IOT APPLICATIONS WITH

Java and Raspberry Pi

BY STEPHEN CHIN

INTRODUCTION

The Raspberry Pi is a powerful and inexpensive embedded computing platform that has great community support. It is powerful enough to run a full Linux operating system, comes with Java SE pre-installed, and has digital input/output ports that you can use to interact with LEDs, buttons, sensors, and motors.

You can use Java on any of the Raspberry Pi models from the original Model B through the latest Model 3. Here is a list of all the different Raspberry Pi models and their capabilities:

MODEL	PROCESSOR	MEMORY	GPIO	USB
Raspberry Pi B	700Mhz 1 Core	256/512MB	28	2
Raspberry Pi A	700Mhz 1 Core	256MB	28	1
Raspberry Pi B+	700Mhz 1 Core	512MB	40	4
Raspberry Pi A+	700Mhz 1 Core	256MB	40	1
Raspberry Pi 2	900Mhz 4 Core	1GB	40	4
Raspberry Pi Zero	700Mhz 1 Core	512MB	40	1
Raspberry Pi 3	1.2Ghz 4 Core	1GB	40	4

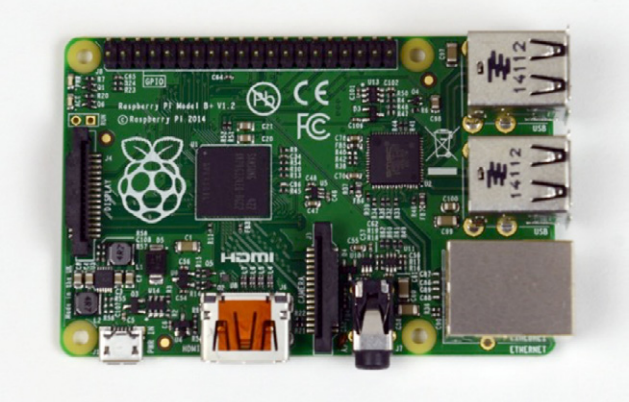


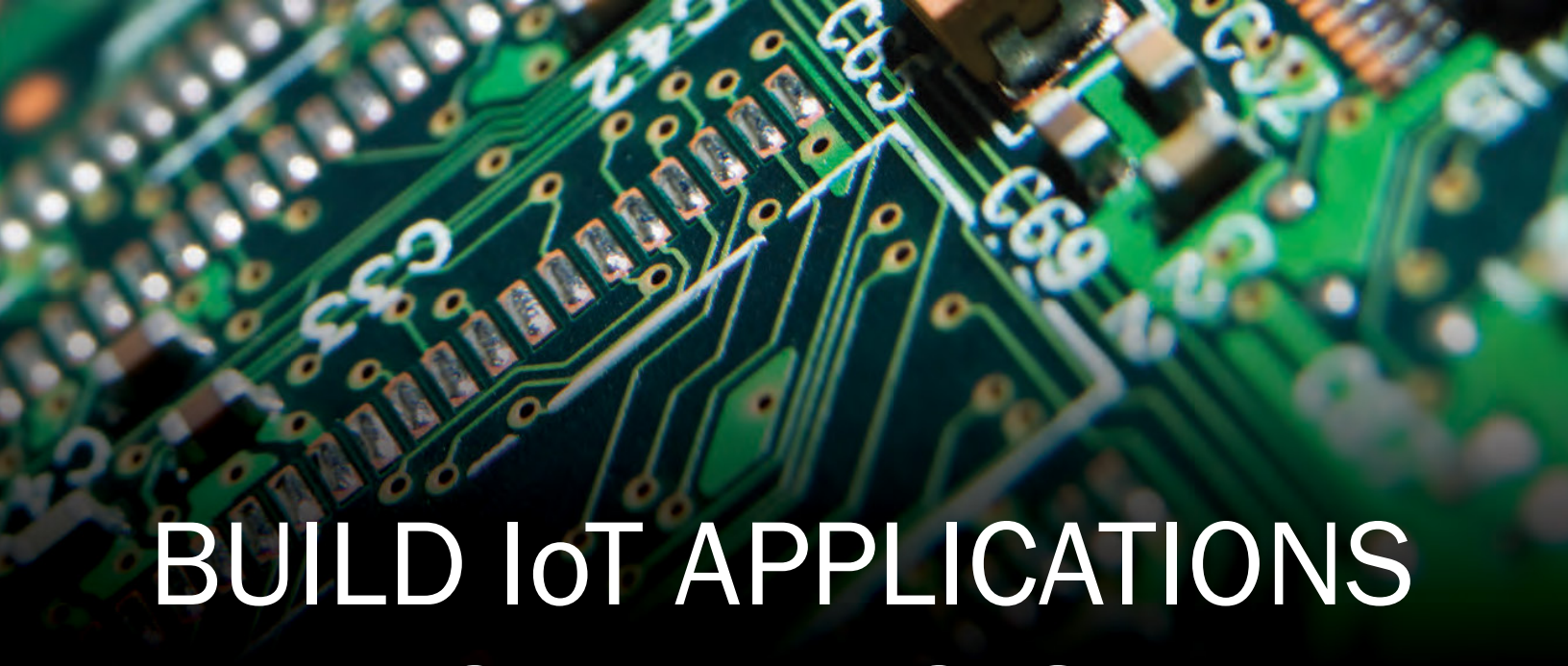
FIGURE 1: RASPBERRY PI B+ WITH 40 GPIO PINS (TOP LEFT), 4 USB PORTS (TOP RIGHT), ETHERNET (BOTTOM RIGHT), HDMI AND AUDIO/COMPOSITE (BOTTOM CENTER).

The Raspberry Pi 2 features a faster processor with exactly the same price point as the original Raspberry Pi B at 35 USD. One of the very latest models, the Raspberry Pi Zero, has an even more compact design than the Model A while keeping all the core features including a host USB port, HDMI display output, and 40 GPIO pins. And all of these boards support running Java right out of the box from the Raspbian distribution that the Raspberry Pi Foundation supports.

SETTING UP YOUR RASPBERRY PI

To setup your Raspberry Pi for the first time you will need the following hardware:

- **A Raspberry Pi of your choice** (The Raspberry Pi 2 and Zero are both great choices)
- **microSD Card** – Recommended to get the official Raspberry Pi SD Card that includes the NOOBS installer
- **Power Supply** – A Micro USB power supply capable of at least 700mA, but preferably 2A
- **Display or TV** – Any HDMI compatible display or TV will work (You can also use composite on all the models except the Zero)
- **Keyboard and Mouse** – A USB keyboard and mouse you can use to configure the Pi. It is possible to get through setup without a mouse, which is handy on the A, A+, and Zero models that only have one USB port.
- **HDMI Cable** – To connect to your TV/Display. If you are using the Zero make sure you have a cable with a mini jack end or the appropriate adapter to full size HDMI.



BUILD IoT APPLICATIONS AT **LIGHTNING SPEED**

The ThingWorx technology platform enables rapid development of IoT-ready applications. Build your app and connect your Raspberry Pi and other devices to the cloud; all in a single platform.

Join the ThingWorx Developer Zone for hands-on app development and device connectivity kits, dynamic learning content and connect with a network of IoT professionals.

GET ACCESS TO:

- IoT Application QuickStarts: Model an IoT application from end-to-end using ThingWorx, with step by step guidance, free for 30 days.
- IoT Device QuickStarts: Create an agent for the device of your choice, including Raspberry Pis, and enable that agent to communicate with the ThingWorx cloud.
- The ThingWorx blog, The ThingWorx Community, and more: Get the latest tips, tricks and real-life use cases from IoT industry experts and champions, analysts, customers and partners.

Join today and get building with ThingWorx™

thingworx.com/go/RaspberryPi

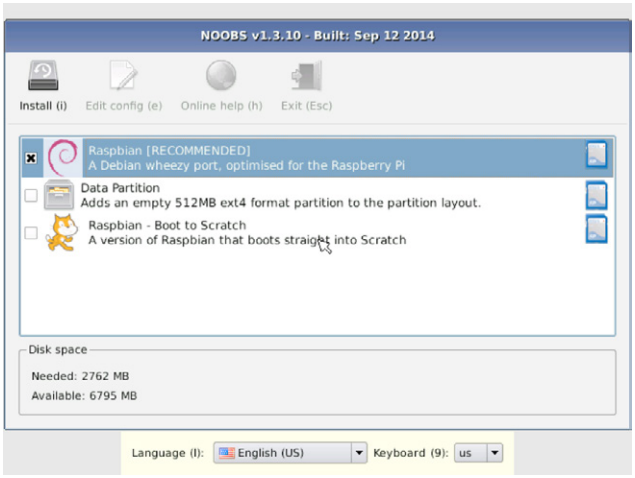


FIGURE 2: NOOBS INSTALLER SCREEN

Insert the SD Card with NOOBS on it, hook your Raspberry Pi up to the keyboard/mouse and Display, and then power it on with the Micro USB power supply. Upon doing this you should see a rainbow boot logo followed by the NOOBS installer.

Choose a standard Raspbian distribution, which is a Debian variant optimized for the Raspberry Pi. This also includes Oracle Java as part of the distribution allowing you to start coding with Java right out of the box. Installation takes around 20 minutes, so this is a good time to grab some liquid Java.

After install is complete, press "Enter" to reboot and the Raspbian operating system will load. On first boot you will be automatically logged in and sent to the Raspberry Pi Configuration Tool (raspi-config).

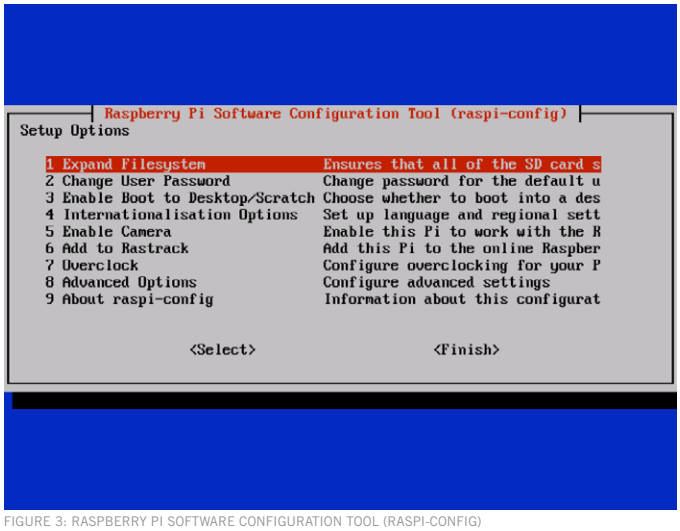


FIGURE 3: RASPBERRY PI SOFTWARE CONFIGURATION TOOL (RASPI-CONFIG)

In the config tool you can change various options, but I would recommend specifically tweaking the following:

- **Password (2)** – Default password is "raspberry" for the "pi" user. Make sure to change this if you use the Raspberry Pi on an unprotected network.
- **Internationalisation Options (4)** – The defaults assume you use a UK keyboard layout. If you are using a US keyboard it is highly recommended to change this or you will have trouble typing special symbols like the # and @ characters.
- **Advanced Options (8)** – And in this submenu:
  - **Overscan** – If you have a modern Display you can turn this off and reclaim your borders
  - **Memory Split** – If you plan to do any Java UI work (e.g. JavaFX), set this to 128mb or higher
  - **SPI** – Recommended to enable this so you can use the SPI bus for high speed communication with devices
  - **I2C** – Recommended to enable this so you can use the I2C bus, which is what most sensors require
  - **Serial** – Recommended to disable this to free up the serial pins for device communication (this option lets you connect via serial port in to the Raspberry Pi for headless setup, which is no longer required if you have made it this far)

Feel free to browse around and look through the other options, but the ones highlighted above are what I recommend tweaking on your first pass. You can always bring up the tool again by typing "raspi-config" on the command line.

The last bit of setup I recommend is networking your Raspberry Pi so you can access it from your computer via SSH. The easiest way to do this is to plug it in via Ethernet in which case it will automatically grab an IP address via DHCP. The IP address will print on startup, or can be queried using the "ifconfig" command. You can also connect to a wireless network using a USB wifi dongle.

To confirm you have everything working, reboot your Raspberry Pi and login (or better yet connect via SSH over the network). The default username and password are "pi" and "raspberry". To confirm you have Java working, type "java –version" on the command prompt and it should tell you the exact Java SE Runtime version that you have installed.

ACCESSING GPIO FROM JAVA

Two libraries are commonly used for accessing the GPIO pins on the Raspberry Pi: Device I/O and Pi4J. The Device I/O library, which is part of the OpenJDK project, provides a standard library for accessing I/O pins across any embedded platform, including the Raspberry Pi. The Pi4J library is an open-source library that is exclusive to the Raspberry Pi. Here is a quick comparison of the two libraries:

	DEVICE I/O	PI4J
Open Source	Yes	Yes
Portable to Java ME	Yes	No
API Optimized for Java 8	Yes	No
Support for New Raspberry Pi Boards	Lagging	Excellent
Advanced Features	None	Button Debouncing, Interrupt-based Listening
Extensible Providers for Add-On Boards	No	Yes
Low-Level GPIO Access	No	Yes

INSTALLING THE DEVICE I/O LIBRARY

Currently the Device I/O library is not released with the version of Java on the Raspberry Pi. Also, there are no binary builds available, so you have to build the library from source. Fortunately, this is a fairly easy process and can be accomplished directly on the Raspberry Pi.

To start, download the source code from the OpenJDK Mercurial repository by executing the following command from an SSH prompt or the console on your Raspberry Pi:

```
sudo apt-get install mercurial
```

Next, clone the Device I/O library source code to your Raspberry Pi. The following command creates a folder called dio in your current working directory with the latest source code:

```
hg clone http://hg.openjdk.java.net/dio/dev dio
```

To build the source code, you need to run the make command on the newly downloaded code. The build script also requires a few variables to be set up so it knows where to access the Raspberry Pi native libraries and the Java 8 JDK:

```
cd dio
export PI_TOOLS=/usr
export JAVA_HOME=/usr/lib/jvm/jdk-8-oracle-arm-vfp-hflt
make
```

Next, install it into the local Java Runtime Environment (JRE):

```
sudo cp -r build/deviceio/lib/* $JAVA_HOME/jre/lib
```

You will also need the libraries locally for your project. To do this you can use any visual Secure Copy (SCP) or Secure FTP (SFTP) client, such as Cyberduck. However, real geeks will do it from the command line (this works out of the box in OS X or Linux). Unlike the other commands, this one must be run from your desktop or laptop that is connected to your Raspberry Pi:

```
scp pi@timerpi.local:~/dio/build/deviceio/lib/ext/dio.jar dio.jar
```

Note: Substitute your Pi IP address or hostname for timerpi.local. Also, this command assumes that the dio folder was created in your user home directory on the Raspberry Pi. If not, adjust the path as appropriate.

To use the Device I/O library in your project, create a new Java project called DioLed in NetBeans and add dio.jar as a compile-time library. To access the library settings, choose File | Project Properties (DioLed), click Libraries in the Categories pane, and make sure the Compile tab is selected. Then click Add JAR/Folder and choose the dio.jar file you copied over from the Raspberry Pi.

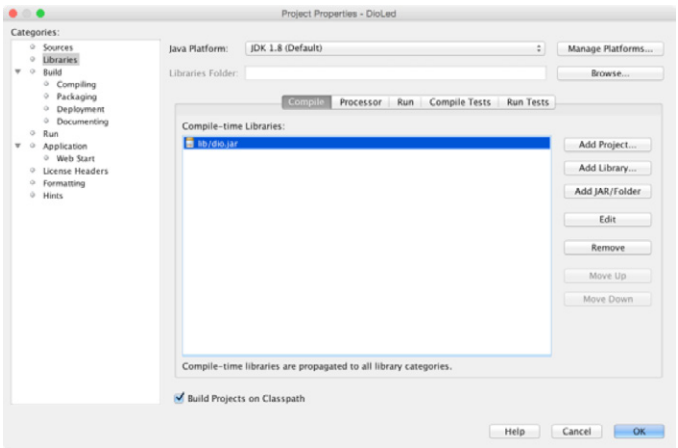


FIGURE 4: CONFIGURATION FOR THE DEVICE I/O LIBRARY IN NETBEANS

DEVICE I/O PIN ASSIGNMENTS

The Java Device I/O library uses the standard GPIO port assignments as specified by the Motorola Broadcom chipset. The following diagram shows a full list of all the ports, their numbers, and what their functions are for the Raspberry Pi B+, A+, 2, and Zero.

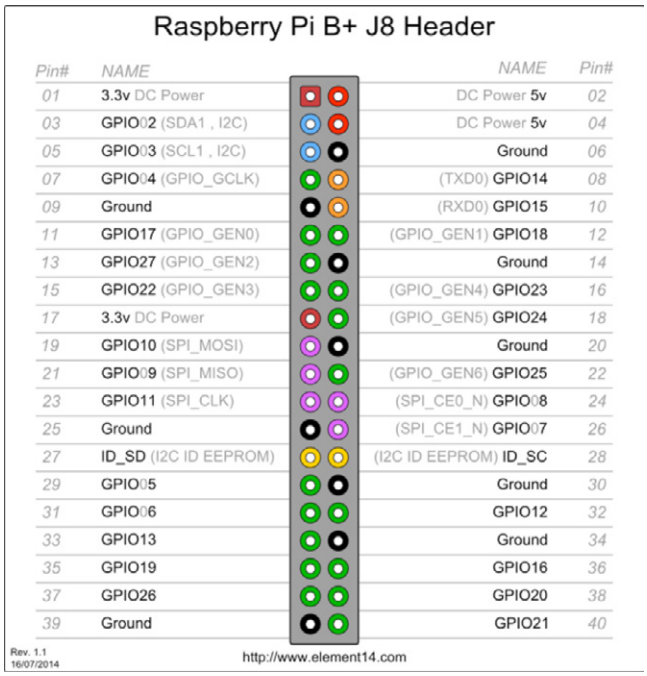


FIGURE 5: GPIO PORTS FOR THE RASPBERRY PI B+, A+, AND 2 (DIAGRAM BY CHRISTOPHER G. STANTON; SOURCE: WWW.ELEMENT14.COM/RASPBERRYPI.)

Even though there are 40 pins total, several of them are not usable for GPIO since they supply power (3.3v/5v) or ground. Most of the remaining pins can be used for GPIO, but may require some configuration before they are accessible. Specifically:

- **Pins 3 and 5 are for I^2C** By default, the I^2C bus is disabled and these pins can be used for GPIO. However, these have hard-wired internal 1.8kΩ pull-up resistors to 3.3V.
- **Pins 8 and 10 are for Serial UART** These pins are reserved for console serial communication by default, which needs to be disabled if you want to use them for GPIO or to connect to a serial device.
- **Pins 19, 21, 23, 24, and 26 are for SPI** This is another communication bus for devices that is disabled by default, so these pins can be used for GPIO.
- **Pins 27 and 28 are for EEPROM** These pins are designed for identifying add-on boards stacked on top of the Raspberry Pi and can't be used for GPIO.

DEVICE I/O LIBRARY LED TEST

To demonstrate how to use the Device I/O library for output, we are going to blink a light-emitting diode (LED). This is a great way to test out your GPIO pins since it is a visible device that can be powered by the current from the pins.

Connecting an LED to the Raspberry Pi GPIO pins just requires a few jumper cables. To prevent the LED from being overloaded, we also need a resistor that we can hook up in serial as shown in the following wiring diagram.

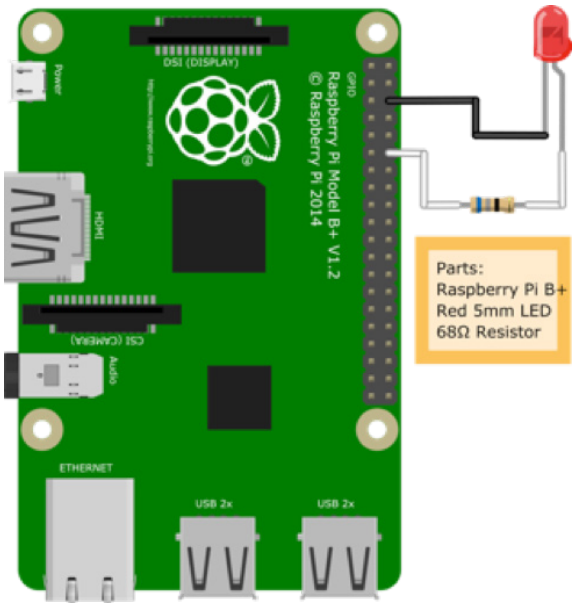


FIGURE 6: AN LED WIRED TO THE RASPBERRY PI GPIO PINS

This is a simple circuit with a 68Ω resistor, an LED, and some wires. Note that LEDs have a positive side and a negative side, usually indicated by the positive lead (the anode) being slightly longer than the negative lead (the cathode). If you hook up the LED backward, it won't damage it, but it won't light up either, so make sure that you hook up the longer end to the GPIO pin.

To turn this on and off with the Device I/O library, we need to create three files:

- **java.policy** The policy file that grants permissions for us to use GPIO
- **dio.properties** Defines the pin setup for our application
- **DioLed.java** Our main Java code that will blink the LED

The policy file is really a legacy from the Java ME Device I/O API and is intended for highly restricted environments where you want to limit the access to peripherals, such as cell phones. For applications running on small, embedded systems like the Raspberry Pi, you will most often have control of the entire embedded device and not be competing with other applications installed by end users, so a permissive policy file is fine. The following code is a great boilerplate permissions file to use for all your projects.

```
// grant all permissions for the DIO framework
grant {
    permission jdk.dio.DeviceMgmtPermission "*:*", "open";
    permission jdk.dio.gpio.GPIOPinPermission "*:*",
  "open,setdirection";
    permission jdk.dio.gpio.GPIOPortPermission "*:*";
    permission jdk.dio.i2cbus.I2CPermission "*:*";
    permission jdk.dio.spibus.SPIPermission "*:*";
    permission jdk.dio.uart.UARTPermission "*:*";
};
```

The pin configuration file requires a little bit more customization since you will often want to change which pins are used for input/output and possibly add or remove features like serial, I²C, and SPI. You can often take an existing pin configuration file and customize it or pare it down to the features you need. For the simplest cases, it is easy enough to create one from scratch as well. The following code opens a single pin for output.

```
gpio.GPIOPin = initValue:0, deviceNumber:0, direction:1, mode:4,
   trigger:3, predefined:true
1 = deviceType: gpio.GPIOPin, pinNumber:18
```

The first line defines the defaults for the gpio.GPIOPin device type.

Possible constants for direction (dir), mode, and trigger in Device I/O property files are as follows:

```
DIR_BOTH_INIT_INPUT = 2;
DIR_BOTH_INIT_OUTPUT = 3;
DIR_INPUT_ONLY = 0;
DIR_OUTPUT_ONLY = 1;
MODE_INPUT_PULL_DOWN = 2;
MODE_INPUT_PULL_UP = 1;
MODE_OUTPUT_OPEN_DRAIN = 8;
MODE_OUTPUT_PUSH_PULL = 4;
TRIGGER_BOTH_EDGES = 3;
TRIGGER_BOTH_LEVELS = 6;
TRIGGER_FALLING_EDGE = 1;
TRIGGER_HIGH_LEVEL = 4;
TRIGGER_LOW_LEVEL = 5;
TRIGGER_NONE = 0;
TRIGGER_RISING_EDGE = 2;
```

There is also a bit of ceremony around how you need to configure your NetBeans project to properly deploy the Device I/O configuration files and refer to them from the run command. To make sure your property and permission files are copied to the output directory, you need to add an extra build post-compile step to your NetBeans build.xml file.

```
<target name="-post-jar">
    <copy todir="${dist.jar.dir}">
        <fileset dir="config" includes="**"/>
    </copy>
</target>
```

The ant target assumes that both your java.policy and dio.properties files are in a folder called config under the project root.

Once these files are configured to copy over to the build folder after compilation, you also need to modify the java run command to make use of them. To do this, open the Project Properties dialog, click Run in the Categories pane, and enter the following VM Options:

```
-Djdk.dio.registry=dist/dio.properties -Djava.security.policy=dist/
  java.policy
```

And finally the Java code to turn on and off the LED in a file called DioLed.java.

```
public class DioLed {
  public static void main(String[] args) throws IOException,
  InterruptedException {
    try (GPIOPin led = DeviceManager.open(1);) {
      for (int i = 0; i < 10; i++) {
        led.setValue(i % 2 == 0);
        Thread.sleep(500);
      }
    }
  }
}
```

Notice that this code is fairly clean and succinct. It uses a try-with-resources block to open the GPIOPin as well as automatically close it at the end. The function for setting the LED value is very straightforward as well. One important thing to note is that the GPIO number referenced in the open method is the ID in our dio.properties file, not the Broadcom GPIO number!

USING PI4J

The Pi4J library is written by Robert Savage and based on the very widely used WiringPi C library. Compared to the Device I/O library, Pi4J has lots of features and is very easy to get started with.

To use Pi4J you just require one JAR file (pi4j-core.jar)—no permissions, no property files, no custom JVM arguments. However, you do require root access for the java process. To run with root from NetBeans, you need to:

1. Choose Tools | Java Platforms.
2. Click your Raspberry Pi platform under the Remote Java SE tree.
3. Click the Exec Prefix field and enter a value of sudo.

PI4J PIN ASSIGNMENTS

By default, Pi4J uses the WiringPi pin assignments. The advantage of this is that they are sequential and in the same location from revision to revision of the Raspberry Pi boards. In practice this only matters between revisions 1 and 2 of the Raspberry Pi B, because the Raspberry Pi Foundation has standardized on the main header pin layout for all subsequent boards, opting to add more pins to the header instead. Here is a diagram showing the pin assignments to use with Pi4J.

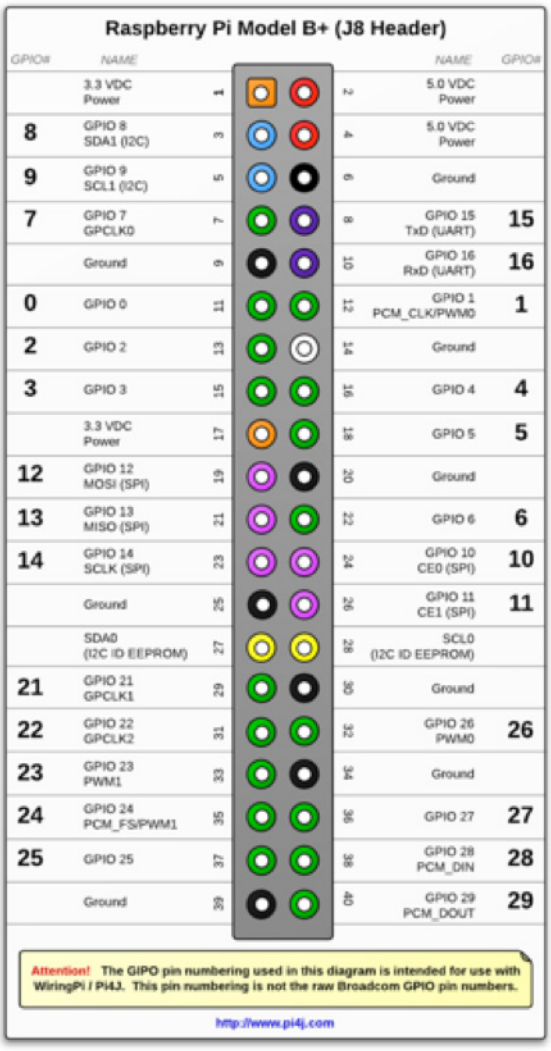


FIGURE 7: PIN ASSIGNMENTS FOR THE PI4J LIBRARY (DIAGRAM BY ROBERT SAVAGE; SOURCE: PI4J.COM/PINS/MODEL-B-PLUS.HTML.)

Pi4J also has the option to initialize pin assignments to the Broadcom GPIO pin numbering. However, I don't recommend changing this, since it will confuse others who are familiar with the Pi4J library and assume certain pin assignments.

PI4J LED TEST

To demonstrate the usage of the Pi4J library, we are going to recode exactly the same LED test using the Pi4J library instead. However, the setup of Pi4J is much simpler and the number of changes is minimal, so this will be a breeze!

To start, create a new project called Pi4JLed as a Java project type. Open the Project Properties dialog of your new project and go to the Libraries category. Click Add JAR/Folder, and select the pi4j-core.jar file. If you don't already have this library, you can download it from the Pi4J Project site here: pi4j.com.

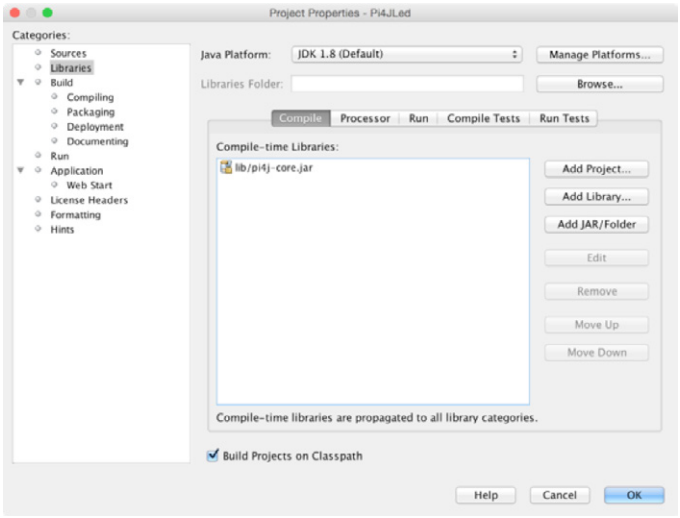


FIGURE 8: PI4J PROJECT IN NETBEANS

Now you can start writing the code for triggering the LED. Here is the same example with an LED blinking five times, coded using Pi4J libraries:

```
public class Pi4JLed {
  public static void main(String[] args) {
    GpioController gpio = GpioFactory.getInstance();
    GpioPinDigitalOutput led = gpio.
  provisionDigitalOutputPin(RaspiPin.GPIO_01);
    for (int i = 0; i < 10; i++) {
      led.setState(i % 2 == 0);
      Gpio.delay(500);
    }
    gpio.shutdown();
  }
}
```

Here are some of the differences that you should take note of:

- There is no try block, because Pi4J doesn't support auto-closable for use with try-with-resources. However, it does register shutdown hooks for you, so you can be lazy and let it do the work for you.
- Pi4J requires initialization before the use of the first pin. Shutdown is optional, but recommended if you don't want the system to wait for threads to time out, which can take up to 30 seconds.
- There are no exceptions in the Pi4J version! Pi4J doesn't throw IOExceptions that we have no idea how to recover from (thankfully!) Also, rather than using Thread.sleep, there are some nice convenience methods for this on the Pi4J GPIO class.
- The GPIO port is the same. Well, this is not a difference, but it is a very important similarity by coincidence. We are

referring to Broadcom GPIO port 18 in both cases. This maps to WiringPi port 1 (per the diagram in the previous section), and we also happen to have slotted it into configuration 1 in the dio.properties file. However, don't rely on ports being the same in general.

And there you have it. In a single section you were able to replicate the same LED blinking code that took three sections of this chapter to explain using the Device I/O library. Now we can move on to more advanced examples for making use of the GPIO capabilities of the Raspberry Pi.

WHERE TO GO FROM HERE

Now that you have learned to code Java on the Raspberry Pi and access the GPIO pins, a wealth of different project possibilities become possible.

Here are just a few different ideas:

SELF DRIVING CAR

Take advantage of an infrared sensor to follow a line, and distance sensor to stop when faced with an obstacle.

AUTONOMOUS DRONE

Hook up a quadcopter using a Raspberry Pi as the controller to enable autonomous flight algorithms.

"MAGIC" RFID HAT

Read RFID tagged playing cards with an RFID/NFC reader hooked up to the Raspberry Pi.

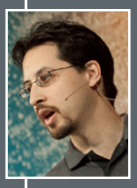
PORTABLE GAMING CONSOLE

Create a 100% Java-based gaming system using a Raspberry Pi, touchscreen, GPIO buttons, and a 3D printed case.

AND MORE!

These examples and more can be found in the "Raspberry Pi with Java" title published by McGraw Hill.

ABOUT THE AUTHOR



STEPHEN CHIN is the Lead Java Community Manager at Oracle, author of Raspberry Pi with Java, co-author of Pro JavaFX Platform, and JavaOne Community Chair. He has keynoted numerous Java conferences around the world including JavaOne, where he is a 5-time Rock Star Award recipient. Stephen is an avid motorcyclist who has done several Pan-European evangelism tours, interviewing hackers in their natural habitat and posting the videos on nighthacking.com. When he is not traveling, he enjoys teaching kids how to do embedded and robot programming together with his 12-year-old daughter.

REVIEWED BY: Mark Heckler, @MkHeck

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

VERSION 1.0