How to Become an Embedded Developer in Minutes

Jump into programming the next big thing using embedded Java.

ANGELA CAICEDO
BIO

In my 10 years as a Java evangelist, I've never been more excited about a Java release. Not only is Java 8 a great release with cool new features in the language itself, but it also provides great things on the embedded side, great optimizations, and really small specifications. If you are a Java developer and ready to jump with me into the new wave of machine-to-machine technology—better said, into programming the next big thing—let's get started with the Internet of Things (IoT).

Before you get started with embedded programming, you need to understand exactly what you are planning to build and where you are planning to run your application. This is critical, because there are different flavors of embedded Java that will fit your needs perfectly.

If you are looking to build applications similar to the ones you run on your desktop, or if you are looking for great UIs, you need to take a look at Oracle Java SE Embedded, which is derived from Java SE. It supports the same platforms and functionality as Java SE. Additionally, it provides specific features and supports additional platforms, it has small-footprint Java runtime environments (JREs), it supports headless configurations, and it has memory optimizations.

> **SHARED ROOTS**
> **Oracle Java SE Embedded supports the same platforms and functionality as Java SE.**

On the other hand, if you are looking for an easy way to connect peripherals, such as switches, sensors, motors, and the like, Oracle Java ME Embedded is your best bet. It has the Device Access API, which defines APIs for some of the most common peripheral devices that can be found in embedded platforms: general-purpose input/output (GPIO), inter-integrated circuit (I^2C) bus, serial peripheral interface (SPI) bus, analog-to-digital converter (ADC), digital-to-analog converter (DAC), universal asynchronous receiver/transmitter (UART), memory-mapped input/output (MMIO), AT command devices, watchdog timers, pulse counters, pulse-width modulation (PWM) generators, and generic devices.

The Device Access API is not present in Oracle Java SE Embedded (at least not yet), so if you still want to use Oracle Java SE Embedded and also work with peripherals, you have to rely on external APIs, such as Pi4J.

In term of devices, embedded Java covers an extremely wide range from conventional Java SE desktop and server platforms to the STMicroelectronics STM32F4DISCOVERY board, Raspberry Pi, and Windows. For this article, I'm going to use the Raspberry Pi, not only because it's a very powerful credit-card-size single-board computer, but also because it's very affordable. The latest model costs only US$35.

Getting Your Raspberry Pi Ready

In order to boot, the Raspberry Pi requires a Linux image on a Secure Digital

Java 8 Is Here

(SD) memory card. There is no hard drive for the computer. Instead, an SD card stores the Linux image that the computer runs when it is powered on. This SD memory card also acts as the storage for other applications loaded onto the card.

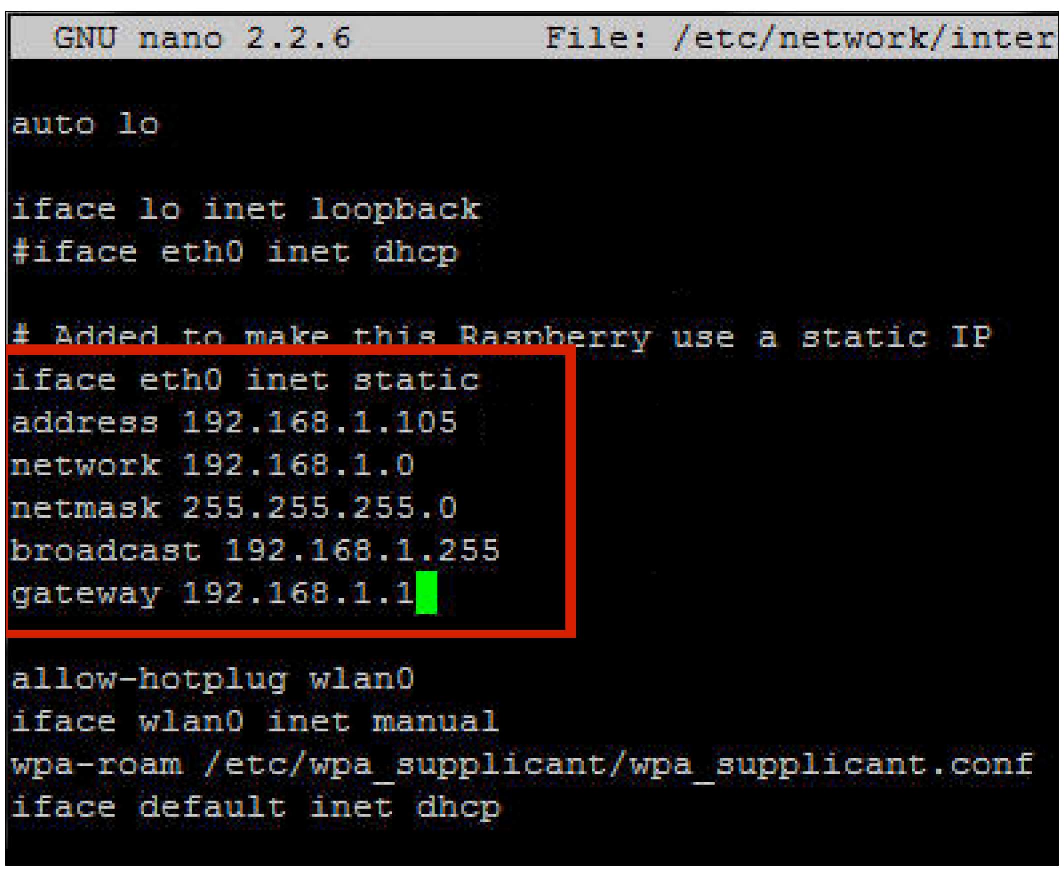
To configure your SD card, perform the following steps:

1. Format your SD card.
2. Download Raspbian, a free operating system based on Debian that is optimized specifically for the Raspberry Pi hardware.
3. Create a bootable image. You can use applications such as Win32 Disk Imager to easily create your image.

Once you have your SD card ready, you can turn on your Raspberry Pi. The first time you boot your Raspberry Pi, it will take you to the Raspberry Pi Software Configuration Tool to perform some basic configuration. Here are the additional tasks you should perform:

1. Ensure that all the SD card storage is available to the operating system by selecting the Expand Filesystem option.
2. Set the language and regional setting to match your location by selecting the Internationalisation option.
3. Set up the Raspberry Pi as a headless (that is, without a monitor attached) embedded device by allowing access via Secure Shell (SSH). To configure this, select the Advanced option from the main menu.
4. Ensure that the Raspberry Pi always has the same IP address by setting a static IP address. Although this is not required, I find it very useful when using the Raspberry Pi headless. To set up the static IP address, edit the /etc/network/interfaces file. An example of this file is shown in **Figure 1**.

Now you are ready to connect to the Raspberry Pi. One option is to use PuTTY. An example of how to connect is shown in **Figure 2**.


```
  GNU nano 2.2.6              File: /etc/network/inter

auto lo

iface lo inet loopback
#iface eth0 inet dhcp

# Added to make this Raspberry use a static IP
iface eth0 inet static
address 192.168.1.105
network 192.168.1.0
netmask 255.255.255.0
broadcast 192.168.1.255
gateway 192.168.1.1

allow-hotplug wlan0
iface wlan0 inet manual
wpa-roam /etc/wpa_supplicant/wpa_supplicant.conf
iface default inet dhcp
```

Figure 1

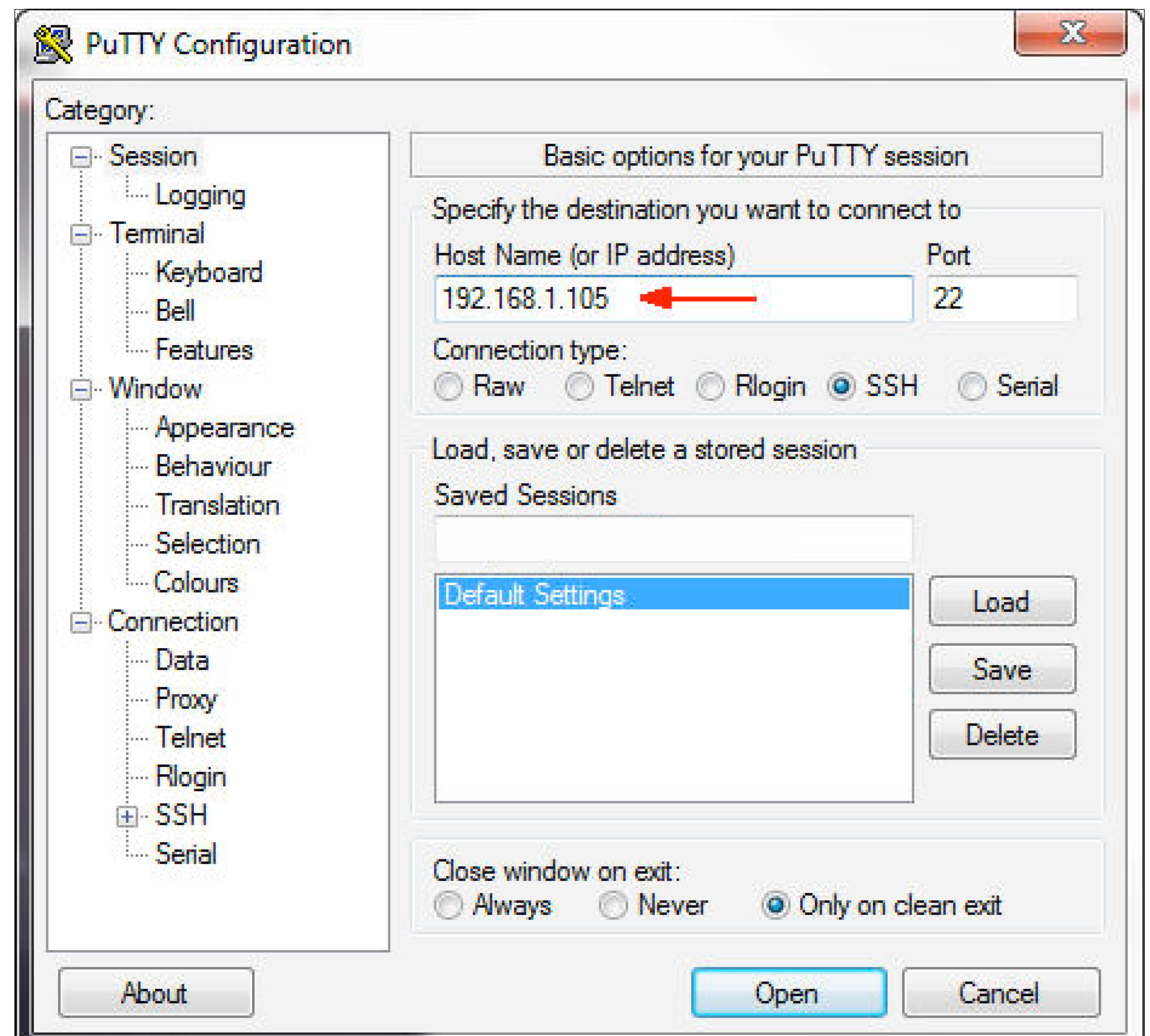


Figure 2

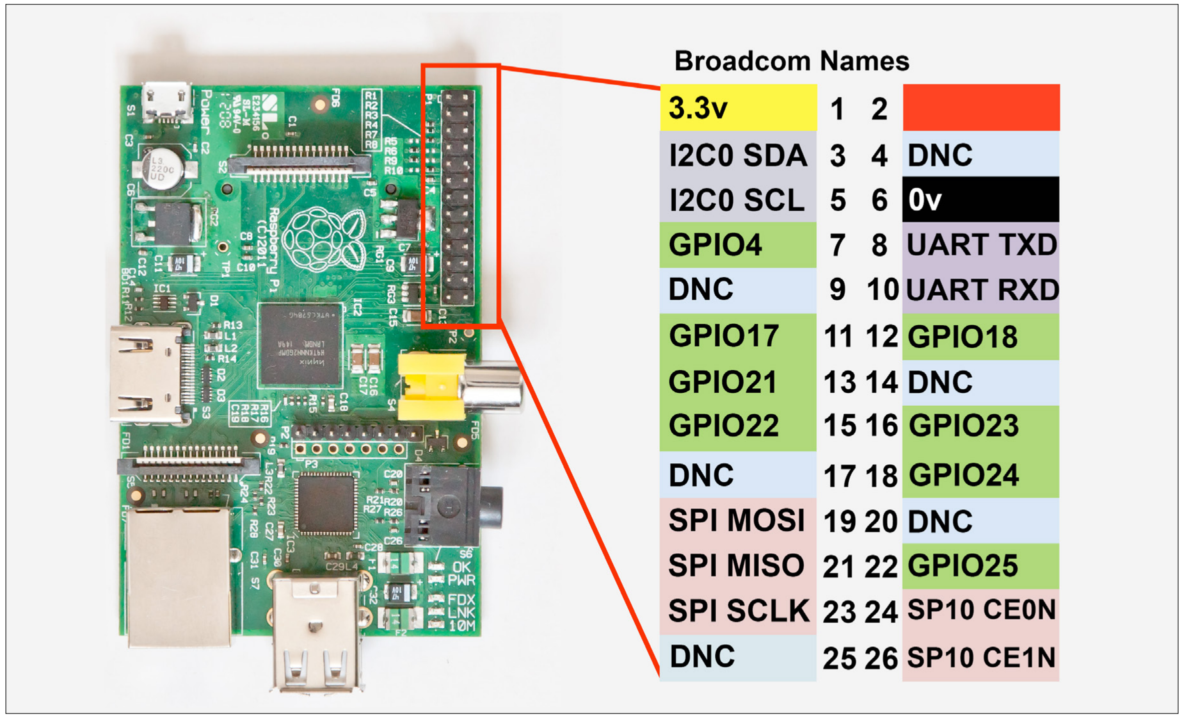


Figure 3

LISTING 1

```
public class Midlet extends MIDlet {

  @Override
  public void startApp() {
    System.out.println("Started...");
  }

  @Override
  public void destroyApp(boolean unconditional) {
    System.out.println("Destroyed...");
  }
}
```

Download all listings in this issue as text

Installing Embedded Java on the Raspberry Pi

Now, this is where you decide the kind of application you want to run on your device. I personally love having fun with peripherals, so in this article I'm going to use Oracle Java ME Embedded, so I can leverage the Device Access API. But remember, you can also run Oracle Java SE Embedded on your Raspberry Pi.

Installing the Oracle Java ME Embedded binary on the Raspberry Pi is very simple. Just use FTP to transfer the Raspberry Pi distribution zip file from your desktop to the Raspberry Pi over the SSH connection. Then unzip the file into a new directory, and you're done.

Putting Things Together

One great option for creating your embedded application is using the NetBeans IDE with the Java ME SDK. Combining these two allows you to test your application, even before you run it on your device, by using an emulator. You will be able to automatically transfer your code and execute it on your Raspberry Pi, and you can even debug it on the fly. All you need to ensure is that the Java ME SDK is part of the Java platforms on your IDE. You need to enable the SDK in the NetBeans IDE by selecting Tools->Java Platforms, clicking Add Platform, and then specifying the directory that contains the SDK.

In order to remotely manage your embedded applications on your Raspberry Pi, you need to have the Application Management System (AMS) running. Through SSH, simply execute the following command:

```
pi@raspberrypi sudo
javame8ea/bin/usertest.sh
```

Your First Embedded App

Oracle Java ME Embedded applications look exactly like other Java ME applications. **Listing 1** shows the simplest example you can have.

Your application must inherit from the MIDlet class, and it should override two lifecycle methods: startApp and destroyApp. These two methods will be invoked when the application gets started and just before it gets destroyed. The code in **Listing 1** just prints a text message on the device console.

Turning on the Lights!

Now let's do something a bit more interesting, such as turning an LED on and off by pressing a switch. First, let's have a look at the GPIO pins on the Raspberry Pi (see **Figure 3**).

The GPIO connector has a number of different types of connections on it:

- GPIO pins
- I^2C pins
- SPI pins
- Serial Rx and Tx pins

This means that we have several options for where to connect our

LED and switch; any of the GPIO pins will work fine. Just make a note of the pin number and ID for each device, because you will need this information to reference each device from your code.

Let's do some basic soldering and create the circuit shown in **Figure 4**. Note that we are connecting the LED to pin 16 (GPIO 23) and the switch to pin 11 (GPIO 17). A couple of resistors are added to make sure the voltage levels are within the required range.

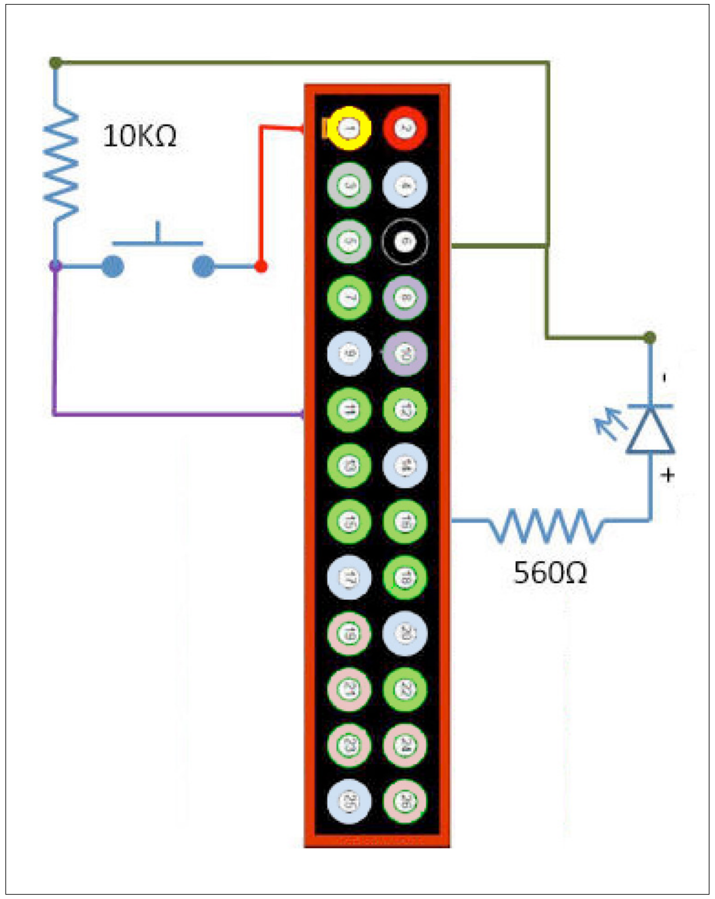


Figure 4

Now let's have a look at the program. In the Device Access API, there is a class called PeripheralManager that allows you to connect to any peripheral (regardless of what it is) by using the peripheral ID, which simplifies your coding a lot. For example, to connect to your LED, simply use the static method open, and provide the pin ID 23, as shown in **Listing 2**. Done!

To change the value of the LED (to turn it on and off), use the setValue method with the desired value:

```
// Turn the LED on
led1.setValue(true);
```

It really can't get any easier.

To connect the switch, we could potentially use the same open method on PeripheralManager, but because we would like to set some configuration information, we are going to use a slightly different approach. First, we create a GPIOPinConfig object (see **Listing 3**), which contains information such as the following:

- Device name
- Pin number
- Direction: input, output, or both
- Mode: pull-up, pull-down, push-pull, or open-drain

LISTING 2 / LISTING 3 / LISTING 4 / LISTING 5 / LISTING 6

```
private static final int LED1_ID = 23;
...
GPIOPin led1 = (GPIOPin) PeripheralManager.open(LED_ID);
```

Download all listings in this issue as text

- Trigger: none, falling-edge, rising-edge, both edges, high-level, low-level, or both levels
- Initial value

Then, we call the open method using this configuration object, as shown in **Listing 4**.

We can also add listeners to the pins, so we will get notified every time a pin value changes. In our case, we want to be notified when the switch value changes, so we can set the LED value accordingly:

```
button1.setInputListener(this);
```

Then implement the valueChanged method that will be called when events occur, as shown in **Listing 5**.

It's also important that you close the pins when you are done, and also make sure you turn your LED off (see **Listing 6**).

The whole class can be found here.

Now, all we are missing is the main MIDlet that invokes our code. The startApp method shown in **Listing 7** will create an object to control our two GPIO devices (LED and switch) and listen to our inputs, and the stopApp method will make sure everything is closed (stopped) properly.

Sensing Your Environment

LEDs and switches are nice, but what is really interesting is when we start sensing our surrounding environment. In the following example, I want to show how to get started with sensors that use the I^2C protocol.

I^2C devices are perhaps the most widely available devices, and their biggest advantage is the simplicity of their design. I^2C devices use only two bidirectional open-drain lines: Serial Data Line (SDA) and Serial Clock Line (SCL).

Devices on the bus will have a specific address. A master controller sets up the communication with an individual component on the bus by sending out a start request on the SDA line, followed by the address of the device. If the device with that address is ready, it responds with an acknowledge request. Data is then sent on the SDA line, using the SCL line to control the timing of each bit of data.

NOW IS THE TIME

If you are a Java developer and you are ready to jump into the new wave of machine-to-machine technology— **let's get started with the Internet of Things (IoT).**

When the communication with a single device is complete, the master sends a stop request. This simple protocol makes it possible to have multiple I^2C devices on a single two-line bus.

Getting I^2C Working on the Raspberry Pi

If you look at the Raspberry Pi pins again (see **Figure 3**), you will see that there are two pins for I^2C: pin 3 is the data bus (SDA) and pin 5 is the clock (SCL). I^2C is not enabled by default, so there are a few steps we need to follow in order to make it available to our application.

First, use a terminal to connect to your Raspberry Pi, and then add the following lines to the /etc/modules file:

```
i2c-bcm2708
i2c-dev
```

It's very useful to have the i2c-tools package installed, so the tools will be handy for detecting devices and making sure everything works properly. You can install the package using the following commands:

```
sudo apt-get install
python-smbus
sudo apt-get install i2c-tools
```

Lastly, there is a blacklist file called /etc/modprobe.d/raspi-blacklist.conf; by default SPI and I^2C are part of this blacklist. What this means is that unless we remove or comment out these lines, I^2C and SPI won't work on your Raspberry Pi. Edit the file and

LISTING 7

```
public class Midlet extends MIDlet{

 private MyFirstGPIO gpioTest;

 @Override
 public void startApp() {
  gpioTest = new MyFirstGPIO();
  try {
   gpioTest.start();
  } catch (PeripheralTypeNotSupportedException |
       PeripheralNotFoundException|
       PeripheralConfigInvalidException |
       PeripheralExistsException ex) {
    System.out.println("GPIO error:"+ex.getMessage());
  } catch (IOException ex) {
    System.out.println("IOException: " + ex);
  }
 }

 @Override
 public void destroyApp(boolean unconditional) {
  try {
    gpioTest.stop();
  } catch (IOException ex) {
    System.out.println("IOException: " + ex);
   }
  }
 }
}
```

Download all listings in this issue as text remove the following lines:

```
blacklist spi-bcm2708
blacklist i2c-bcm2708
```

Restart the Raspberry Pi to make sure all the changes are applied.

Adding Sensors

The BMP180 board from Bosch Sensortec is a low-cost sensing solution for measuring barometric pressure and temperature. Because pressure changes with altitude, you can also use it as an altimeter. It uses the I^{2}C protocol and a voltage in the range of 3V to 5V, which is perfect for connecting to our Raspberry Pi.

Let's go back to soldering to connect the BMP180 board to your Raspberry Pi using the diagram shown in **Figure 5**. Normally, when you use an I^{2}C device, a pull-up resistor is required for the SDA and SCL lines. Fortunately, the Raspberry Pi provides pull-up resistors, so a simple connection is all you need.

Once we connect the board to the Raspberry Pi, we can check whether we can see an I^{2}C device. On your Raspberry Pi run the following command:

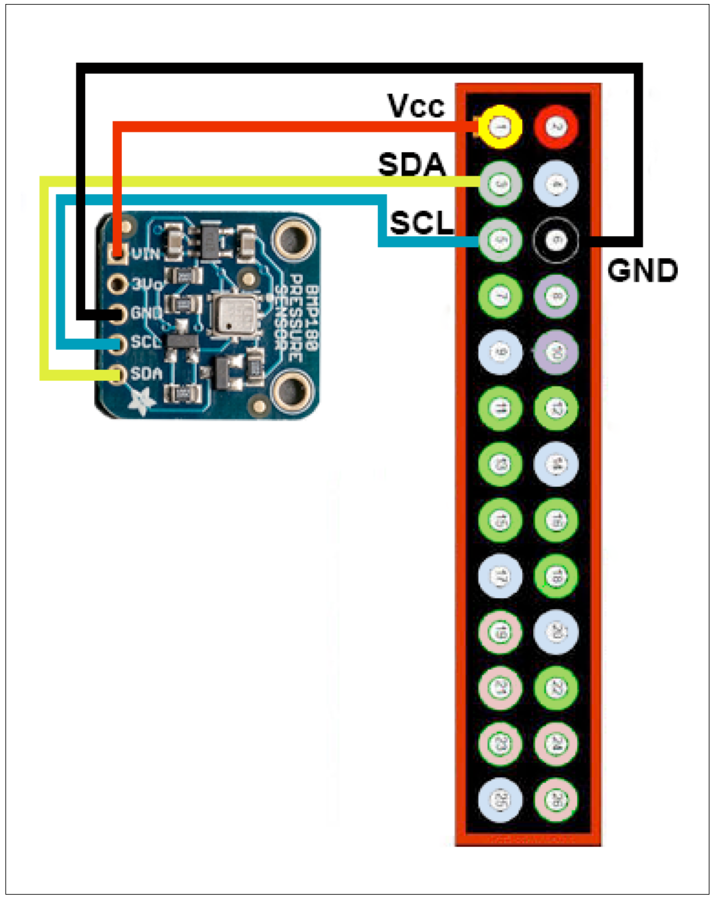


Figure 5

```
sudo i2cdetect -y 1
```

You should be able to see your device in the table. **Figure 6** shows two I^{2}C devices: one at address 40 and one at address 70.

Using an I^{2}C Device to Get the Temperature

There are a few things you need to know before you programmatically connect to an I^{2}C device:

- What is the device's address? I^{2}C uses 7 bits for a device address, and the Raspberry Pi uses I^{2}C bus 1.

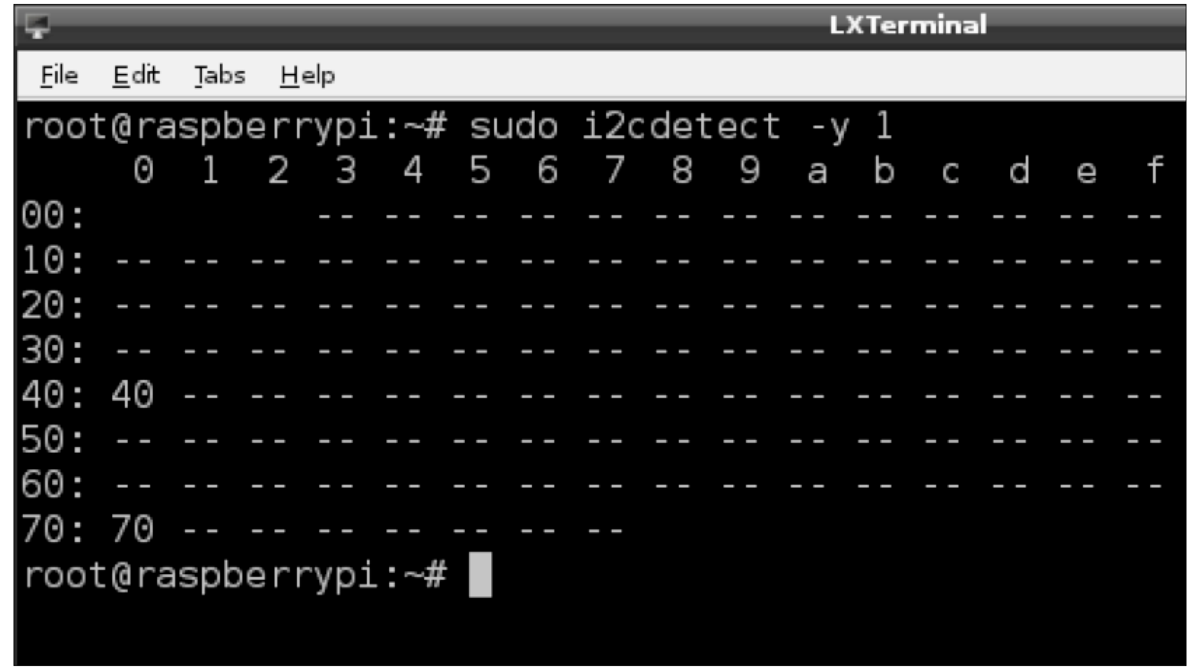


Figure 6

- What is the register's address? In our case, we are going to read the temperature value, and this register is located at address 0xF6. (This is specific to the BMP180 board.)
- Do you need to configure any control registers to start sensing? Some devices are in a sleep mode by default, which means they won't be sensing any data until you wake them up. The control register for our device is address 0xF4. (This is specific to the BMP180 board.)
- What is the clock frequency for your device? In our case, the BMP180 board uses 3.4 MHz.

Listing 8 defines the values for the BMP180 as static variables to be used later in the code.

Once again, the way we connect programmatically to the device is using the PeripheralManager's static method open. In this case, we will provide an I2CDeviceConfig object that is specific to an I^{2}C device (see **Listing 9**). The I2CDeviceConfig object allows us to specify the device's bus, address, address size (in bits), and clock speed.

To take a temperature reading, we need to follow three steps:

1. Read calibration data from the device, as shown in **Listings 10a** and **10b**. This is specific to the BMP180 board, and you might not need to perform this step when using other temperature sensors.
2. Write to a control register on the device to initiate a tem-

LISTING 8 / LISTING 9 / LISTING 10a / LISTING 10b / LISTING 11 / LISTING 12

```
//Raspberry Pi's I2C bus
private static final int i2cBus = 1;
// Device address
private static final int address = 0x77;
// 3.4MHz Max clock
private static final int serialClock = 3400000;
// Device address size in bits
private static final int addressSizeBits = 7;
...

// Temperature Control Register Data
private static final byte controlRegister = (byte) 0xF4;
// Temperature read address
private static final byte tempAddr = (byte) 0xF6;
// Read temperature command
private static final byte getTempCmd = (byte) 0x2E;

...
// Device object
private I2CDevice bmp180;
```

Download all listings in this issue as text

perature measurement (see **Listing 11**).

3. Read the uncompensated temperature as a two-byte word, and use the calibration constants to determine the true temperature, as shown in **Listing 12**. (Once again, this is specific to this sensor.)

Finally, the temperature in Celsius will be stored in the celsius variable. You can find the entire program here.

As an exercise, you can extend the program to read the pressure, the altitude, or both.

Conclusion

This article took you through the steps required to start creating embedded Java applications by showing real examples of how to use GPIO and the I^2C devices. Now it's your turn to find more devices that you would like to connect to your Raspberry Pi so you can have fun with embedded Java on the Raspberry Pi. </article>

MORE ON TOPIC:

Java 8 Is Here

LEARN MORE

- Getting Started with Oracle Java ME Embedded 3.3 on the Keil Board