Networking Processor Peripherals With I2C

By (Mark Hastings, Electrical Design Engineer, Cypress Semiconductor Corp.)

Executive Summary

How do you pick a good serial interface for your application? A few important features must be evaluated first. Four of the most important issues to consider are, topology, bus bandwidth, firmware overhead, and physical layer implementation. This article explains the advantages of an I2C interface and explores how an I2C bus works. It explains how to design the interface between the master and slave, and gives examples of coding.

As microcontrollers drop in price and offer more capabilities, designers have found it more cost effective to utilize multiple small controllers in both single-board and multiboard systems. Such auxiliary processors can relieve the main processor of time consuming tasks such as scanning keyboards, display controllers, and motor control. These controllers can also be configured as a wide range of application-specific peripherals.

Recently, I was given the task of developing an interface (software/hardware) that could easily be adapted to many applications and be based on an industry standard commonly found in embedded processors. After reviewing some of the typical applications, I came up with a list of requirements to help zero in on a hardware interface.

- Common on both 32- and 8-bit processors
- Supported by many off-the-shelf peripherals
- Peripheral interface code less than 0.5 kbyte
- Low pin count
- Data bandwidth up to 10 kbytes/s
- Low RAM usage
- Support multiple peripherals on a single bus
- Easy API to use
- No external interface drive hardware required

Because of the low pin-count requirement, a serial interface was mandatory. Some of the more common serial interfaces found in today's processors include SPI, I2C, USB, and RS-232. After weighing the various pros and cons, I settled on the I2C because of its simplicity, flexibility, and availability on most low-cost controllers. Low pin count and flow control also give I2C a big advantage over SPI if higher speed isn't required.

How I2C works

I2C is a two-wire bi-directional interface consisting of a clock and data signals (SCL and SDA). A dozen devices or more may be included into a single bus without additional signals. The spec calls out three speeds of operation: 100 kbits/s, 400 kbits/s, and 3.4 Mbits/s. Most common controllers only support the 100- and 400-kbit/s modes. The spec allows for both a single master with multiple slaves or a multi-master configuration.

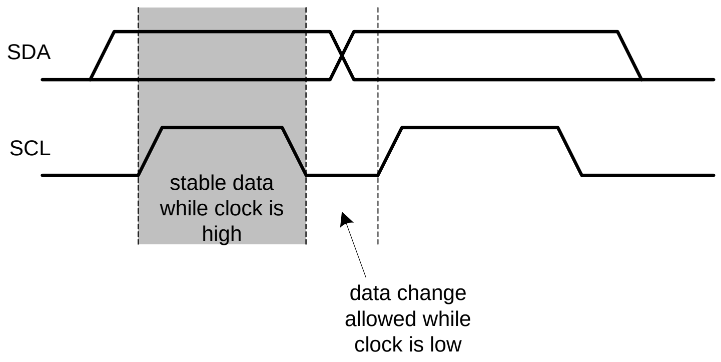
One very important attribute of I2C is that it supports flow control. If a slave can't keep up between bytes, it may halt the bus until it can catch up. This is very useful for slaves that contain minimal I2C hardware and must support part of the protocol in firmware. The I2C bus specification supports both a 7- and 10-bit address protocol. I've found that the 7-bit addressing is more than sufficient for most applications.

Before starting to write code, we need a good understanding of how the I2C bus works. The I2C bus will always have at least one master and at least one or more slaves. The master always initiates a transfer from the master to the slave. The I2C interface has only two signals, no matter how many peripherals are attached to the bus.

Both signals are open-collector with pull-up resistors of about 2.7k to VCC. The SDA signal is bi-directional and can be driven by either the master or slave. The SCL signal is driven by the master, but the slave may hold it low at the end of a data byte to hold off the bus until the slave can process the data. The master releases the SCL line after the last bit of the byte, then checks to see if the SCL signal goes high. If it doesn't, the master knows that the slave is requesting the master to hold off until the data is processed.

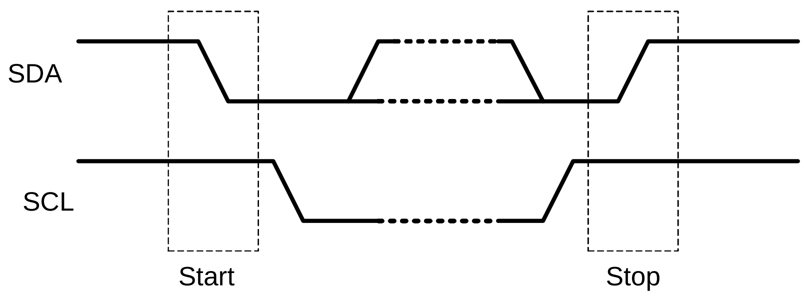
When data is being sent on the bus, data transitions occur only when SCL is low. When the SCL signal is high, the data in either direction should be stable.

Figure 1: I2C bit transfers


SDA
SCL
stable data while clock is high
data change allowed while clock is low

When the bus is idle, neither the master nor the slaves pull down the SDA and SCL. To initiate a transfer, the master drives the SDA line from high to low while SCL is high. Typically, the SDA line doesn't change state when SCL is high, except for a start or stop condition. A stop condition occurs when SCL is high and the SDA line changes from low to high.

Figure 2: I2C Start and Stop signaling



The I2C bus transfers data in 8-bit increments. Each time a byte is transferred, it must be acknowledged by the device receiving the data. All data is transferred most significant bit (MSB) first.

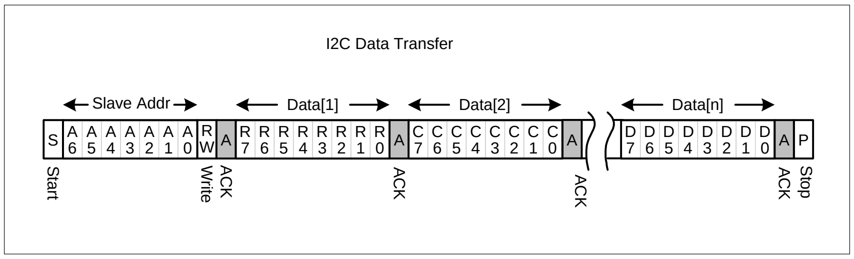
[+] Feedback

At the beginning of each transfer, a START initiates the transfer, then a 7-bit slave address, followed by an R/W flag. The I2C standard also supports a 10-bit address, but this application requires only a 7-bit address. If a slave recognizes the address, it will pull down the SDA line during the ACK state, then release it.

The R/W bit will determine the direction of the data between the master and slave. If the R/W bit is low, data will be transferred from the master to the slave. If this bit is high, data will be read from the slave by the host.

All data bytes in a single packet will be in the same direction. After each byte is transferred, it will be ACKed by either the master or slave, depending on the direction of the data flow. Figure 3 shows an example of a multibyte read or write.

Figure 3: I2C Data transfer


I2C Data Transfer
Slave Addr
Data[1]
Data[2]
Data[n]
S A6 A5 A4 A3 A2 A1 A0 RW A R7 R6 R5 R4 R3 R2 R1 R0 A C7 C6 C5 C4 C3 C2 C1 C0 A D7 D6 D5 D4 D3 D2 D1 D0 A P
Start
Write
ACK
ACK
ACK
ACK
Stop

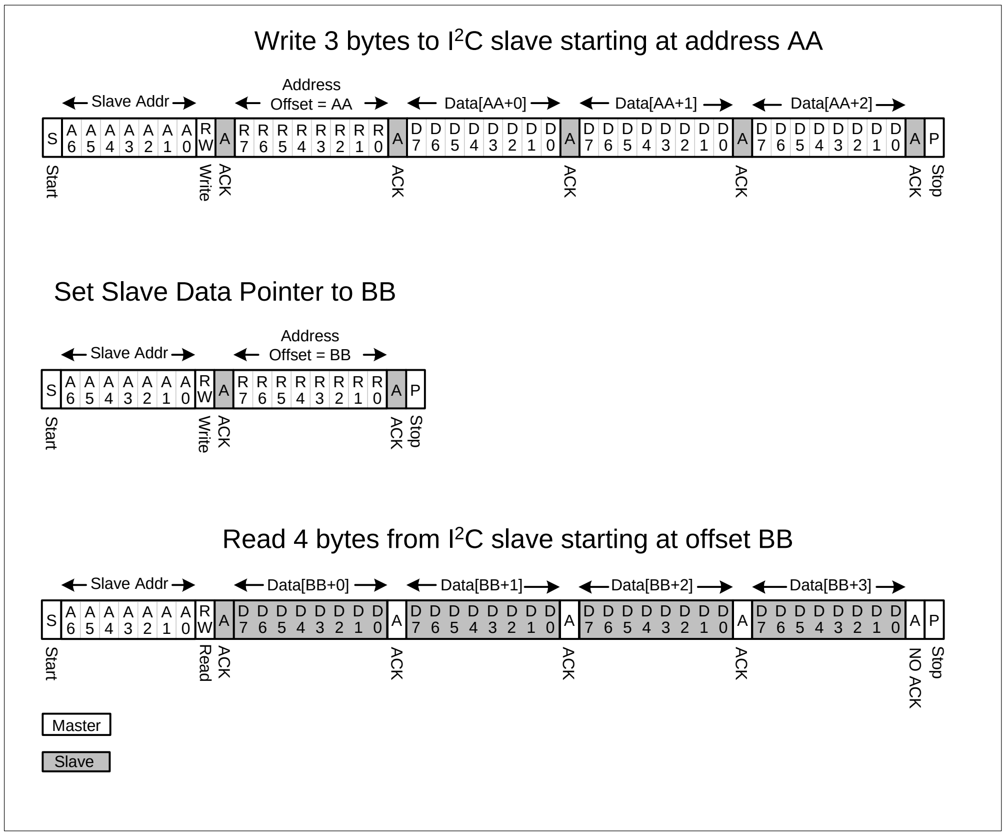
The I2C interface can be thought of as a simple stack. The lowest level of the stack is the physical layer, which consists of the electrical signaling. The next level up is the Transfer Protocol. It defines how addressing and data transfers are handled by the master and slave. The third layer from the bottom, the “Data Format” layer, is usually defined by the peripheral. It dictates how the data is stored and addressed in the peripheral. The top level “Optional Command Protocol” isn’t part of the I2C specification. This will be defined by the user. Later in this article, we’ll discuss an example of a possible implementation.

Since the Data Format layer is imposed by the peripheral, each will determine what format the data is stored. Most peripherals have one or more bytes that can be read or written. Some may have 128 or more bytes that can be accessed by the master.

To optimize data transfers, we need to impose an internal offset scheme so that if the master wants to read or write the 100th byte, it doesn’t have to read or write the preceding 99 bytes before it. Therefore, the first byte in a write sequence will always be the offset in the array of data stored in the peripheral. If more than one byte is written, the second byte will be written in the offset determined by the first byte.

The offset is sticky, meaning if a read is performed after a write sequence, the data being read will start at the offset of the previous write. If a single byte is sent in a write sequence, only the offset pointer is changed. Actual data will not be written to the peripheral.

Figure 4: Peripheral Read and Write sequence



The first sequence in Figure 4 shows three bytes written to a peripheral starting at offset AA. For example, if a peripheral has 10 byte locations where data can be written and AA is equal to 4, data will be written to the fifth, sixth, and seventh bytes in that array, since an offset of zero would have written to the first byte in the array. The second sequence in Figure 4 only writes the offset. The third sequence reads four bytes starting at offset "BB." If the third sequence is executed again, it would read the same four bytes. Until the pointer is changed, a read will start at the same offset.

Peripheral API

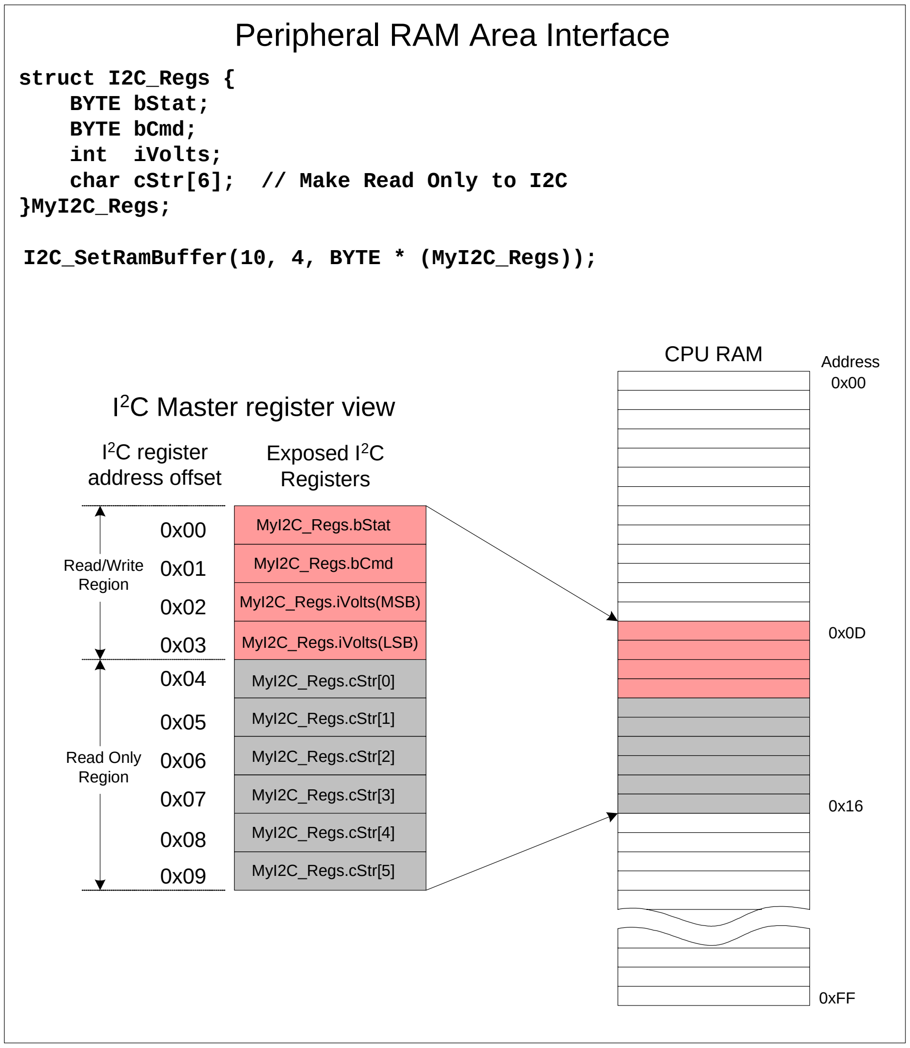
Now that the interface to the external processor is defined, we need to define the API for the slave. Often a communication interface must be integrated tightly in the peripheral application, but what if the application doesn't even have to know that the I2C interface exists beyond a couple of setup API commands? This way you could easily add the I2C interface without making significant changes to the application. For example, you could create an interface in which your peripheral CPU memory is easily accessed by the I2C master, whereby the master only has access to the area in RAM that's allowed by the slave.

The first step is to have the I2C interface run in the background as an Interrupt Service Routine (ISR). This allows memory reads and writes by the master to be transparent to the peripheral application, meaning no polling of registers, no redirecting or copying the data, and no interlacing of I2C interface code within the application.

Setup APIs are necessary to tell the I2C ISR where to put the data, as well as what boundaries or length of the data it could read and write. However, you don't want the I2C master to have access to data that it shouldn't. For example, you don't want the master to accidentally write over the main application stack. The API should tell the interface about the data's location and length. It also would be nice to have a read/write area as well as a read-only area.

Figure 5 shows how memory may be mapped between the peripheral CPU and the I2C master. The API command "I2C_ SetRamBuffer (BufferSize, R/W_Length, DataPointer)" sets the length (BufferSize), read/write length (R/W_Length), and a pointer to the data (DataPointer). The data can be placed anywhere in the peripheral CPU RAM space.

Figure 5: Interface RAM structure



Peripheral RAM Area Interface

```
struct I2C_Regs {
    BYTE bStat;
    BYTE bCmd;
    int  iVolts;
    char cStr[6];  // Make Read Only to I2C
}MyI2C_Regs;

 I2C_SetRamBuffer(10, 4, BYTE * (MyI2C_Regs));
```

The I2C master, on the other hand, sees only the memory that's exposed by the API call. Only the 10 bytes in the example can be seen, and only the first four bytes can be written. No matter where the buffer is placed, the master sees an array of data that starts at address 0x00 and goes to address 0x09.

In this example, the 10 bytes of data are defined with a structure. The application may use these variables just as it would any other local or global variables. If the structure is defined as global during compile time, most compilers will flatten it out so that it doesn't have to calculate the offset each time an element is referenced. In other words, there will be no code penalty for using such a structure.

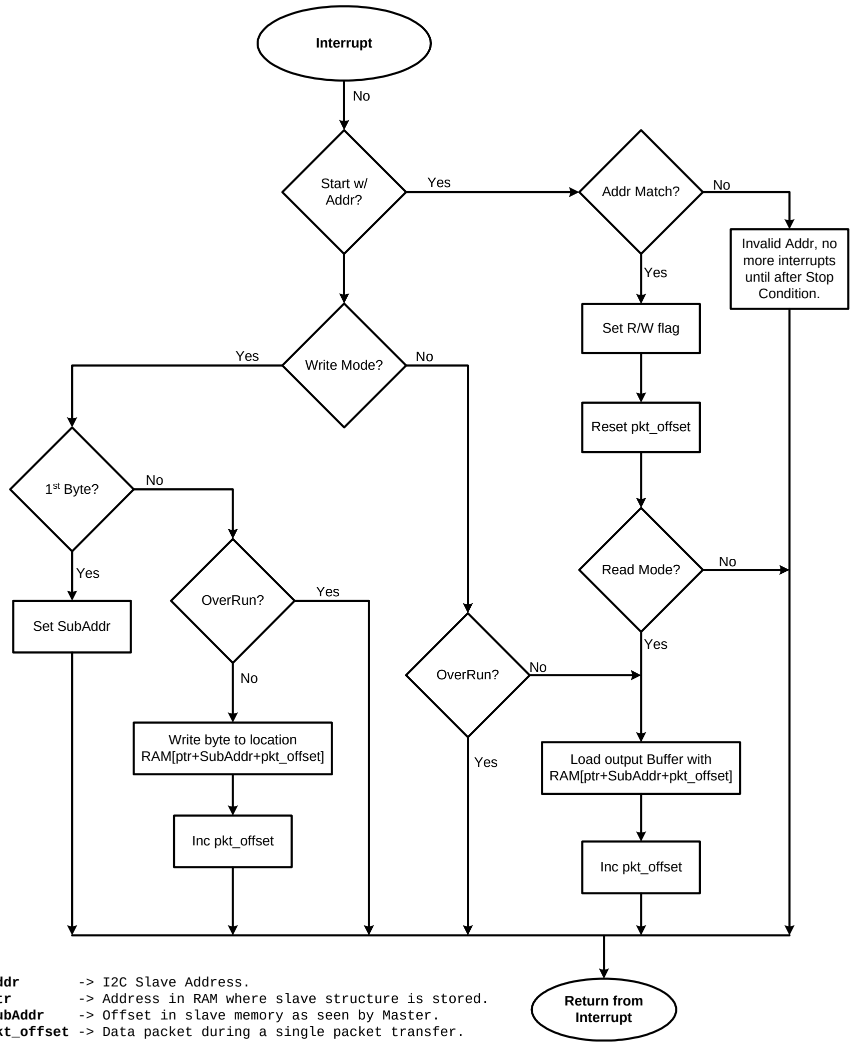
Implementation

Now that the interface between the master and slave is designed, it's time to write some code. Given the availability of I2C in a wide range of capable microprocessors, many vendors also supply I2C-friendly development tools and libraries. You'll still need to write some of your own code, but these will accelerate your development. For example, Cypress PSoC microcontrollers contain low-level I2C hardware that can be customized using PSoC Designer and application-specific EzI2C user modules.

Other than basic hardware setup commands, like I2C_Start( ) and I2C_Stop( ) that enable and disable the interface, the bulk of the code will be implemented in the Interrupt Service Routine (ISR). The low level I2C hardware understands I2C bus Start and Stop conditions and sets a status flag when the slave address and R/W bit are received. It doesn't check for an address match, but requires the firmware to perform that task.

The flow chart shows the basic firmware flow. Note that some hardware details specific to the manufacturers hardware aren't covered in the flowchart.

Figure 6: EzI2Cs program flow



```
Addr       -> I2C Slave Address.
ptr        -> Address in RAM where slave structure is stored.
SubAddr    -> Offset in slave memory as seen by Master.
pkt_offset -> Data packet during a single packet transfer.
```

For many applications in which each byte is independent of the other, this interface works well. A good example can be seen in the example application. Each of the three bytes is independent of each other.

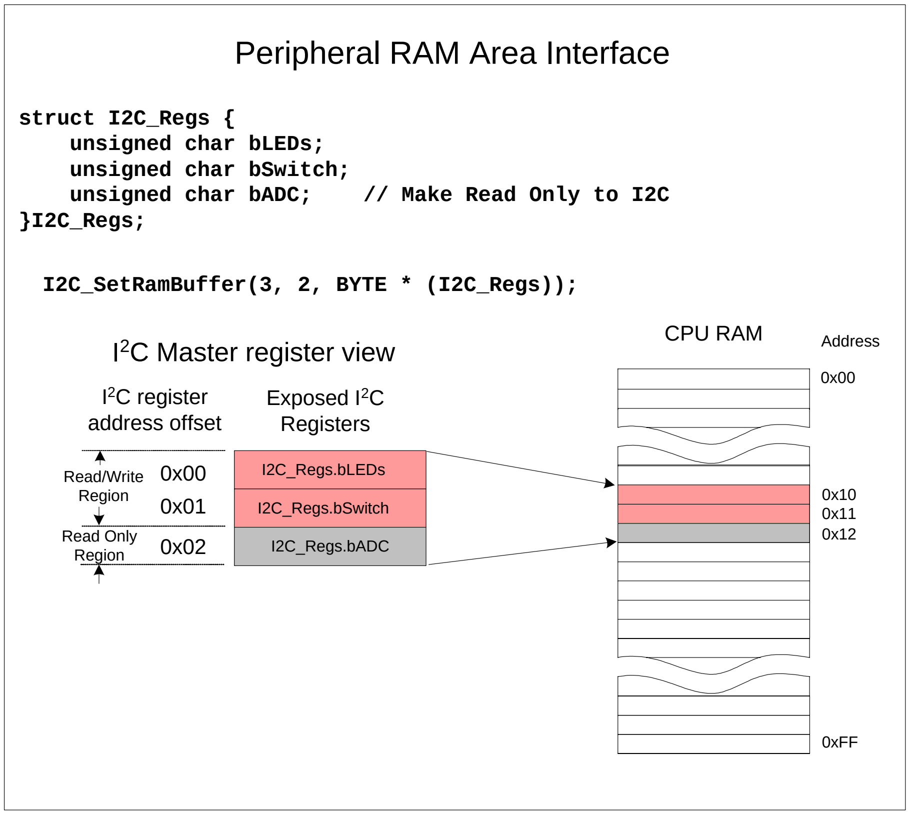
Code Listing 1

```
//----------------------------------------------------------------------------

// This program is a simple example of how to use the EzI2Cs.

//

// For this example, the 8-Pin PSoC CY8C27143-24PXI is connected as shown below;

//

//                         +---------+

//          +5 --Switch---=|P05   Vdd|=  +5

//  ADC-IN +--------------=|P03   P04|=---LED1--/\/\/---GND

//         |    I2C_SCL---=|P11   P02|=---LED2--/\/\/---GND

//         |        GND---=|GND   P10|=---I2C_SDA

//        \/               +---------+

// +5---/\/\/---GND

//

//  The EzI2Cs User Module uses the structure "I2C_Regs" as shared memory

//  between the master and the slave. The slave code (this code) only has

//  to set a pointer to the data and start the ExI2Cs User Module.  The

//  EzI2Cs is totally interrupt driven, so once started, the user code does

//  not have to deal with the I2C traffic.

//

//  This example has two inputs, a switch and a voltage input between 0 and

//  5 volts. Also, two ouputs, LED1 and LED2.

//

//----------------------------------------------------------------------------


#include <m8c.h>          // part specific constants and macros

#include "PSoCAPI.h"      // PSoC API definitions for all User Modules


struct stat{              // Structure used for I2C master to access slave

   unsigned char bLEDs;   // Output, LED bits  LED1 => 0x01,  LED2 => 0x02
   unsigned char bSwitch; // Input, Set to 1 if switch closed, else 0
   unsigned char bADC;    // 8-bit ADC result,  0x00 = 0 volts, 0xFF = 5 volts
}I2C_Regs;

#define LED1_MSK 0x01     // LED and Switch bit masks
#define LED2_MSK 0x02
#define SWITCH   0x20

void main()
{
   PRT0DR = 0x00;                  // Init port for switch pulldown
   LED1_Off();                     // Turn off LED1
   LED2_Off();                     // Turn off LED2
   PGA_SetPower(PGA_MEDPOWER);     // Turn on Amplifier
   ADC_Start(ADC_MEDPOWER);        // Turn on and setpower to ADC
   ADC_GetSamples(0);              // Set ADC to run continuously

   // Set buffer 3 bytes long, with only one byte writable
   I2C_SetRamBuffer(3, 1, (BYTE *) &I2C_Regs);
   I2C_Start();                    // Enable I2C interface

   M8C_EnableGInt;                 // Enable global interrupts

   while(1) {                      // Main Loop,  loop forever

      if(PRT0DR & SWITCH) {        // Is switch pressed?
         I2C_Regs.bSwitch = 1;     // If so, set bit.
      } else {
         I2C_Regs.bSwitch = 0;     // If not pressed, clear bit.
      }
      LED1_Switch(I2C_Regs.bLEDs & LED1_MSK);     // Set LED1 On or Off

      LED2_Switch(I2C_Regs.bLEDs & LED2_MSK);     // Set LED2 On or Off


      if(ADC_fIsDataAvailable() != 0) {           // Is ADC reading ready?

        I2C_Regs.bADC = ADC_bClearFlagGetData(); // Return ADC reading.

      }

   }

}
```

This example consists of an 8-pin PSoC CY8C27143-24PXI microcontroller, two LEDs, two current-limiting resistors for the LEDs, a pushbutton, and a potentiometer to simulate a variable voltage. Internally, the following components are instantiated: ADC, PGA, two LED drivers, and the EzI2C's User Module. The code in the listing is the only firmware the user must write for this application. The I2C interface code is handled in the ISR as discussed. The I2C master can monitor the ADC value, check the switch status, and set the state of the LEDs in this application. This interface can be reused across many projects without having to modify the interface again.

Figure 7 shows the memory representation down to the actual memory locations. The project used 1076 bytes of flash and 19 bytes of RAM. The I2C code comes to about 275 bytes, well under the 512 bytes allotted for this interface.

Figure 7: I2C Peripheral RAM structure for Example project



```
struct I2C_Regs {
    unsigned char bLEDs;
    unsigned char bSwitch;
    unsigned char bADC;    // Make Read Only to I2C
}I2C_Regs;

  I2C_SetRamBuffer(3, 2, BYTE * (I2C_Regs));
```

Some applications require handshaking between the master and slave instead of just anonymous data read and writes. Extending this interface to perform handshaking is a minor addition to the master and slave/application code. There are many ways to add this functionality.

For instance, if an ADC result is more than 8 bits, it would be possible for the host to read the MSB of one ADC conversion and the LSB of the next conversion. If the readings are very stable, you might not get into trouble. But if the result is between two values, for example 0x0200 and 0x01FF, you could accidentally get a reading of 0x02FF.

To avoid this, we can add a command byte or semaphore. Listing 2 shows a modified structure from the previous example. An additional element has been added to the structure "bCMD," and the ADC result variable was changed from an 8-bit value "bADC" to a 16-bit value "iADC."

Code Listing 2

```
struct stat{            // Structure used for I2C master to access slave

   unsigned char bCMD;    // Command byte

   unsigned char bLEDs;   // Output, LED bits  LED1 => 0x01,  LED2 => 0x02

   unsigned char bSwitch; // Input, Set to 1 if switch closed, else 0

   unsigned int  iADC;    // 16-bit ADC result,  0x00 = 0 volts, 0x0FFF = 5 volts

}I2C_Regs;
```

Now instead of the peripheral firmware blindly updating the ADC result, it waits for a command or semaphore from the master. The command could be any nonzero value of bCMD, or bCMD could be a wide range of commands that the slave/ peripheral can perform. To keep it simple, the LEDs and the switches will continue to update constantly. The iADC value, on the other hand, will only update when the bCMD value is set to a non-zero value.

The application now monitors bCMD, and when it is non-zero, it will put the latest ADC result in iADC and then set bCMD to zero. The master will then monitor bCMD and only retrieve iADC when bCMD returns to zero. In this way, the master will never get an ADC result that's out of sync. The rule for the command/ semaphore is that the master may set it, and the slave can only clear it. This is the implementation of the top layer "Optional Command Protocol" discussed previously. There's no need to make it any more complicated than that.

Code Listing 3

```
//---------------------------------------------------------------------------

// This program is a simple example of how to use the EzI2Cs.

// Added simple command layer.

//

// For this example, the 8-Pin PSoC CY8C27143-24PXI is connected as shown below;

//

//                        +---------+

//         +5 --Switch---=|P05   Vdd|=  +5

//  ADC-IN +--------------=|P03   P04|=---LED1--/\/\/---GND

//         |    I2C_SCL---=|P11   P02|=---LED2--/\/\/---GND
//       |       GND---=|GND   P10|=---I2C_SDA
//      \/              +---------+
// +5---/\/\/---GND
//
//  The EzI2Cs User Module uses the structure "I2C_Regs" as shared memory
//  between the master and the slave. The slave code (this code) only has
//  to set a pointer to the data and start the ExI2Cs User Module.  The
//  EzI2Cs is totally interrupt driven, so once started, the user code does
//  not have to deal with the I2C traffic.
//
//  This example has two inputs, a switch and a voltage input between 0 and
//  5 volts. Also, two ouputs, LED1 and LED2.
//
//-----------------------------------------------------------------------------

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

struct stat{              // Structure used for I2C master to access slave
   unsigned char bCmd;    // Command byte
   unsigned char bLEDs;   // Output, LED bits  LED1 => 0x01,  LED2 => 0x02
   unsigned char bSwitch; // Input, Set to 1 if switch closed, else 0
   unsigned int  iADC;    // 16-bit ADC result,  0x00 = 0 volts, 0xFF = 5 volts
}I2C_Regs;

#define LED1_MSK 0x01     // LED and Switch bit masks
#define LED2_MSK 0x02
#define SWITCH   0x20

void main()
{
```

```
unsigned int iResult;          // ADC result
PRT0DR = 0x00;                  // Init port for switch pulldown
LED1_Off();                    // Turn off LED1
LED2_Off();                    // Turn off LED2
PGA_SetPower(PGA_MEDPOWER);    // Turn on Amplifier
ADC_Start(ADC_MEDPOWER);       // Turn on and setpower to ADC
ADC_GetSamples(0);             // Set ADC to run continuously

// Set buffer 3 bytes long, with only one byte writable
I2C_SetRamBuffer(5, 2, (BYTE *) &I2C_Regs);
I2C_Start();                   // Enable I2C interface

M8C_EnableGInt;                 // Enable global interrupts

while(1) {                     // Main Loop,  loop forever

  if(PRT0DR & SWITCH) {        // Is switch pressed?
    I2C_Regs.bSwitch = 1;     // If so, set bit.
  } else {
    I2C_Regs.bSwitch = 0;     // If not pressed, clear bit.
  }

  LED1_Switch(I2C_Regs.bLEDs & LED1_MSK);     // Set LED1 On or Off
  LED2_Switch(I2C_Regs.bLEDs & LED2_MSK);     // Set LED2 On or Off

  if(ADC_fIsDataAvailable() != 0) {          // Is ADC reading ready?
    iResult = ADC_bClearFlagGetData();       // Return ADC reading.
    if (I2C_Regs.bCmd != 0x00) {             // Command to update reading
      I2C_Regs.iADC = iResult;              // Update reading
      I2C_Regs.bCmd = 0x00;                 // Reset command
    }
```

```
      }
   }
}
```

As can be seen, there are a few changes from the previous listing. A bCmd byte is added to the structure and the bADC unsigned char is changed to iADC. an unsigned int. A couple lines of code were added to the bottom of the listing to add the simple command protocol.

Old Code

```
       if(ADC_fIsDataAvailable() != 0) {          // Is ADC reading ready?
         I2C_Regs.bADC = ADC_bClearFlagGetData(); // Return ADC reading.
      }
```

New Code

```
      if(ADC_fIsDataAvailable() != 0) {          // Is ADC reading ready?
         iResult = ADC_bClearFlagGetData();       // Return ADC reading.
         if (I2C_Regs.bCmd != 0x00) {             // Command to update reading
            I2C_Regs.iADC = iResult;              // Update reading
            I2C_Regs.bCmd = 0x00;                 // Reset command
         }
      }
```

Conclusion

The big hurdle in developing such an interface is writing the I2C driver code in first place. The driver in this case was written in M8C assembly language. I'd rather use C, but at the time and with the tools available, it was the best way to guarantee fast and efficient code. This interface works for most I2C slave applications.

Once the driver was written, I found I could create a new custom peripheral in under an hour. This has been extremely useful in quickly implementing runtime debugging. Variables can be monitored with an I2C master while the slave code is running.

References

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
http://www.cypress.com