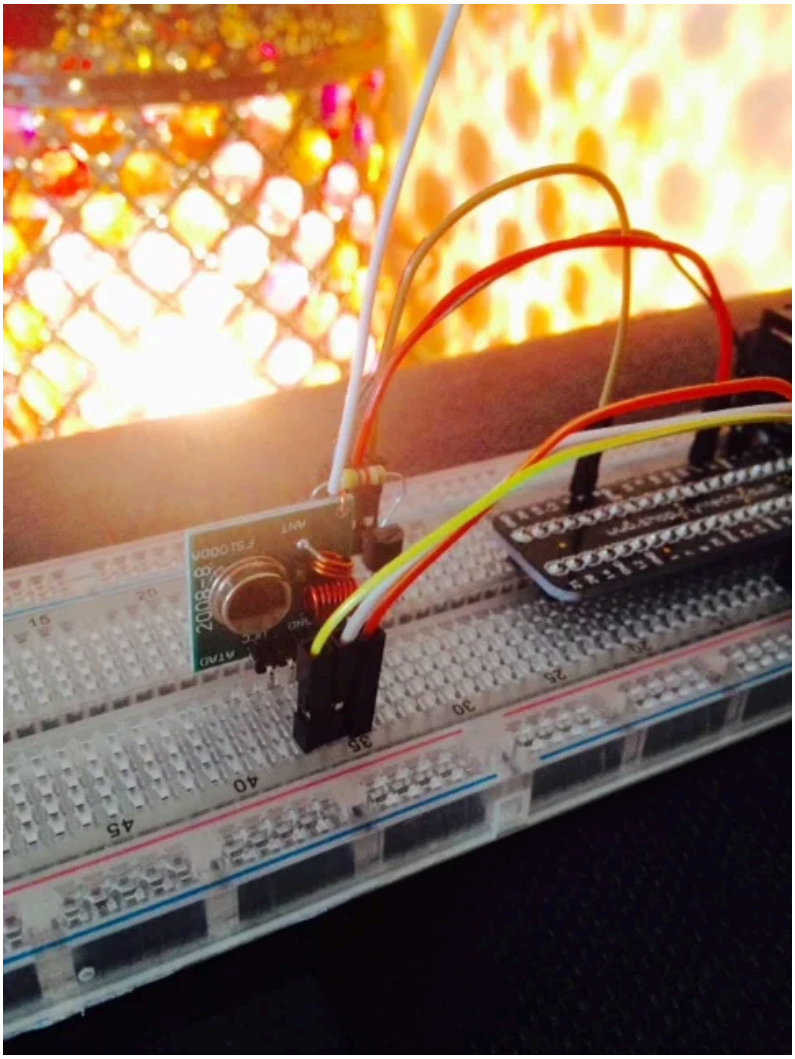
AUTODESK Instructables

Super Simple Raspberry Pi 433MHz Home Automation

By george7378 in CircuitsRaspberry Pi

Introduction: Super Simple Raspberry Pi 433MHz Home Automation



This tutorial is one among many when it comes to using a Raspberry Pi to control wireless devices around the home. Like many others, it will show you how to use a cheap transmitter/receiver pair hooked up to your Pi to interact with devices operating on the commonly used 433MHz radio frequency band. It will specifically show you how to turn any electrical device on or off using your Pi by transmitting commands to a set of 433MHz remote-controlled power sockets.

Why did I create this tutorial if so many already exist? Mainly because pretty much all the other tutorials I came across seemed to overcomplicate things, especially on the software side. I noticed that they relied heavily on third-party libraries, scripts or code snippets to do all the work. Many wouldn't even explain what the underlying code was doing - they would just ask you to shove two or three pieces of software on your Pi and execute a bunch of commands, no questions asked. I really wanted to try and use my Pi to turn electrical devices on and off around my home using a set

of 433MHz remote-controlled sockets, but I wanted to create my own version of the system that I could understand, hopefully eliminating the need to use someone else's libraries or scripts.

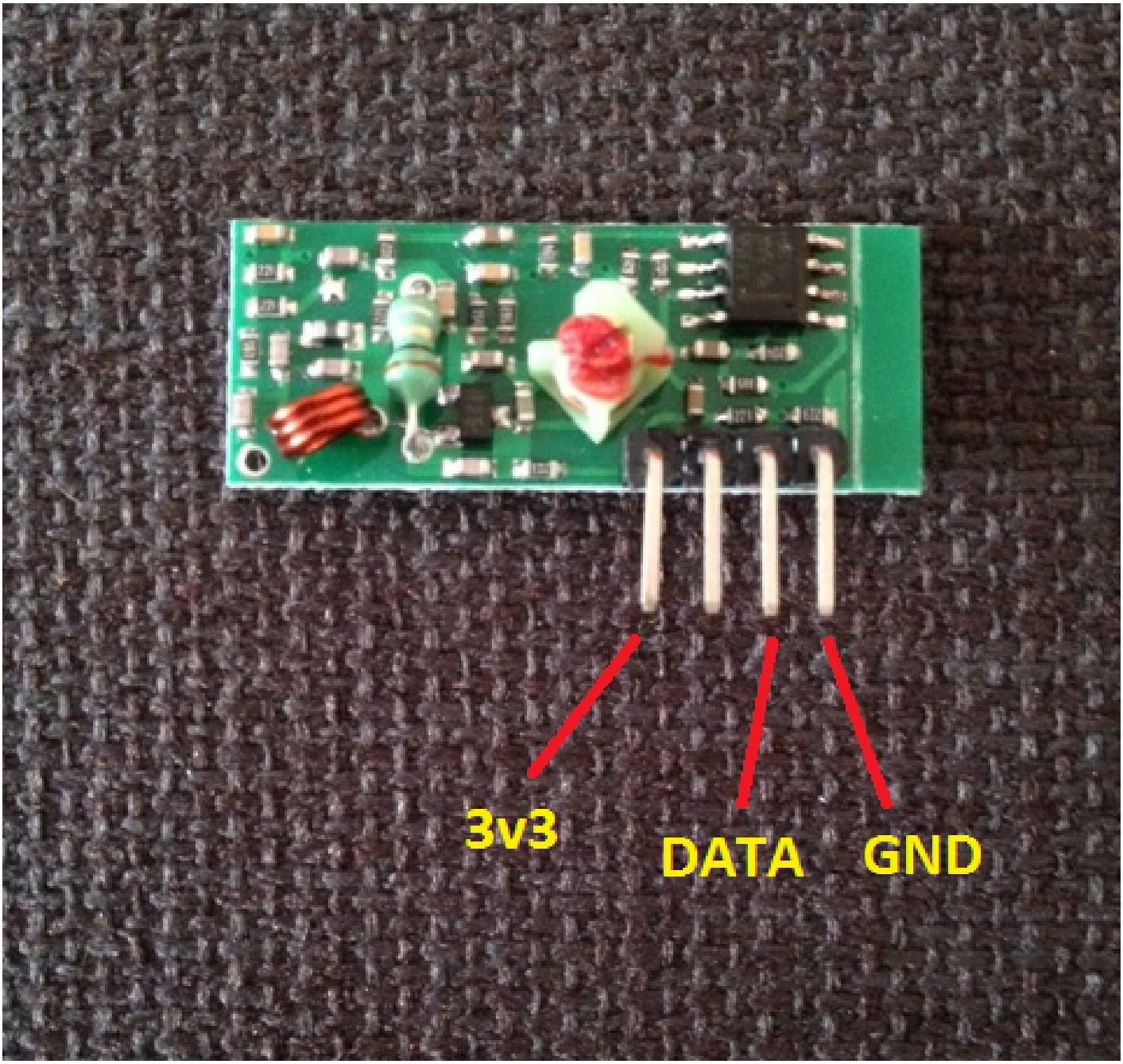
That is what this tutorial is about. The software side of this system consists of two very simple Python scripts - one for receiving and recording signals, and one for transmitting these signals back to the wireless power sockets. The actual reception/transmission of the signal relies only on the easy-to-use RPi.GPIO library which, at least for me, came pre-installed with Raspbian. This library can also be imported directly into Python.

For this project you will need:

1. *A Raspberry Pi*. Any model should work, I used an all-in-one starter kit, but perhaps you need the central unit only.
2. *A 433MHz transmitter/receiver pair*. The ones most commonly used in this type of project seem to be these. Buying a pack of five like the one linked ensures that you have a few spares.
3. *A set of 433MHz remote-controlled power sockets*. I used these which I'd highly recommend, but there are countless models available. Just make sure they operate on this frequency!
4. *Some circuit-building accessories*. I'd recommend using a breadboard and some jumper cables to make the circuit building process as easy as possible.

[If you decide to buy any of these products, I would greatly appreciate it if you access the listings using the above links - that way, I get a tiny share of the profits at no extra cost to you!]

Step 1: Setting Up the Receiver Unit



Before you can use your Pi to send commands to the remote-controlled sockets, you need to know what specific signals they respond to. Most remote-controlled sockets ship with a handset that can be used to turn specific units on or off. In the case of the ones I bought, the handset has four rows of paired ON/OFF buttons, each of which sends out an ON or OFF signal to a particular socket unit.

This brings up a question - how do we know which buttons correspond to which socket? This actually depends on the model you have. One of the main reasons I chose my particular style of socket (linked in the introduction) is that the units can be configured with a physical switch to make a particular socket respond to a particular set of ON/OFF buttons on the handset. This also means that you can unplug and move the sockets around the house knowing that each unit will always respond to the same ON/OFF signals.

Once you have figured out how your sockets interact with the handset, you will need to use your 433MHz receiver unit (pictured above) to 'sniff' the codes being sent out by the handset. Once you have recorded the waveforms of these codes, you can replicate them using Python and send them out using the transmitter unit.

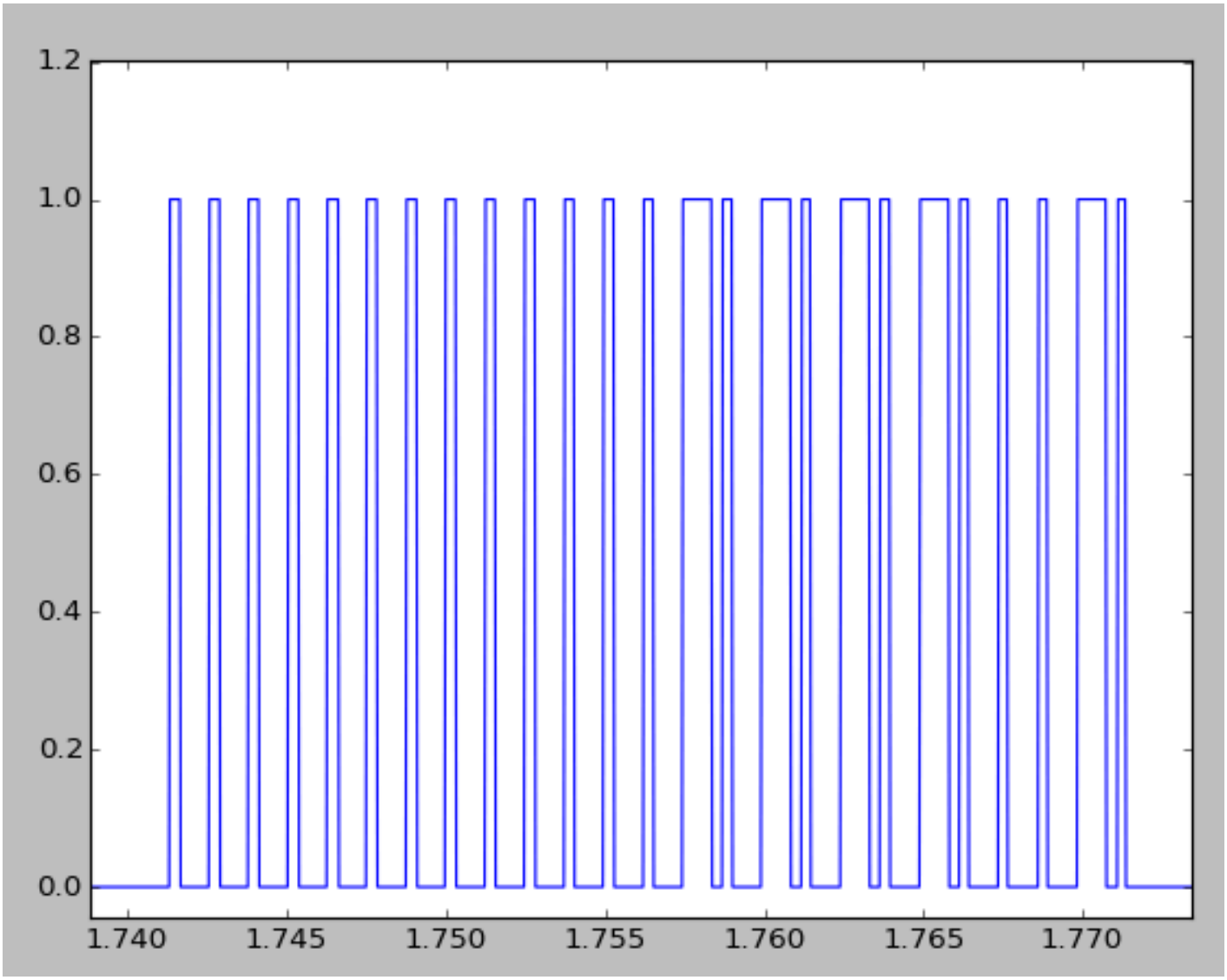
The first thing to do here is wire the pins on your receiver to the correct GPIO pins on the Pi. The receiver unit has four pins, but only three of them are needed. I think both of the central pins give the same output, so you only need to connect to one of them (unless you want to stream the received signals to two separate GPIO pins).

The image above pretty much summarises the wiring. Each pin on the receiver can be wired directly to the corresponding pin on the Pi. I use a breadboard and jumper cables to make the process a bit more elegant. Note that you can choose any GPIO data pin to connect to either of the central receiver pins. I used the pin marked as '23' on my Pi header.

IMPORTANT: If you connect the pin marked '3v3' in the above image to a higher voltage pin on the Pi (e.g. 5v), you will probably damage the Pi as the GPIO pins cannot tolerate voltages above 3v3. Alternatively, you can power it with 5v and set up a voltage divider to send a safe voltage to the DATA pin.

The range of the receiver will not be very large at this voltage, especially if an antenna is not connected. However, you don't need a long range here - as long as the receiver can pick up the signals from the handset when they are held right next to each other, that is all we need.

Step 2: Sniffing the Handset Codes



Now that your receiver is wired up to the Pi, you can start the first exciting stage of this project - the sniff. This involves using the attached Python script to record the signal transmitted by the handset when each button is pressed. The script is very simple, and I'd highly recommend you have a look at it before you run it - after all, the point of this project is that you won't just blindly run someone else's code!

Before you start this process, you will need to make sure you have the Python libraries needed to run the sniffer script. They are listed at the top of the script:

```
from datetime import datetime
import matplotlib.pyplot as pyplot
import RPi.GPIO as GPIO
```

The *RPi.GPIO* and *datetime* libraries were included with my Raspbian distribution, but I had to install the *matplotlib* library as follows:

```
sudo apt-get install python-matplotlib
```

This library is a commonly used graph plotting library that is very useful even outside of this project, so installing it definitely can't hurt! Once your libraries are up to date, you are ready to start recording data. Here's how the script works:

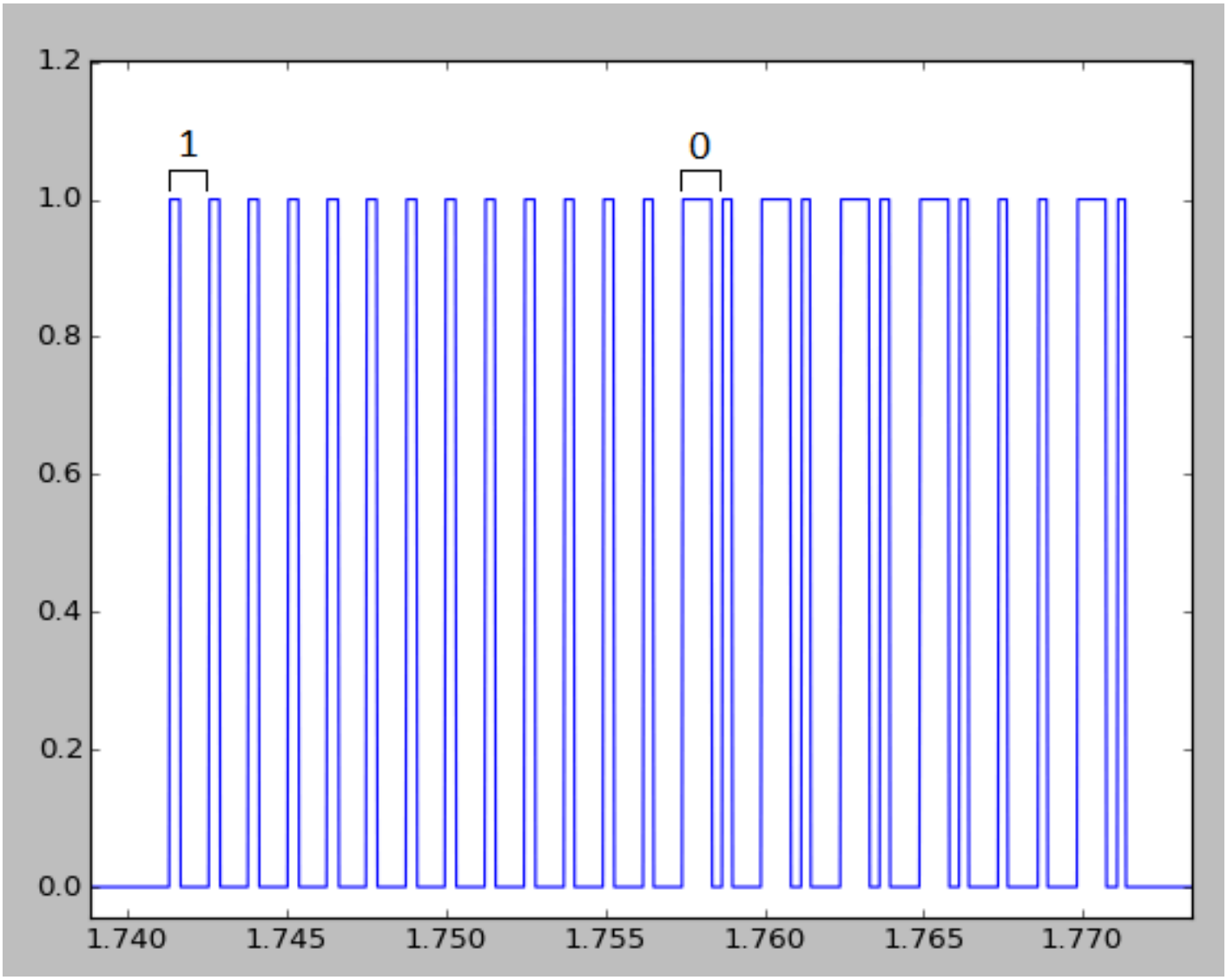
When it is run (using the command 'python ReceiveRF.py'), it will configure the defined GPIO pin as a data input (pin 23 by default). It will then continually sample the pin and log whether it is

receiving a digital 1 or 0. This continues for a set duration (5 seconds by default). When this time limit is reached, the script will stop recording data and will close off the GPIO input. It then performs a little post-processing and plots the received input value against time. Again, if you have questions about what the script is doing, you can probably answer them yourself after looking at how it works. I have tried to make the code as readable and simple as possible.

What *you* need to do is look out for when the script indicates that it has **Started recording**. Once this message appears, you should press and hold one of the buttons on the handset for about a second. Be sure to hold it close to the receiver. Once the script has finished recording, it will use matplotlib to plot a graphical waveform of the signal it has received during the recording interval. Please note, if you are connected to your Pi using an SSH client such as PuTTY, you will also need to open an X11 application to allow the waveform to display. I use xMing for this (and for other things such as remote-desktopping into my Pi). To allow the plot to be displayed, simply start xMing before you run the script and wait for the results to appear.

Once your matplotlib window appears, the area of interest within the plot should be pretty obvious. You can use the controls at the bottom of the window to zoom in until you are able to pick out the highs and lows of the signal transmitted by the handset while the button was being held down. See the above image for an example of a complete code. The signal will probably consist of very short pulses separated by similar periods of time where no signal is received. This block of short pulses will probably be followed by a longer period where nothing is received, after which the pattern will repeat. Once you have identified the pattern belonging to a single instance of the code, take a screenshot like that at the top of this page, and continue to the next step to interpret it.

Step 3: Transcribing the Resulting Signal



Now that you have identified the block of periodic highs and lows corresponding to a particular button's signal, you will need a way of storing and interpreting it. In the above signal example, you will notice that there are only two unique patterns that make up the whole signal block. Sometimes you see a short high followed by a long low, and sometimes it's the opposite - a long high followed by a short low. When I was transcribing my signals, I decided to use the following naming convention:

1 = short_on + long_off
0 = long_on + short_off

Look again at the labelled waveform, and you will see what I mean. Once you have identified the equivalent patterns in your signal, all you have to do is count the 1's and 0's to build up the sequence. When transcribed, the above signal can be written as follows:

```
111111111111010101011101
```

Now you just need to repeat this process to record and transcribe the signals corresponding to the other buttons on your handset, and you have completed the first part of the process!

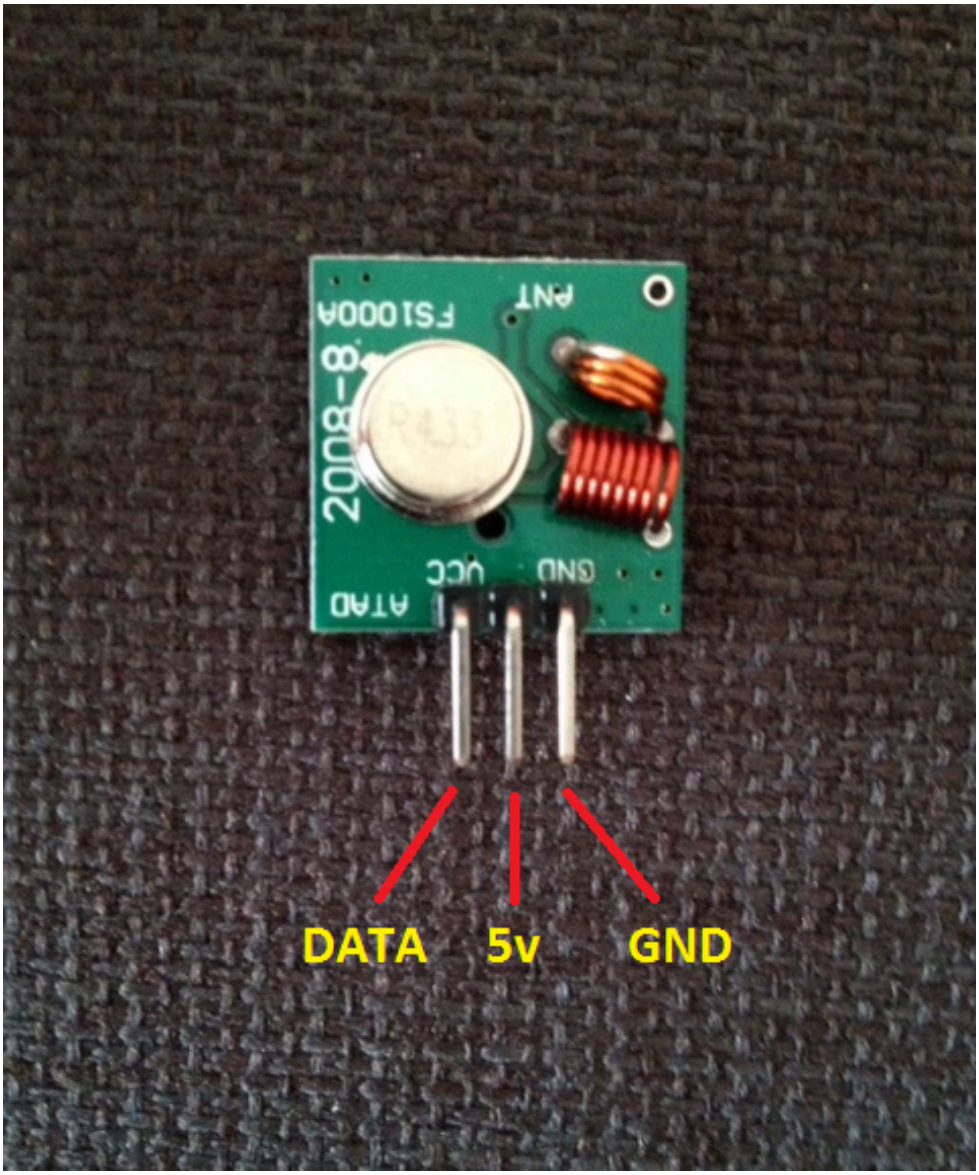
Before you can re-send the signals using the transmitter, there is a little more work to do. The timing between the highs and lows corresponding to a 1 or a 0 is very important, and you need to make sure that you know how long a 'short_on' or a 'long_off' actually lasts. For my codes, there were three pieces of timing information I needed to extract in order to replicate the signals:

- The duration of a 'short' interval, i.e. the beginning of a 1 or the end of a 0.
- The duration of a 'long' interval, i.e. the end of a 1 or the beginning of a 0.
- The duration of an 'extended' interval. I noticed that when I held a button down on the handset, there was an 'extended_off' period between each repeated instance of the signal block. This delay is used for synchronisation and has a fixed duration.

To determine these timing values, you can use the zoom function on the matplotlib window to zoom all the way in and place the cursor over the relevant parts of the signal. The cursor location readout at the bottom of the window should allow you to determine how wide each part of the signal is that corresponds to a long, short or extended interval. Note that the x-axis of the plot represents time, and the x component of the cursor readout is in units of seconds. For me, the widths were as follows (in seconds):

- short_delay = 0.00045
- long_delay = 0.00090 (twice as long as a 'short')
- extended_delay = 0.0096

Step 4: Setting Up the Transmitter Unit



Once you have collected your codes and timing data, you can disconnect your receiver unit as you will no longer need it. You can then wire up the transmitter directly to the relevant Pi GPIO pins as shown in the above image. I've found that the pins on the transmitter units are labelled, which makes the process easier.

In this case, it is OK to power the unit using the 5v supply from the Pi as the DATA pin will not be sending signals to the Pi, only receiving them. Also, a 5v power supply will provide more transmission range than using the 3v3 supply. Again, you can connect the DATA pin to any appropriate pin on the Pi. I used pin 23 (the same as for the receiver).

Another thing I'd recommend doing is adding an antenna to the small hole on the top right of the transmitter. I used a 17cm long piece of straight wire. Some sources recommend a coiled wire of similar length. I'm not sure which is better, but the straight wire provides enough range for me to turn the sockets on/off from any location in my small flat. It is best to solder the antenna, but I just removed some of the plastic from the wire and wrapped the copper through the hole.

Once the transmitter is wired up, that's all the hardware setup done! The only thing left to do now is set your sockets up around the house and have a look at the transmitter program.

Step 5: Transmitting Signals Using the Pi

This is where the second Python script comes in. It is designed to be just as simple as the first, if not more so. Again, please download it and look over the code. You will need to edit the script to transmit the correct signals according to the data you recorded in step 3, so now's a good time to have a quick glance at it.

The libraries needed to run this script were all pre-installed on my Pi, so no further installation was needed. They are listed at the top of the script:

```
import time
import sys
import RPi.GPIO as GPIO
```

Underneath the library imports is the information you will have to edit. Here is how it looks by default (this is the information corresponding to my sockets as determined using step 3):

```
a_on = '111111111111010101011101'
a_off = '111111111111010101010111'
b_on = '111111111101110101011101'
b_off = '111111111101110101010111'
c_on = '111111111101011101011101'
c_off = '111111111101011101010111'
d_on = '111111111101010111011101'
d_off = '111111111101010111010111'
short_delay = 0.00045
long_delay = 0.00090
extended_delay = 0.0096
```

Here we have eight code strings (two for each pair of on/off buttons on my handset - you may have more or fewer codes) followed by the three pieces of timing information also determined in step 3. Take the time to make sure you have entered this information correctly.

Once you're happy with the codes/delays you've entered into the script (you can rename the code string variables if you like), you are pretty much ready to try out the system! Before you do, take a look at the transmit_code() function in the script. This is where the actual interaction with the transmitter occurs. This function expects one of the code strings to be sent in as an argument. It then opens up the defined pin as a GPIO output and loops through every character in the code string. It then turns the transmitter on or off according to the timing information you entered to build up a waveform matching the code string. It sends each code multiple times (10 by default) to reduce the chance of it being missed, and leaves an extended_delay between each code block, just like the handset.

To run the script, you can use the following command syntax:

```
python TransmitRF.py code_1 code_2 ...
```

You can transmit multiple code strings with a single run of the script. For example, to turn sockets (a) and (b) on and socket (c) off, run the script with the following command:

```
python TransmitRF.py a_on b_on c_off
```

Step 6: A Note on Timing Accuracy

As mentioned, the timing between the transmitted on/off pulses is quite important. The TransmitRF.py script uses python's time.sleep() function to build up the waveforms with the correct pulse intervals, but it should be noted that this function is not entirely accurate. The length for which it causes the script to wait before executing the next operation can depend on the processor load at that given instant. That is another reason why TransmitRF.py sends each code multiple times - just in case the time.sleep() function is not able to properly construct a given instance of the code.

I have personally never had issues with time.sleep() when it comes to sending the codes. I do however know that my time.sleep() tends to have an error of about 0.1ms. I determined this using the attached SleepTest.py script which can be used to give an estimate of how accurate your Pi's time.sleep() function is. For my particular remote-controlled sockets, the shortest delay I needed to implement was 0.45ms. As I said, I haven't had issues with non-responsive sockets, so it seems like 0.45 ± 0.1ms is good enough.

There are other methods for ensuring that the delay is more accurate; for example, you could use a dedicated PIC chip to generate the codes, but stuff like that is beyond the scope of this tutorial.

Step 7: Conclusion

This project has presented a method for controlling any electrical appliance using a Raspberry Pi and a set of 433MHz remote-controlled sockets, with a focus on simplicity and transparency. This is the most exciting and flexible project that I have used my Pi for, and there are limitless applications for it. Here are some things I can now do thanks to my Pi:

- Turn on an electric heater next to my bed half an hour before my alarm goes off.
- Turn the heater off an hour after I've gone to sleep.
- Turn my bedside light on when my alarm goes off so that I don't fall back to sleep.
- and many more...

For most of these tasks, I use the *crontab* function within Linux. This allows you to set up automatic scheduled tasks to run the TransmitRF.py script at specific times. You can also use the Linux *at* command to run one-off tasks (which, for me, needed to be installed separately using 'sudo apt-get install at'). For example, to turn my heater on half an hour before my alarm goes off the next morning, all I need to do is type:

```
at 05:30
python TransmitRF.py c_on
```

You could also use this project in conjunction with my Dropbox home monitoring system to control appliances over the internet! Thanks for reading, and if you would like to clarify something or share your opinion, please post a comment!