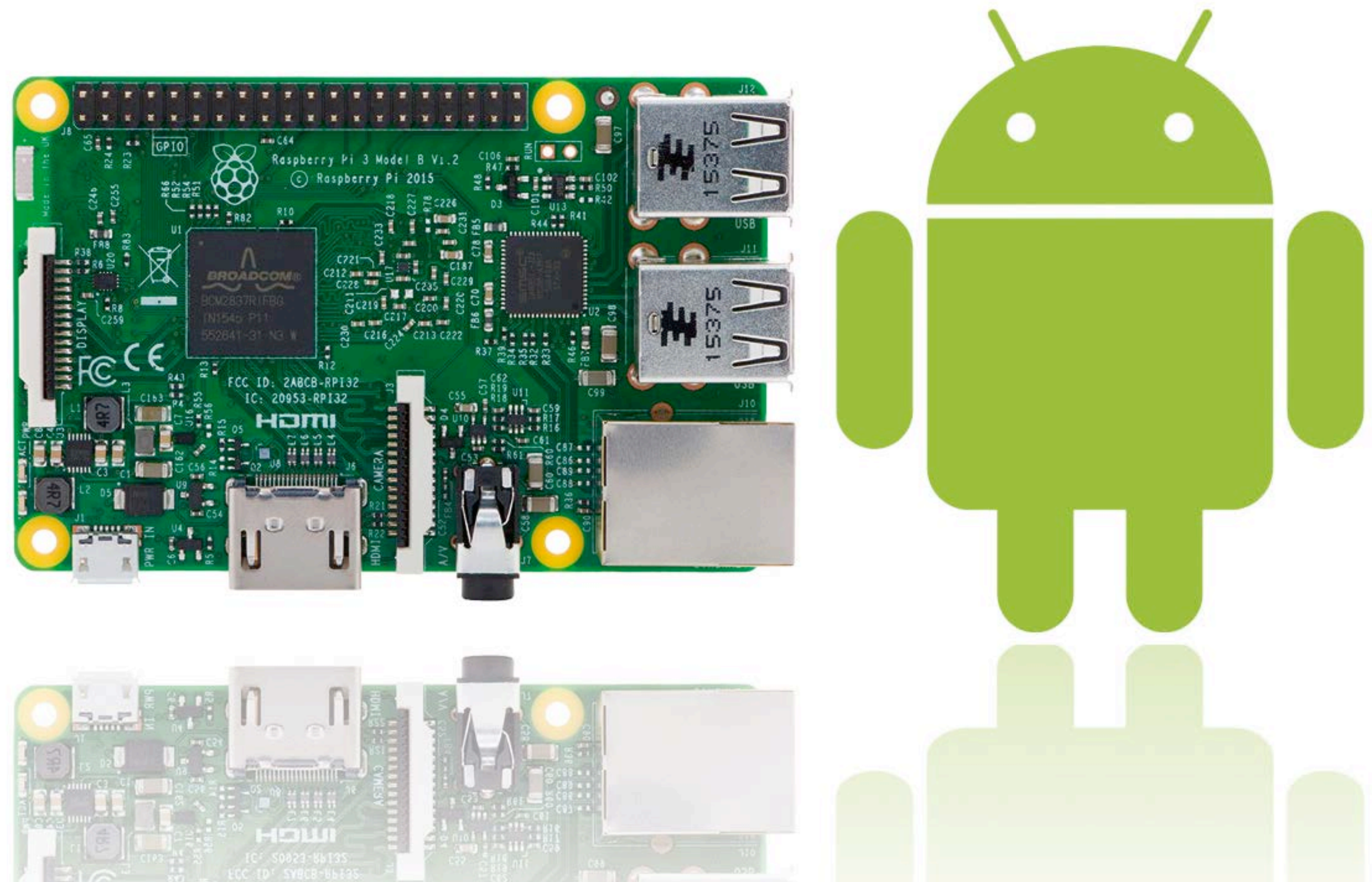
Android on your Rpi (1)

Using GPIO Pins for measurement and control functions



Google has released a new version of Android — especially for single-board computers and for measurement and control applications. Having tested Android on the Raspberry Pi, we can now show you in simple steps how to get a first 'Hello World' project to work.

By **Tam Hanna** (Slovakia)

Microsoft's rapid advance into Raspberry Pi territory has certainly caused Google some headache. After desktop computers, tablets and smartphones, the Internet of Things (IoT) looks like becoming the next battleground.

Pay attention to quality!

If you are ordering memory cards from China make sure you carry out detailed quality tests. Many manufacturers offload batches of inferior quality products that may well report the full complement of memory to your microcontroller. In actual use, however, your data may be lost.

With the Redmond brigade now well advanced along the road towards platform independence, convergence is the new keyword. Google simply cannot afford to ignore the IoT arena, as otherwise Microsoft could exploit this and open a second front on which to attack the smartphone market.

However, Google's *Brillo* platform and language for the Internet of Things announced about a year ago made little impact. Maybe the name was the problem. Whatever the reason, for its second attempt, Google knew it must go direct for the jugular and named its new product simply 'Android Things'.

Simple access to hardware

Let's begin with the obvious. Despite all the kerfuffle — comparisons with Microsoft's 'Windows 10 IOT Core' are purely coincidental — Android Things is by and large a normal version of Android, albeit one that contains some extensions for interacting with hardware. **Figure 1** shows the structure of the operating system.

The *Things Support Library* incorporated with this consists of two modules. First up is the *Peripheral I/O API*, which at the time of writing provides access to the following four types of peripheral (whether OneWire will be retrofitted is currently unclear):

- GPIO with PWM
- I²C
- SPI
- UART

The second component is the *User Driver API*. We're talking here about a program-

ming interface that developers can use to make information on their own proprietary sensors available to the rest of the operating system. Further information on the modules included in the *Things Support Library* can be found at [1].
Let's venture now into some first attempts using practical hardware.

Preconfigured images

At press time Google (doubtless having learnt from its flop with *Brillo*) already offered ready-built Images for the Intel Edison, NXP Pico and Raspberry Pi 3 platforms. In this first article we shall focus on the (by far most widely selling) Raspberry Pi 3.
The first stage involves downloading the Image that awaits you at [2] and then burning it in the normal way onto a memory card (in this connection don't miss the **text panel** *Pay attention to quality*). For the steps that follow the author used an 8 GB capacity card. Using cards of greater capacity should not be a problem; on the other hand the Image size of 4.6 GB means that it will not work with cards any smaller than this.

The next step is to connect the screen and a network cable to your router on the RPi, and start it up by connecting the power supply. Note that a keyboard and mouse are not required, because Android in itself registers only one desktop, without relevant interaction possibilities. We do not want to get involved with debugging over WLAN, because the higher latency of wireless connections can lead to delays while you are tracking down for errors. Nevertheless, if you do want to try this, you will need to connect the RPi to the wireless network following the instructions given in [3].
The first time you start up Android Things on a Raspberry Pi 3 it will take a couple of minutes. At the very beginning, a message appears about Ethernet being missing; you can safely ignore this if the network cable is already connected.
With this work done, the operating system presents the start screen shown in **Figure 2**. The IP address — 10.42.0.44 in this case — will be needed shortly.

Development environment

For space reasons, we have to assume at this point that you already have a working installation of Android Studio on your development PC. If this is not the case, please refer to [4] for help and advice. The author used an AMD eight-kernel workstation with Ubuntu 14.04 to perform the following stages (it's very similar under Windows and Mac OS).
In the next step, change to the root directory of the *Android Debug Bridge* (ADB) on the PC. Then connect the development computer to the RPi by typing the console commands shown in bold as follows:

```
tamhan@TAMHAN14:~/Android/Sdk/
   platform-tools$ ./adb connect
   10.42.0.44

connected to 10.42.0.44:5555

tamhan@TAMHAN14:~/Android/Sdk/
   platform-tools$ ./adb devices

List of devices attached
10.42.0.44:5555 device

tamhan@TAMHAN14:~/Android/Sdk/
   platform-tools$
```

Inputting adb devices is not absolutely necessary here — we are showing you the command because it allows you to identify all devices connected to the Android Debug Bridge.
Also, remember that the connection between the ADB and the RPi can be lost, for example when your PC falls asleep.

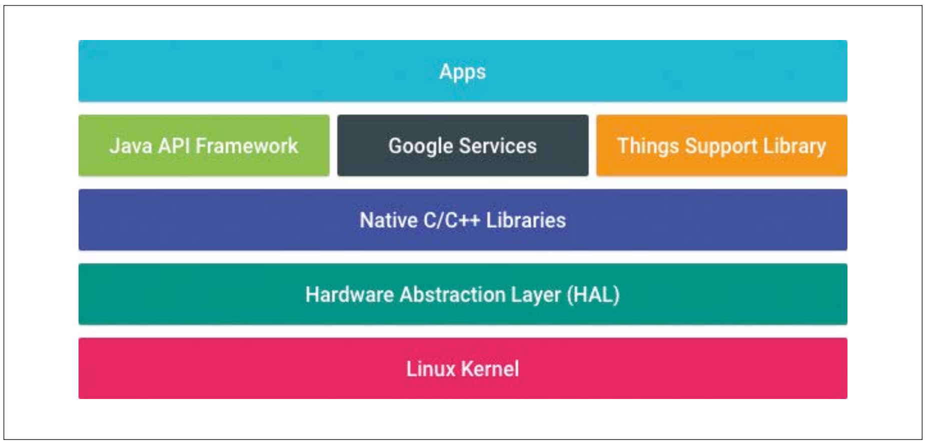


Figure 1. Android Things is a variant of Android (image credit: Google).

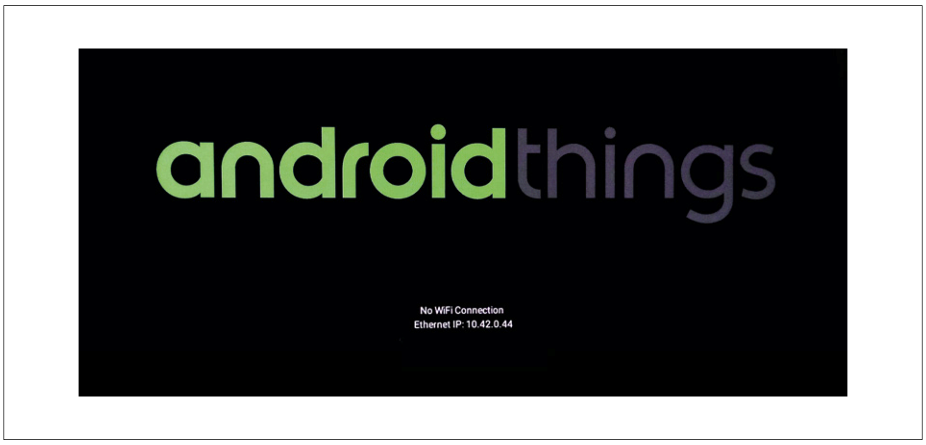


Figure 2. Our own 'Thing' is ready to put into use — remember to make a note of the IP address.

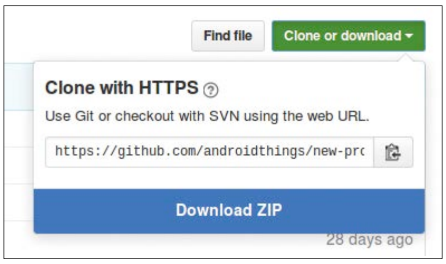
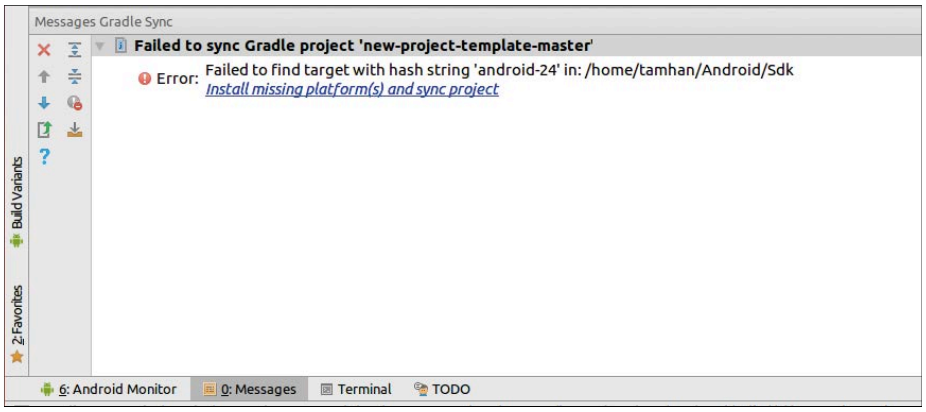

Find file
Clone or download
Clone with HTTPS
Use Git or checkout with SVN using the web URL.
https://github.com/androidthings/new-prc
Download ZIP
28 days ago

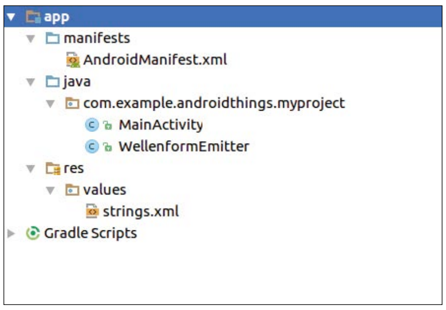
Figure 3. Using GitHub has always been a challenge and remains so.



Failed to sync Gradle project 'new-project-template-master'

Error: Failed to find target with hash string 'android-24' in: /home/tamhan/Android/Sdk
Install missing platform(s) and sync project

Figure 4. A compatible SDK is missing here.



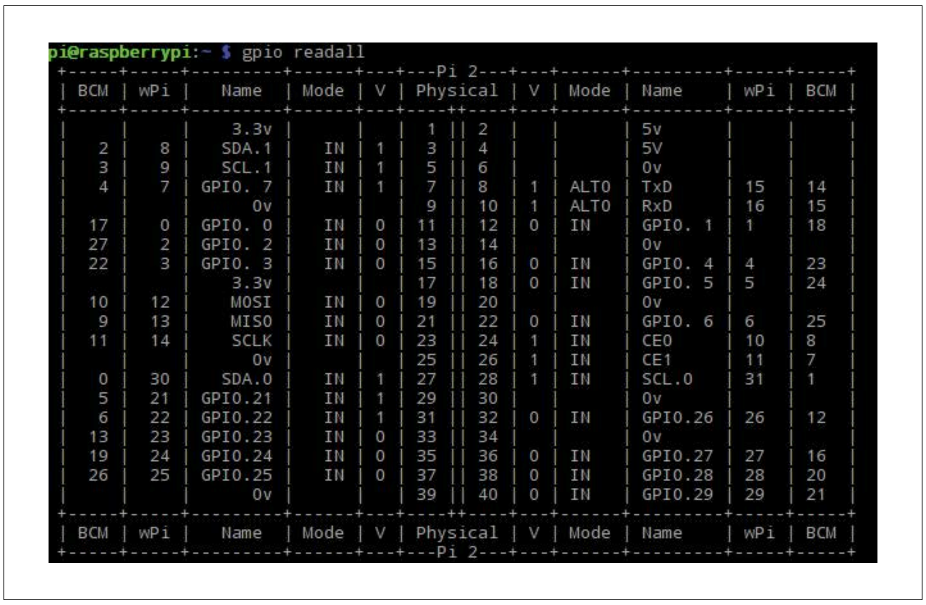
```
app
  manifests
    AndroidManifest.xml
  java
    com.example.androidthings.myproject
      MainActivity
      WellenformEmitter
  res
    values
      strings.xml
Gradle Scripts
```

Figure 5. This application gets by without XML markup.

Hello World isn't responding.

Do you want to close it?

Figure 7. Anyone who blocks the GUI Thread will be punished (image credit: Google).



```
pi@raspberrypi:~ $ gpio readall
 +-----+-----+---------+------+---+---Pi 2---+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 |     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
 |   2 |   8 |   SDA.1 |   IN | 1 |  3 || 4  |   |      | 5V      |     |     |
 |   3 |   9 |   SCL.1 |   IN | 1 |  5 || 6  |   |      | 0v      |     |     |
 |   4 |   7 | GPIO. 7 |   IN | 1 |  7 || 8  | 1 | ALT0 | TxD     | 15  | 14  |
 |     |     |      0v |      |   |  9 || 10 | 1 | ALT0 | RxD     | 16  | 15  |
 |  17 |   0 | GPIO. 0 |   IN | 0 | 11 || 12 | 0 | IN   | GPIO. 1 | 1   | 18  |
 |  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
 |  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO. 4 | 4   | 23  |
 |     |     |    3.3v |      |   | 17 || 18 | 0 | IN   | GPIO. 5 | 5   | 24  |
 |  10 |  12 |    MOSI |   IN | 0 | 19 || 20 |   |      | 0v      |     |     |
 |   9 |  13 |    MISO |   IN | 0 | 21 || 22 | 0 | IN   | GPIO. 6 | 6   | 25  |
 |  11 |  14 |    SCLK |   IN | 0 | 23 || 24 | 1 | IN   | CE0     | 10  | 8   |
 |     |     |      0v |      |   | 25 || 26 | 1 | IN   | CE1     | 11  | 7   |
 |   0 |  30 |   SDA.0 |   IN | 1 | 27 || 28 | 1 | IN   | SCL.0   | 31  | 1   |
 |   5 |  21 | GPIO.21 |   IN | 1 | 29 || 30 |   |      | 0v      |     |     |
 |   6 |  22 | GPIO.22 |   IN | 1 | 31 || 32 | 0 | IN   | GPIO.26 | 26  | 12  |
 |  13 |  23 | GPIO.23 |   IN | 0 | 33 || 34 |   |      | 0v      |     |     |
 |  19 |  24 | GPIO.24 |   IN | 0 | 35 || 36 | 0 | IN   | GPIO.27 | 27  | 16  |
 |  26 |  25 | GPIO.25 |   IN | 0 | 37 || 38 | 0 | IN   | GPIO.28 | 28  | 20  |
 |     |     |      0v |      |   | 39 || 40 | 0 | IN   | GPIO.29 | 29  | 21  |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+---Pi 2---+---+------+---------+-----+-----+
```

Figure 6. The table indicates the Pins of the Raspberry Pi.

Android Things is still at an early stage of development and there is no template available yet in the project generator of Android Studio. In its absence you need to call up the existing GitHub Repository [5] and download a sample project skeleton by clicking on the Button shown in **Figure 3**.
If you are already running a project in Android Studio now, close it by clicking *File → Close Project*. The IDE will then display the Welcome dialog. At this point, you must extract the Archive that you downloaded in the previous step and move this into a convenient location in the file system. Then click *Open existing Android Studio Project* and cross over into the directory that contains this project. After you have clicked OK, the IDE will begin to deploy the project. As part of the synchronization of the Gradle project, the IDE will attempt to download any missing components automatically. If any problems arise, you will see an error message along the lines of **Figure 4**; clicking the hyperlink usually resolves the problem.

Do not be surprised if Android Studio complains more than once to report it has found obsolete versions. Android Things requires you to use the absolutely latest version of several components of the operating system, which must be installed in several steps.

After launching the IDE successfully, you should click *Build → Make Project* once more in order to launch a complete compilation. It is important to note that an Internet connection is required when you make the first compilation; afterwards you will also have a realistic chance of being able to work offline with your project skeleton.

Alles anderDifferent strokes...

The MainActivity code serving as starting point is different from normal Android applications, because Google has added some extra logic calls. These are necessary because Android Things is able to work either with or without a monitor screen. In the latter case, thanks to the Log Calls, you do at least receive information in the debugger console.

Also note that the *res* directory is blank, as shown in **Figure 5**. There is no XML Markup supporting this activity. This affects the code as shown in **Listing 1** — the call to load of resources from XML normally available is missing here.

Another thing to concern us is the construction of the Manifest file, shown in abbreviated form as **Listing 2**. Two modifications are of interest here.
Firstly, the Library section ensures that the Libraries mentioned above are inte-

grated into the project. Secondly, MainActivity has an additional *Intent Filter* written into it, which characterizes it as an entry point for Android Things devices.

This is relevant because Android Things has to make do without a program start. RPi is basically a 'one trick pony' that simply processes its host application.

Input and output

For our first demo let's turn to the GPIO Engine. There's an old saying stating that the real-time capability of an operating system decreases linearly (or even logarithmically!) with its complexity. Java-based systems are particularly disagreeable in this respect, with the *Garbage Collector* causing grief from time to time. At this point you must be utterly and completely forewarned that the Raspberry Pi is a 3.3 V platform — connecting 5 V subsystems will result in certain disaster!

With that said we can now turn our attention to the application code in MainActivity (**Listing 3**). First we declare two additional Member Variables. The Peripheral API is implemented in the form of a System Service. A Service is generally available across the entire operating system; applications that need to make use of it obtain a Referral and then interact with it. The Gpio Class is responsible for the actual switching on and off of the Pins.

The Function onCreate (Listing 3) is used to procure the Service. Even if Google is not 100 % orientated towards the Arduino where GPIO Functions are concerned, it's perfectly clear what's going on here. The BCM Strings can be assigned to individual Pins on the basis of the table in **Figure 6**. This table is obtained by entering the command GPIO Readall on a Raspberry Pi (loaded with the standard operating system Raspbian, not Android).

Start the music!

In order to determine the reliability or real-time efficiency of an operating system, it is a good idea to implement a periodically changing output on the GPIO Pin as quickly as possible. If you examine the spectrum on a modulation domain analyzer, you can draw some conclusions. In theory, there should be nothing wrong with implementing periodic on and off-switching of the Pin directly in the OnCreated Function, which is called up at the start of the program. However, if you try this, you will be confronted with the error message shown in **Figure 7**.

The reason for this admittedly rather strange behavior lies in a peculiarity of the Android operating system: the program's user interface is managed in a dedicated Thread called a GUI Thread. If you block this, the operating system punishes you without leniency by 'shooting down' your application. An endless loop would be a classic 'obstruction' that would certainly earn you no mercy.

To avoid this problem we can outsource our Routine in a further Thread.
The simplest way of generating a

Listing 1. Framework of the MainActivity — starting point of the program without user interface.

```
public class MainActivity extends Activity {
    private static final String TAG = MainActivity.class.
   getSimpleName();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d(TAG, “onCreate”);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.d(TAG, “onDestroy”);
    }
}
```

Listing 2. Manifest file with Library integration.

```
<manifest . . .>
    <application
      - - ->
        <uses-library android:name=”com.google.android.Things”/>

        <activity android:name=”.MainActivity”>

            . . .

            <intent-filter>
                <action android:name=”android.intent.action.MAIN”/>
                <category android:name=”android.intent.category.
   IOT_LAUNCHER”/>
                <category android:name=”android.intent.category.
   DEFAULT”/>
            </intent-filter>

        </activity>
    </application>
</manifest>
```

Listing 3. The Gpio Class responsible for switching Pins on and off.

```
public class MainActivity extends Activity {

    private static final String TAG = MainActivity.class.
   getSimpleName();
    PeripheralManagerService service;
    Gpio myGPIO;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d(TAG, "onCreate");

        PeripheralManagerService service = new
   PeripheralManagerService();
        try{
            myGPIO = service.openGpio("BCM6");
            myGPIO.setDirection(Gpio.DIRECTION_OUT_INITIALLY_LOW);
            myGPIO.setValue(true);
        }
        catch (Exception e){Log.d(TAG, "Fehler:" + e.getMessage());}
```

Thread is to create a *Runnable.* So your first step is to create a new Class employing the following structure:

```
public class WaveformEmitter
   implements Runnable {
    @Override
    public void run() {

    }
}
```

Listing 4. The endless loop of signal generation is performed in a Thread of its own — a Runnable serves as Container.

```
public class WaveformEmitter implements Runnable {
    Gpio myGpio;
    public WaveformEmitter(Gpio _which){
        myGpio=_which;
    }
    @Override
    public void run() {
        try {
            while(1==1){
                myGpio.setValue(true);
                myGpio.setValue(false);
                myGpio.setValue(true);
                myGpio.setValue(false);
                myGpio.setValue(true);
                myGpio.setValue(false);
            }
        }
        catch (Exception e){}
    }
}
```

Runnables are basically a kind of container in which we 'pack' the logic that executes the new Thread. In addition to the run Method described here, you can of course implement various Members in order to provide the information necessary for executing the Thread.
In its fully completed state the resulting Class now looks like **Listing 4**.
In addition to a Constructor responsible for accepting the GPIO Instance, we also have now populated the Method run(), which is what produces the actual waveform output. We generate a characteristic waveform from three rectangles: what's interesting in this instance is that the duration of the execution of the while loop is shown separately.

The question now arises how we activate the Runnable in OnCreated. A classic mistake made by novices is to invoke run() directly — doing this executes the code in the context of the activating Method (that means in the GUI Thread again). The correct way to activate a second Thread is to create a new Thread Class, with an Object indicated as *Payload* for the Parameter (**Listing 5**). The Thread is initiated using the start()Method.

We are now ready to toss the program in the RPi's direction.
Thanks to the ADB, which acts as an abstraction layer, it's sufficient to merely click on Run — the RPI behaves like a phone connected to the PC via USB. Since the MainActivity does not contain a user interface, a plain white screen appears if a monitor is connected, letting us know that our activity is cheerfully getting on with its work.

Debriefing

The next step is to connect your RPi to a modulation domain analyzer, so you can admire the screenshot shown in Figure 8 (the author provides an English language video at [6] giving further information on the function and benefits of using a modulation domain analyzer).
The most conspicuous feature, apart from the occasional jitter, is the formation of two prominent peaks. The region around 2.073 kHz shows the two 'linear sweeps' during the transition through the while loop, leading to the somewhat lower frequency of 'only' 2.062 kHz.

It's obvious that, in this situation, Android simply cannot match the pace of classic Unix; the interposed Java VM is taking its toll and slowing things down. You can read more on the real-time behavior of Android in various papers on the Internet (for instance in [7]).

The verdict

The program just created is a classic example of overkill in the embedded domain: a shrewd programmer could generate a stable waveform without any real-time operating system, using just a few lines of code and an IC.

Mind you, Bit banging and all that — let's be honest — is definitely not the intended application use of Android Things. On the other hand, the platform will always play its strengths when it comes to realizing demanding applications. In our next issue we'll show you how to interpret data from sensors and display the result in a diagram. Until then I wish everybody good coding! ⏮

(160361)

Listing 5. Start of our signal generation Thread in MainActivity.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    . . .

    WaveformEmitter myEmitter=new WaveFormEmitter(myGPIO);
    new Thread( myEmitter ).start();
}
```

Web Links

[1] https://developer.android.com/things/sdk/index.html

[2] https://developer.android.com/things/preview/download.html

[3] https://developer.android.com/things/hardware/raspberrypi.html

[4] https://developer.android.com/studio/install.html

[5] https://github.com/androidthings/new-project-template

[6] www.youtube.com/watch?v=IBLEfVUVGyU

[7] www.utdallas.edu/~cxl137330/courses/fall14/RTS/papers/4a.pdf

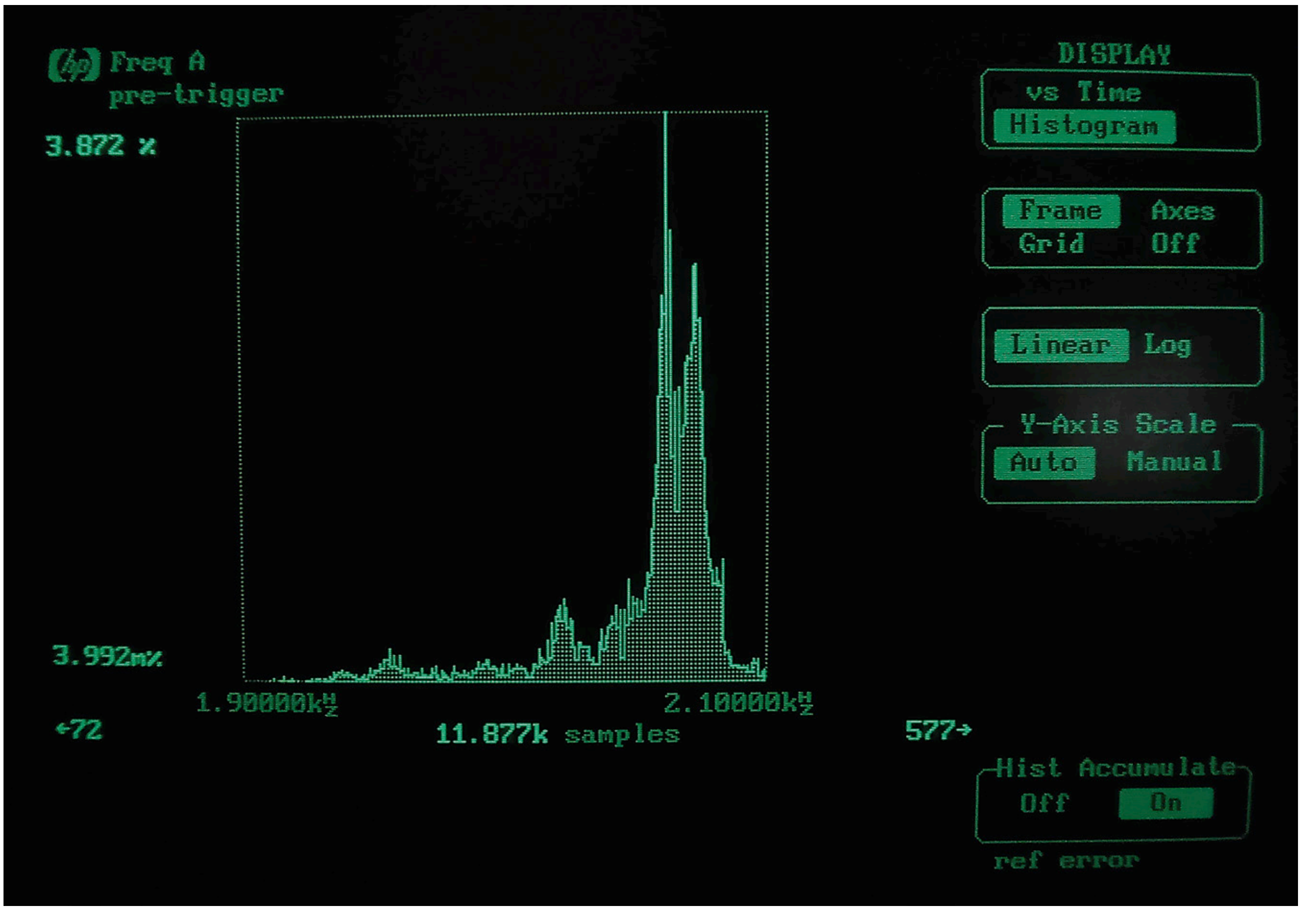


Figure 8. Arduinos generate significantly 'prettier' histograms in this application.