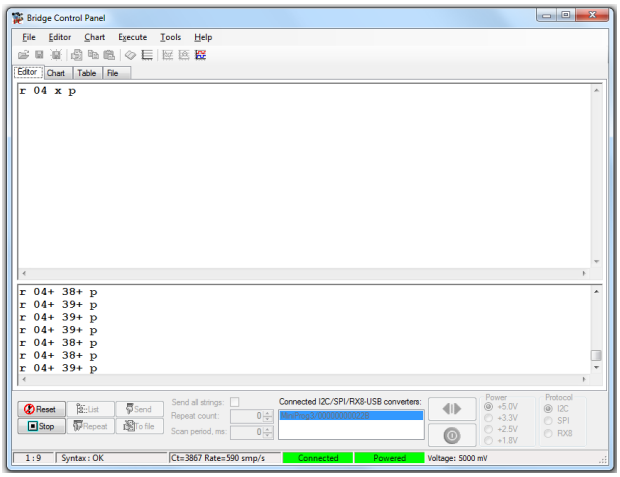
   w 04 02 p

   This write command sets the subaddress in the EzI2Cs. Any further read operation will take place on subaddress 0x0, which is the ADC value.

   Next, type the following command and select the Repeat button. Observe only the ADC value being read.

   r 04 x p

Figure 32. Read Only the ADC Register



Using an External MCU to Test

You can also test the project with another microcontroller acting as an I²C master. This section demonstrates the I²C interface with an Arduino Duemilanove board configured as an I²C master and assumes that you are familiar with the Arduino prototyping platform. For more information on this board and the Arduino platform, visit www.arduino.cc. Refer to Getting Started with Arduino for instructions on connecting the Arduino board to the PC and uploading the program.

The Arduino device is configured to have an I²C master interface on the dedicated pins (AIN4, AIN5) and one PWM output on pin 11.

The CY3210 is configured as an I²C slave having a 4-bit digital display (four LEDs), one analog output (DAC), one analog input (potentiometer connected to analog input pin), and an LCD display.

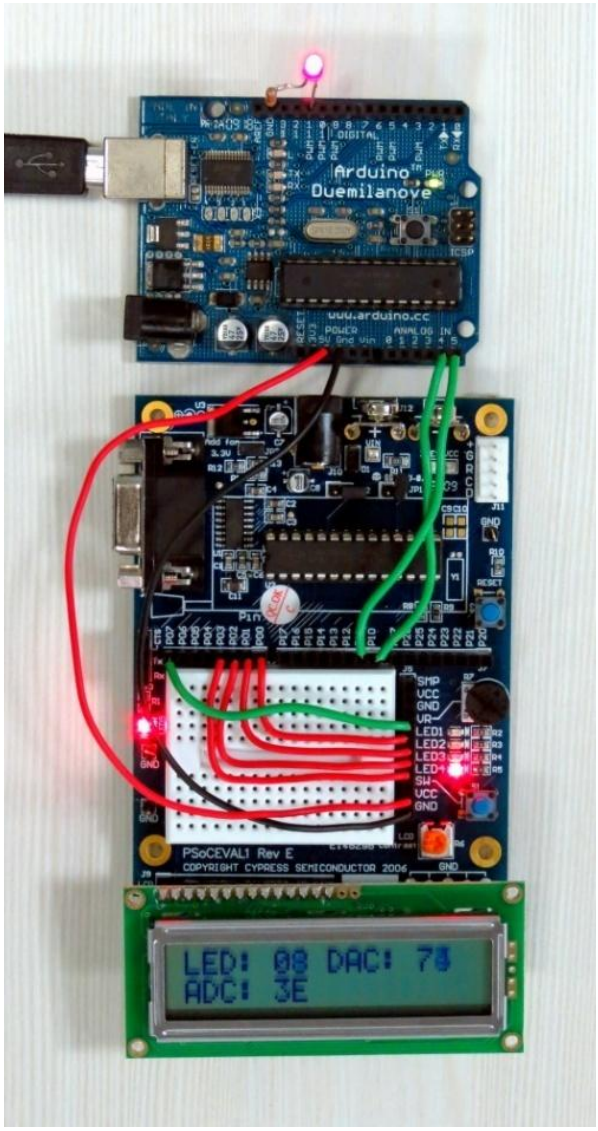
1. Program the CY3210 board with the example project.
2. Open the included Arduino project file "Arduino_I2C_Master_PSoC1_Slave.ino" using the Arduino software.
3. From Tools, select Board and then the Arduino Duemilanove option.
4. From Tools, select Serial Port and then the port on which the board is connected.
5. Download the program to the board using a USB A-B cable. Detach the USB cable after programming is complete.
6. Make hardware connections according to Table 7.

Table 7. Hardware Connections

Connections on Arduino Duemilanove		
Pin	**Connection**	**Description**
Pin 11 (PWM)	Connect an LED	PWM output on Arduino
Connections on CY3210-PSoC Evaluation Board		
Pin	**Connection**	**Description**
P00	LED1 on J5	Digital output Bit 0
P01	LED2 on J5	Digital output Bit 1
P02	LED3 on J5	Digital output Bit 2
P03	LED4 on J5	Digital output Bit 3
P05	Multimeter/Scope	Analog output (DAC)
P07	VR on J5	Analog Input (ADC)
CY3210 to Arduino Connections		
Arduino Pins	**PSoC 1 Pins**	**Description**
Analog in 4 (A4)	P10 on J7	SDA for I²C
Analog in 5 (A5)	P11 on J7	SCL for I²C
5 V	VCC on J5	Power PSoC 1 board using Arduino 5 V
GND	GND on J5	Ground

7. Connect a CY3217 MiniProg1 or CY8CKIT-002 MiniProg3 to the ISSP header on the CY3210 board. After programming, remove the programmer from the ISSP header.
8. Reattach the USB cable to the Arduino board. This will power both the Arduino board and the CY3210 board. Figure 33 gives a snapshot of the setup.

Figure 33. Arduino-to-PSoC Interface

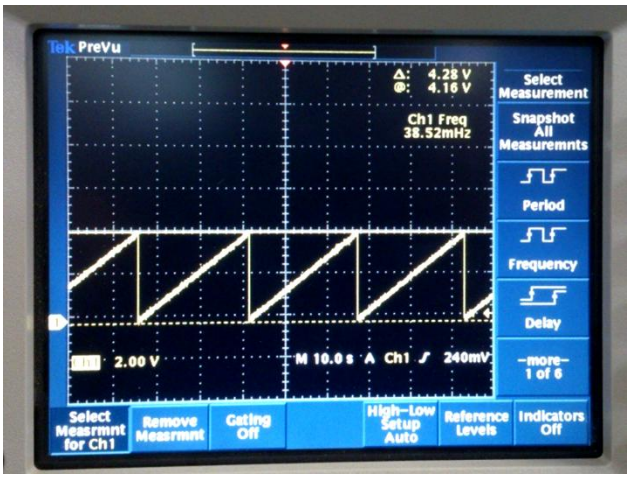


PSoC reads the voltage on P07 (connected to potentiometer) and stores the value in ADCValue. Arduino reads this value using the I²C and controls the PWM output on pin 11. Turning the potentiometer on the CY3210 varies the brightness of the LED on the Arduino board.

Arduino continuously sends a 4-bit pattern over the I²C interface. PSoC reads this value and writes to Port0 pins, which are connected to the four LEDs. Arduino also continuously sends an 8-bit digital value, which is converted to an analog voltage by the PSoC DAC. The DAC code sent is an incremental value from 0 to 255. Figure 34 shows the DAC output waveform on the PSoC pin (P05) when viewed on an oscilloscope.

This example thus shows how an external MCU can interface with a PSoC device using I²C.

Figure 34. DAC Output Waveform on P05



I²CHW Slave Example Project

This project demonstrates how to configure I²CHW as a slave.

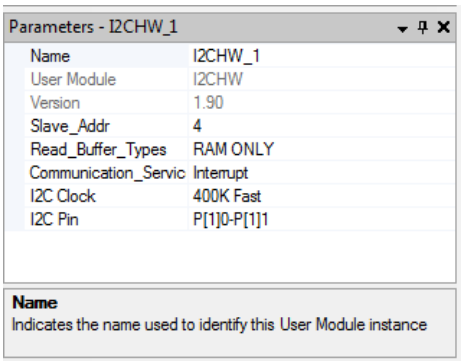
In this project, the data that is written to the I²CHW slave is echoed back to the master. This is done by configuring the read and write buffers as the same buffer. To do so, use the following API:

```
/*When master writes data it will write to
rxtxBuffer*/ I2CHW_1_InitWrite(rxtxBuffer,
10);
/*When master reads data it will read from
rxtxBuffer*/
I2CHW_1_InitRamRead(rxtxBuffer, 10);
```

The main code checks to see if a master device has read or written to the I²CHW slave. If it has, then the code will reset the buffers and clear the appropriate status flags.

Figure 35 shows the configuration of the I²CHW UM. The slave address for the device is 0x04.

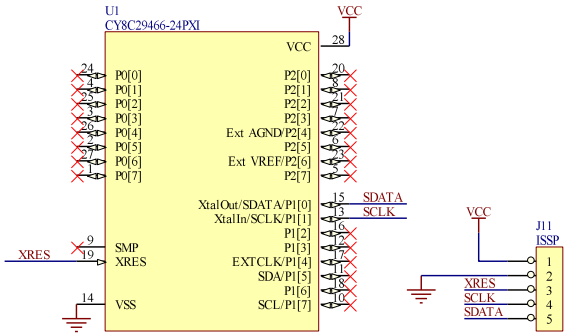
Figure 35. I²CHW UM Slave Configuration



Testing the Project

Figure 36 shows the setup for project testing.

Figure 36. Schematic for I²CHW_Slave Project



The only connections required to test the project are the ISSP connections. The same ISSP connector is also used to connect the I²C master. (See Table 8.)

Table 8. Connections for Project Testing

PSoC 1 Pins	CY3210 Connections	Description
-	ISSP header (J11)	Connect MiniProg1 or MiniProg3 for programming
-	ISSP header (J11)	Connect Cy3240 or MiniProg3 for I²C communication to PC

You can test the I²CHW slave project using the CY3210 PSoC 1 evaluation board.

Use a CY3217 MiniProg1 or CY8CKIT-002 PSoC MiniProg3 to program the device, using the ISSP header on the CY3210 board.

Use a CY3240 USB-I²C Bridge or MiniProg3 as the I²C master.

Figure 37 shows how to use the Bridge Control Panel to write and read values from the I²CHW slave. Refer to the EzI2Cs_ADC_LED_DAC example project for a brief description of how to set up the Bridge Control Panel.

1. Type the following commands in the command window of the Bridge Control Panel:

   ```
   w 04 00 01 02 03 04 05 06 07 08 09 p

   r 04 x x x x x x x x x x p

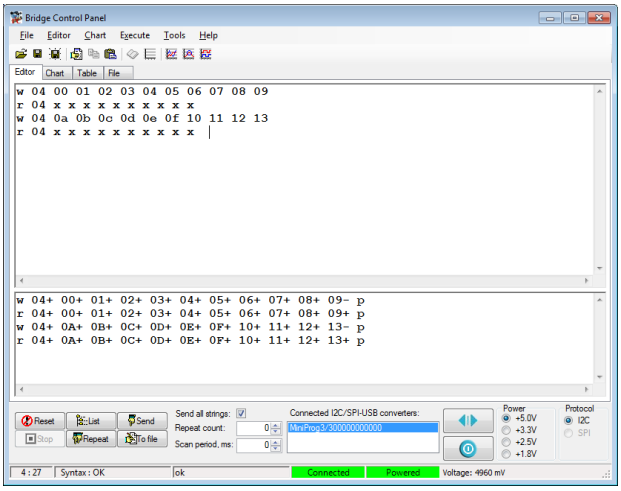
   w 04 0a 0b 0c 0d 0e 0f 10 11 12 13

   r 04 x x x x x x x x x x p
   ```

2. Select the “Send all strings” option and click the Send button to execute all of the commands in sequence.

Observe in the results window that the same values written using the write command are being read back while executing the read command.

Figure 37. Writing and Reading the I²CHW Slave

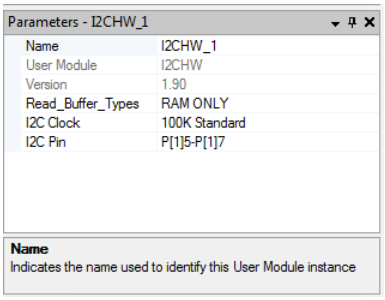


I²CHW Master Example Project

This project demonstrates how to use the I²CHW UM in master mode. Specifically, it demonstrates how to use the I²CHW UM to read and write to an external I²C EEPROM.

Figure 38 shows how to configure the I²CHW UM.

Figure 38. I²CHW Master Configuration



In the main code, the I²CHW UM is initialized. After that, it writes out 66 bytes to the EEPROM from the RAMBuffer array. The first two bytes are the subaddress where the data is written in the EEPROM. This project is tested with a 32Kb EEPROM, which has a page size of 64 bytes. When writing to smaller EEPROMs, limit the write to the page size of the particular EEPROM.

When data is written to the EEPROM, the EEPROM enters a write cycle. During this time, the slave does not generate an ACK to any I²C transaction. Any further operation can be done only when the EEPROM completes the write cycle. To detect this, the master enters a “while” loop in which it continuously sends a start and checks the ACK status.

```
while(!(I2CHW_fSendStart(0x50,
I2CHW_READ)))
{
I2CHW_SendStop();
}
```

When the slave generates an ACK, the master exits the while loop.

Next, the master reads the data that it just wrote to the EEPROM. This is done by first writing the two subaddress bytes and then reading out 64 bytes of data from the EEPROM. The master then compares the read data with the written data. When all 64 bytes match, an LED is turned on.

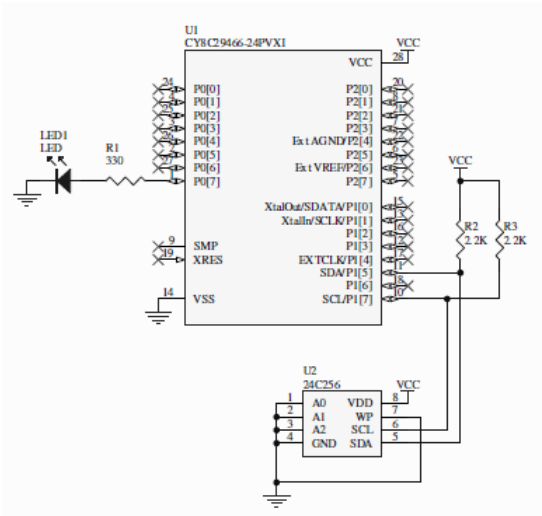
To test this project, connect P1.5 and P1.7 to the SDA and SCL of an external I²C EEPROM. Make sure that you have external pull-up resistors on those lines. Connect an LED with a series resistor to P0_7. You can test the project using a CY3210 PSoC evaluation board. (See Table 9.)

Table 9. Connections for Project Testing

PSoC 1 Pins	CY3210 Connections	Description
P1[5]	2.2-kΩ pull-up to V_{DD}	Connect to SDA of 24C256 IC
P1[7]	2.2-kΩ pull-up to V_{DD}	Connect to SCL of 24C256 IC
P0[7]	LED1	LED indicator

Figure 39 shows the setup for project testing.

Figure 39. I2CHW_Master Test Schematic

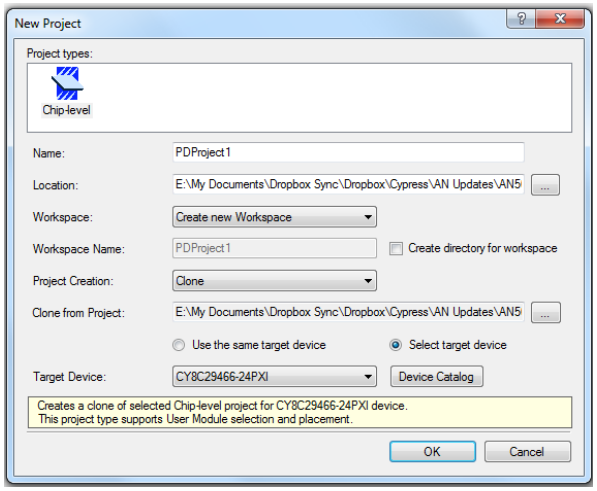


Migrating Example Project

The projects can easily be migrated to other devices using the built-in cloning feature of PSoC Designer. For example, to migrate the EzI2Cs_ADC_LED_DAC project:

1. Open PSoC Designer and create a new project.
2. Select the Clone option in the Project Creation parameter.
3. In the “Clone from Project” parameter, browse and select the EzI2Cs_ADC_LED_DAC.cmx file in the project folder of the EzI2Cs_ADC_LED_DAC project. Then choose the device that you want to run this project on by clicking the Device Catalog button.
4. After selecting the Device, click OK; PSoC Designer will migrate the project to the new device. (Refer to Figure 40.)

Figure 40. Migrate Project



Document History

Document Title: Getting Started with I²C in PSoC® 1 – AN50987

Document Number: 001-50987

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2641969	TDU	01/21/08	New Application Note
*A	3147658	TDU	01/19/2011	Updated Title to read "PSoC® 1 I²C Overview." Updated Abstract section. Updated I2C and Sleep section. Minor edits. Specified PSoC 1 parts this Application Note is applicable too.
*B	3427861	TDU	11/11/2011	Updated to New Template Modified Title Added 3 example projects Added Hardware Blocks in CY8C28xxx Updated sections I2CHW, Clock Speeds, and Clock Stretching and Interrupt Latency Added sections Glitch Filtering, I2C and Dynamic Reconfiguration, The SCL Line Gets Stuck , and Basic I2C Flow
*C	3672732	GRAA	07/11/2012	Improved the flow in EzI2Cs section Added a section explaining the EzI2Cs parameters and steps to get it working in a project Added summary and important notes section for EzI2Cs Added sections explaining in detail the hardware and firmware configuration for the I²CHW in slave, master, and multimaster modes. Added summary and important notes section for I2CHW Added section explaining how I²Cm may be incorporated in a design. Added more details about using the shadow registers when using I²Cm Added summary and important notes section for I²Cm Modified the I²CHW example project to interface to 24C256 EEPROM and updated Appendix D with the operation of the project.
*D	4078726	GRAA	07/26/2013	Minor modifications throughout document. Updated the example projects.
*E	4682397	GRAA	03/10/2015	Updated the example projects to PSoC Designer 5.4 Updated template Sunset review
*F	5688054	AESATMP7	04/10/2017	Updated Cypress Logo and Copyright.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc
Memory	cypress.com/go/memory
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/go/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners..

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

© Cypress Semiconductor Corporation, 2009-2017. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.