In the previous part of this article, we discussed how to get C#.NET applications running on the Raspberry Pi. In addition, we covered the requirements for interacting with I2C devices from C# programs using the Rpi.I2C.Net library. Finally, we tested out a sample program using this library, which displayed 60 temperature and pressure readings from the sensor at one second intervals. In this part, we will take a closer look at the some of the code in the sample program. It is assumed that the reader has downloaded & extracted the sample code from the link provided in the previous part, as well as the BMP180 sensor datasheet which is very important to have handy. To begin, open the Pi\_I2CTest solution with Visual Studio and then open the Program.cs file.

USING RPI.I2C.NET.DLL

In order to communicate with a given I2C device connected to an I2C bus, a programmatic reference to the I2C bus in question is required. To do this, the I2CBus.Open(string busName) method is used, whence busName would be “/dev/i2c-1” for the B and B+. The method returns the I2CBus reference to the chosen bus. Naturally, this I2CBus class is part of the Rpi.I2C.Net namespace, so a using directive would be in order (after adding a reference to the assembly). Thus, the code in the Main() method would be:

using System;

//other using directives

using RPi.I2C.Net;

//namespace and class declaration

static I2CBus bus;

static void Main(string[] args) {

bus = I2CBus.Open("/dev/i2c-1");

}

}

Using the I2CBus reference (called *bus* above), we will thus be able to read from and write any devices on the bus. In the code snippet above, the reference is declared with class scope so it is accessible to other methods in the program, not only the Main() method.

The BMP180 sensor, by design, uses an I2C address of 0x77 (119 in decimal). As such, any reads or writes on the bus will utilize that address.

The BMP180 sensor contains a number of registers, each containing configuration or measurement data. Configuration of the sensor is thus performed by writing the appropriate bit combinations to the appropriate registers. Similarly, temperature and pressure measurements are initiated by writing certain bits to certain registers, waiting for specified periods of time, then reading the contents of registers designated to contain the results of the measurements. To explore basic read-write functionality, we shall read the chip ID register of the sensor, which, from the datasheet, exists at location 0xD0 (208 in decimal). Also from the datasheet, a functioning BMP180 should contain the value 0x55 in that register. Thus, to ensure that there is in fact a BMP180 sensor connected to the Pi, we shall:

1. Read the value in register 0xD0
2. Compare the value read to 0x55

To read a value from any register, the BMP180 requires that the address of the desired register is first written to it, then a single byte is read back. The single byte thus read is the value in the register specified in the preceding write operation. Thus, to complete step (a) above, we will:

1. Write the value 0xD0 to the BMP180 (which selects that register for the next operation)
2. Read a single byte back from the BMP180

To perform (i), we will use the WriteByte() method of the I2CBus class, which takes two arguments: the first is the device address (integer), the second is the byte to be written (byte). From earlier, the device address is known to be 0x77. The byte to be written would be 0xD0 as explained. Thus, the code is:

bus.WriteByte(0x77, 0xD0);

To perform (ii), we will use the ReadBytes() method of the I2CBus class, which also takes two arguments: the device address (integer), and the number of bytes to read (integer). Unlike the WriteByte() method, this method returns a byte array (byte[]) containing the bytes read from the device, in the order they were read. That is, if the device returns 4, 3, 2, then the first element of the array would be 4, the second would be 3, etc.

In this case, we’re reading one byte back, so we have:

byte[] results = bus.ReadBytes(0x77, 1);

Assuming all goes well, the results array should contain a single value, which should be 0x55. We can confirm this by using an if statement, comparing results[0] with 0x55, as required by (b) above. If results[0] is indeed 0x55, then the BMP180 sensor is present and functional. If it isn’t, then perhaps there’s a wiring issue or the sensor is nonfunctional.

The complete code for the detection routine is contained in the Detect() method in the Program.cs file.

READING THE TEMPERATURE

The BMP180 contains certain calibration constants, which are required to calculate the temperature and pressure read from the sensor. These constants are designated as AC1 through AC6, B1, B2, and MB through MD. They are all sixteen-bit values, stored in register pairs starting from address 0xAA through 0xBE. Therefore, AC1’s MSB and LSB are stored in 0xAA and 0xAB respectively, and so on.

For purposes of brevity, only AC1 shall be considered. To read it in, a single byte will be read from address 0xAA as described earlier, and a single byte will be read from address 0xAB. Next, the first byte will be shifted 8 positions to the left (so it becomes the MSB), and the resultant value OR’ed with the second byte. Thus, the code will look something like below:

bus.WriteByte(0x77, 0xAA); //select the MSB of AC1

byte[] result = bus.ReadBytes(0x77, 1); //read one byte from 0xAA

bus.WriteByte(0x77, 0xAB); //select the LSB of AC1

byte[] result2 = bus.ReadBytes(0x77, 1); //read one byte from 0xAB

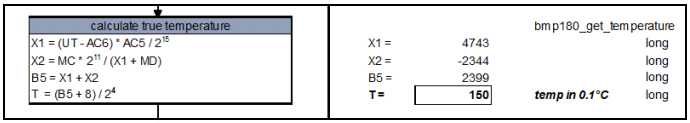
short finalValue = (short)(result[0] << 8); //shift the MSB, store it in a 16bit variable

finalValue |= (short)result2[0]; //OR the MSB with the LSB

//variable finalValue contains the calibration constant AC1

A similar process is required for the other calibration constants, thus, the code was placed in a method called ReadSignedWord() for easy reuse. Note also that constants AC4 through AC6 are unsigned, therefore the exact same code cannot be used to read them. For this reason, another method called ReadUnsignedWord() was implemented, within which the variable holding the final result is declared as ushort (unsigned short) as opposed to short (which is a signed type). Finally, a method called InitParams() was implemented, which read and stored all the calibration constants by using the ReadSignedWord() and ReadUnsignedWord() methods. The reader is strongly encouraged to read the implementation of these three methods to gain a clearer understanding.

To read the temperature from the sensor, the value 0x2E is written to register address 0xF4 to begin the measurement process. After a delay of 4.5 milliseconds, the result can be read from registers 0xF6 (MSB) and 0xF7 (LSB). The read value is unsigned, and is converted to a temperature value using the following formula (courtesy of the datasheet):



Where UT is the raw value read from the sensor.

To write multiple bytes at a time, the I2CBus instance provides a WriteBytes() method, which differs from the WriteByte() method in that the second parameter is not a single byte but a byte array (byte[]) containing the bytes to be written. As such, the code will look as below:

byte[] bytes = new byte[] { 0xF4, 0x2E }; //0xF4 (address), 0x2E (data)

bus.WriteBytes(0x77, bytes); //write address, then data

System.Threading.Thread.Sleep(7); //delay

long ut = ReadUnsignedWord(0xF6, 0xF7); //read raw value from sensor – see source file

long x1 = (long)((ut - AC6) \* AC5 / Math.Pow(2, 15)); //calculate x1

long x2 = (long)(MC \* Math.Pow(2, 11) / (x1 + MD)); //calculate x2

long b5 = x1 + x2; //calculate b5

long temp = (long)((b5 + 8) / Math.Pow(2, 4)); //calculate temp

temp = temp \* 0.1;

//temp now contains the temperature in centigrade

The code above is used in the ReadTemperature() method. The user is urged to explore the datasheet and the code for reading the pressure.

CONCLUSION

The Rpi.I2C.Net library provides a well-designed, low-level interface to the I2C bus. This makes it possible to write drivers for particular I2C devices with maximal flexibility. A basic BMP180 driver is implemented in the BMP180.cs file.

Note also that error checking/exception handling code has been omitted both above and in the driver code. In practice, all calls to the I2CBus methods should be wrapped in try-catch blocks. All programs using Rpi.I2C.Net should be run with root privileges, else, they will be unable to access the I2C bus due to permission issues. Finally, the Dispose() method should be called on the I2CBus instance once it is no longer required. This explicitly ensures that other programs can access it after your program exits.

Happy hacking!