C# was unleashed onto an unsuspecting world sometime in 2002, alongside Microsoft’s dotNet framework, which was intended to be a secure runtime environment in the same way Java was, using its JVM. C# itself shares many similarities with Java, both languages being C/C++ derived.

Although the initial releases of the dotNet framework ran exclusively on Microsoft Windows OSes, the dotNet framework specification was released in 2000, which allowed anyone code an implementation on any other operating system. Fast forward a few years later, and Novell came up with the Mono project, which aimed to provide a dotNet-compatible runtime environment for Linux (initially) and MacOS. Today, Mono is maintained by Xamarin Ltd and supported on Android and iOS as well, as the Xamarin.Android and Xamarin.iOS frameworks respectively. The current version of the Mono framework available from the Debian Wheezy repos is 3.2.8, which supports up to dotNet framework version 4.5.

This article aims to show you how to get up and running with C# on the Pi, with a practical view to interfacing with a BMP180 pressure and temperature sensor (a breakout board from Sparkfun Electronics is used, available at [8]). It is assumed that Microsoft Visual Studio 2012/2013 (any edition) is used, and the Pi is running the Raspbian Wheezy distribution.

GETTING MONO INSTALLED ON THE RASPBERRY PI

To be able to run C# (and essentially, any app written in a dotNet language like VB.NET, F#.NET, etc) apps on the Pi, you’ll need to install the Mono runtime. For the most basic purposes, you could install the base mono-runtime package, but for purposes of completeness I’d recommend installing the mono-complete package instead, which bundles development tools and a whole lot more dotNet libraries. This is as easy as typing the following commands in your Pi’s terminal:

*sudo apt-get update*

*sudo apt-get install mono-complete*

This will take a while, so I’d suggest taking a long bath or watching an episode of How I Met Your Mother. Hopefully when you return, it’d be complete. To ensure you have the latest version of Mono available (3.2.8), type the following command in the terminal:

*mono --version*

This should return something that looks like the text below:

*Mono JIT compiler version 3.2.8 (Debian 3.2.8+dfsg-4+rpi1)*

*Copyright (C) 2002-2014 Novell, Inc, Xamarin Inc and Contributors. www.mono-project.com*

*TLS: \_\_thread*

*SIGSEGV: normal*

*Notifications: epoll*

*Architecture: armel,vfp+hard*

*Disabled: none*

*Misc: softdebug*

*LLVM: supported, not enabled.*

*GC: sgen*

To test a C#/dotNet app, you could simply copy it over (it would be an .exe file on Windows) perhaps via WinSCP or your preferred method. Navigate to the directory where you placed the file, and type the following command:

*mono <appname>.exe*

This will invoke the Mono runtime with your app’s name as an argument, which would cause Mono to load and run your program. For a console app, you’d see any Console output and be able to interact with it directly within the terminal. For a Windows Forms app, ensure that you’re in the X environment (startx, anyone?) beforehand. You can then double-click the executable to run it.

Note that if your program relies on any external assemblies (DLL files which are not part of the default dotNet set), you’ll need to copy them across to the same directory the app itself is placed before running the command. If you don’t, you’ll most likely end up with a nasty error about assemblies not being found. Another caveat is that apps running on Mono must not rely on Windows-only technology (such as DirectX, Win32 or similar) for obvious reasons.

GETTING READY FOR I2C COMMUNICATION

For brevity, I won’t detail the working principle of the I2C bus or how to get set up for I2C here. An excellent guide can be found at [1]. In a nutshell, you should’ve gone through the following steps:

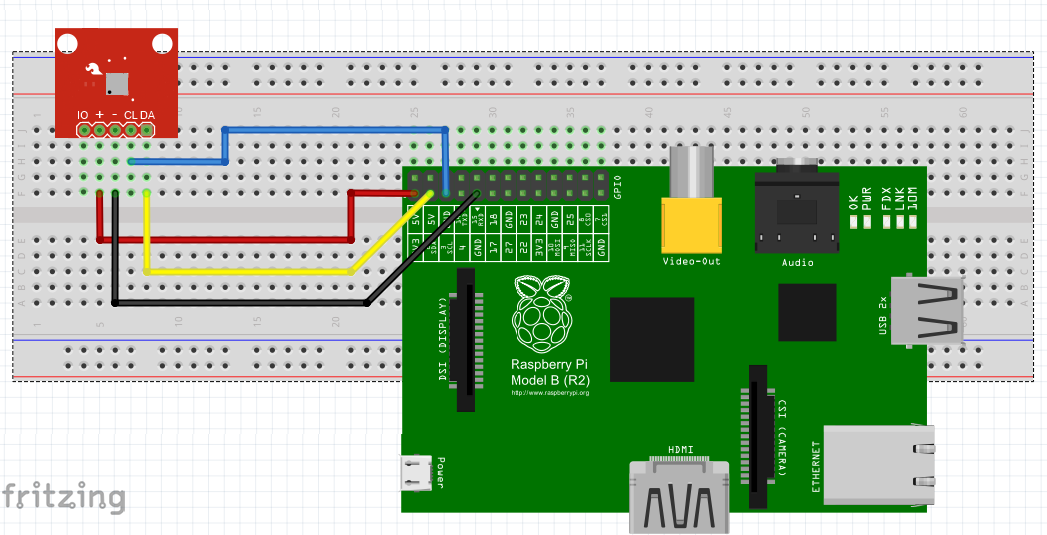
1. Removing the i2c-bcm2708 kernel module from the blacklist
2. Adding “i2c-dev” to the end of the /etc/modules file so it is loaded on boot (optional)

The last step is recommended, especially for deployment scenarios. If you choose not to do so, be sure to type “sudo modprobe i2c-dev” at the command-line BEFORE attempting ANY I2C communications. The command will load the I2C driver and create the necessary /dev entry, without which I2C communications would be impossible.

Next, we’ll need to install some diagnostic tools to make sure we’re good to go. They come in a package called i2c-tools, so to install we type the following command at the terminal:

*sudo apt-get install i2c-tools*

This will install the i2c-tools package, which contains a program (amongst others) called i2cdetect, which is used for…detecting I2C devices. Next, wire up the breakout as described by the schematic below. To be on the safe side, shut the Pi down beforehand, and pay close attention to the power connections.

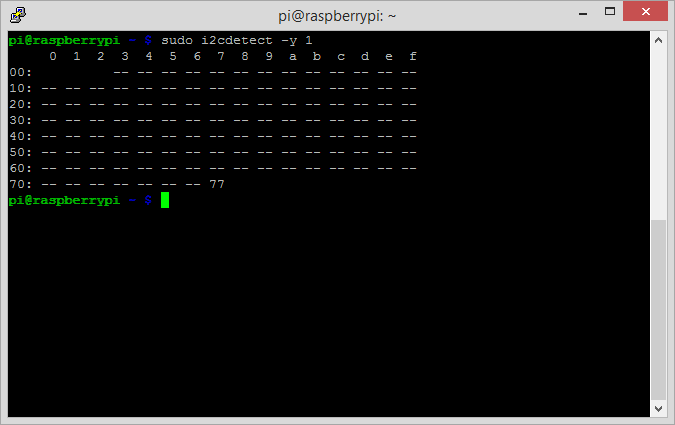


Now, reboot the Pi. Type the following at the console prompt:

*sudo i2cdetect –y 1*

This will run the i2cdetect program, which will show us all devices on the I2C bus. The –y part is to override a yes/no prompt by providing a default ‘yes’ answer, and 1 specifies the I2C bus to be scanned. For the Model A, that would be 0 as the I2C bus which is broken out is bus 0 not bus 1 (for the Model B/B+).

The following should then appear:



This tells us that a device with ID 0x77 has been detected, which is the BMP180 sensor. This address is hardcoded at the factory. With that, we’re ready to move onto the next step.

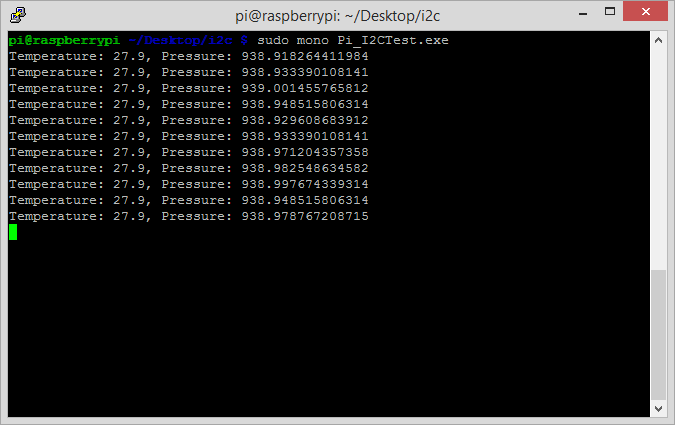
USING RPI.I2C.NET

We’ll need a copy of the Rpi.I2C.Net.dll file, which contains classes written by Dmitry Mshmelev. These classes allow I2C communication from within a C# (dotNet, actually) app. In addition, a native shared library called libnativei2c.so is needed to get the Rpi.I2C.Net.dll file to work. Dmitry provided excellent instructions on how to compile this from source at [2], as well as a copy of the libnativei2c.so file targeting Arch Linux (Alarmpi). Pre-compiled versions of the Rpi.I2C.Net.dll and libnativei2c.so files are provided at [3] and [4] respectively, the latter targeting Raspbian. If you need to make any changes to the Rpi.I2C.Net code, then [5] will be of some use (for setting up Visual Studio to target Mono, which is needed to compile the assembly).

To hit the ground running, I’ve provided a simple test binary at [6]. Place the files obtained from [3], [4] and [6] in the same folder on the Pi. Next, run the following command at the terminal (don’t forget the sudo):

*sudo mono bmp180test.exe*

The output from the program should be something like this:



It will take sixty readings, print “Done” and then close.

For your own subsequent programs, place the files from [3] and [4] in the same folder as your custom app. That is all that is required to use Rpi.I2C.Net, assuming you’re on Raspbian. For any other distribution, the libnativei2c.so file might need to be recompiled on the target OS to be on the safe side. See [2] for further details.

If you get results similar to those shown in the picture above, congratulations! It means you’re ready to move onto the next step, which is writing custom code to interface with the sensor. We’ll take a deeper look into the code of the sample program (all the Visual Studio code and project files can be downloaded from [7] and are required for the next part of this series) and exact steps regarding how to develop this custom code with Visual Studio.

**Links**

[1] <https://learn.adafruit.com/adafruits-raspberry-pi-lesson-4-gpio-setup/configuring-i2c>

[2] <https://github.com/mshmelev/RPi.I2C.Net>

[3] <https://www.dropbox.com/s/8u5q8cm2e3wxj31/RPi.I2C.Net.dll?dl=0>

[4] <https://www.dropbox.com/s/1uep265if4mbvpf/libnativei2c.so?dl=0>

[5] <http://erictummers.wordpress.com/2012/01/25/target-mono-from-visual-studio/>

[6] <https://www.dropbox.com/s/n06ba8vsyopwidi/Pi_I2CTest.exe?dl=0>

[7] <https://www.dropbox.com/s/5hfkzvwcc7519zl/Pi_I2CTest.zip?dl=0>

[8] <https://www.sparkfun.com/products/11824>