LINUXVOICE TUTORIAL

RASPBERRY PI: BUILD AN EMERGENCY BEACON

LES POUNDER

Combine simple Python modules with hardware programming to build your own emergency distress beacon.

WHY DO THIS?

- Keep relatively safe from natural disasters.
- Program components connected to the Raspberry Pi's GPIO pins.
- Learn code concepts including loops, data storage and conditional statements.

YOU WILL NEED:

- Raspberry Pi (Model A or B can be used).
- Battery with integrated solar cell (or you could use the Pi powered from the mains).
- PiGlow (Available from Pimoroni.com).
- Buzzer/piezo speaker.
- Soldering iron (optional – I've breadboarded the example diagram for this tutorial).
- Jumper wire (female to male, from Pi to breadboard and male to male for breadboard connections.
- 100 ohm resistor
- Momentary switch (push button).
- Breadboard.
- Insulation tape.
- Micro USB to USB lead (to power the Pi).
- 20cm of wire (shielded, but you could use a female to male jumper wire).
- An FM radio tuned in to 103.3MHz.

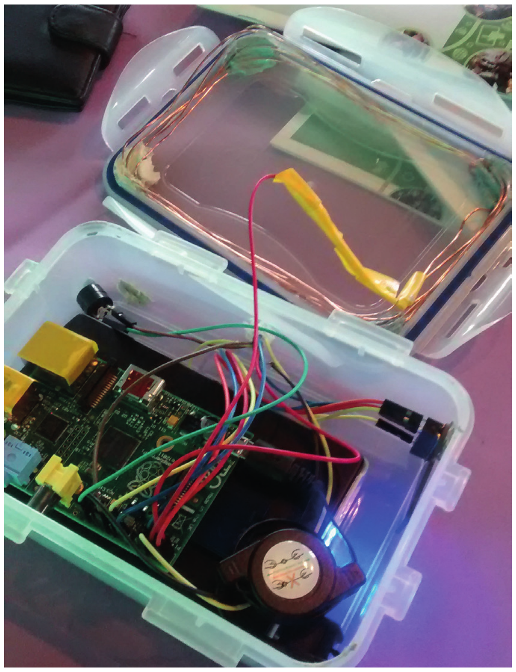
The background to this project is that I've been working with a class at Mereside Primary School in Blackpool. The children were learning about natural disasters such as tsunamis and earthquakes. During the course of their lessons they learnt that one of the first issues faced by the victims was a loss of communication as mobile phone towers were quickly damaged. The children worked as a team to understand the impact that this would have and how they could make a difference.

Their idea was to create a beacon that attracts help in three ways.

- An FM radio transmitter, that can be tuned to work on many different frequencies.
- An LED unit, to visually attract people to the beacon.
- A buzzer, to attract people using audio output.

The beacon must be completely self supporting and have its own self-charging power source. To accomplish this we found a cheap USB battery pack with a built-in solar cell on Amazon, but for the purposes of this tutorial you can just plug into the mains.

To keep the project as simple as possible we'll use only one method of input, which is a single push button that when pressed will launch the Python code. Finally, the project must be weatherproof, and at this prototype stage the best solution was every Raspberry Pi hacker's best friend, a plastic lunchbox.


The finished PiBeacon project encased inside its protective lunchbox shell.

The PiBeacon was entered into PA Consulting's Pi Awards event on 2 April 2 2014. I am proud to say that my team came second in their year group and really proved how far they had come in such a short time. I'd like to say a very big "well done" to the hackers from Mereside Primary School.

Pin reference

Throughout this tutorial, I will refer to the GPIO pins of the Raspberry Pi via their board reference. With pin 1 being the top-left pin, nearest the SD card slot, and pin 2 being directly to pin 1's right. Please refer to the guide, right, for the location of 3.3V, 5V and ground pins. Don't use use these pins unless instructed to do so, but you can use any other pin in your program.

The only user with permission to use the GPIO pins in Raspbian is root, so in order for you to use the GPIO in Idle, open a terminal and type

```
sudo idle
```

Type in your password (by default in Raspbian this is **raspberry**) and press Enter. In a few seconds the editor for our Python code will be on the screen. By launching Idle in this manner you will be able to access the GPIO pins – just remember to open any Python programs using the File > Open menu option.

Building the project

This build is not complex but it does have four areas that need to be carefully wired together. If you are unsure about your wiring, please ask someone to check before you connect any power to your Pi or attached components.

- **Antenna** This is the most simple section of the build. All you will need to do is attach a maximum of 20cm of wire to pin 4 of your Raspberry Pi. The greater the length of wire, the larger your antenna, but also the greater your signal may become. Please refer to the section on radio transmissions for safety instructions.
- **Button** I used a momentary switch, attached to pin 8 to act as the only method of input. The switch is attached to 3V power from pin 1 and a resistor is used inline with Ground to ensure that the switch does not accidentally trigger from a slight press.
- **Buzzer** A simple buzzer is attached to pin 26 and Ground (pin 20). This buzzer is used as an audio output that will send a message in Morse Code.

- **PiGlow** Rather than use just one LED, we used 18 super-bright LEDs courtesy of Pimoroni's tiny board.

Normally this board covers all the GPIO pins, but thanks to a phone call with Jon and the team we worked out the minimum number of pins necessary, and these are as follows:

- **Pin 1** 3V3 Logic level voltage.
- **Pin 2** 5V LED source current.
- **Pin 3** SDA i2c Communications.
- **Pin 5** SCL i2c Communication.
- **Pin 14** Ground (GND).
- **Pin 17** Logic level voltage.

Remember when inserting the wires into the PiGlow that you will need to work out where each pin should be inserted. When the board is attached to the GPIO, the "P" of PiGlow should be near the SD card slot. Once you have located Pin 1 of PiGlow, insert a red jumper wire to help you remember that Pin 1 is 3.3V power, and refer to the diagram for more information.

Set up PiGlow, i2c and PiFM

PiGlow uses something called i2c to control the 18 onboard LEDs, and by using i2c PiGlow is able to use far fewer wires than a conventional series of 18 LED would require. I2c was developed by Philips in the 1980s as a means to send data to multiple devices using the a minimal number of wires. It's useful, but the Raspberry Pi does not have i2c set up by default.

To set up i2c on your Raspberry Pi, download a copy of Michael Rimmican's excellent setup script from GitHub: **https://github.com/heeed/pi2c**.

Open a terminal, navigate to where you downloaded the file and then used **chmod** to make it executable:

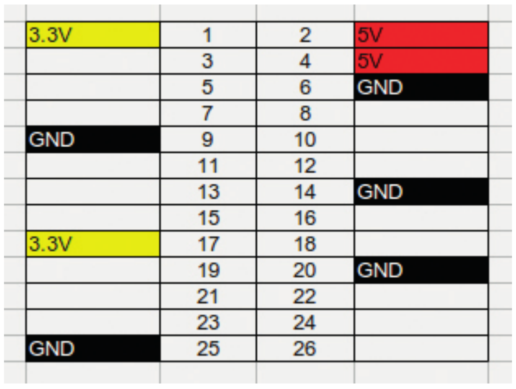
```
chmod +x pi2c.sh
```

Then run the script using **sudo** or as root:

```
sudo .pi2c.sh
```

After a few minutes your Pi will be reconfigured to use i2c; at this time it would be prudent to reboot your Pi to ensure that the configuration is complete.

Now you will need to download the Python library for PiGlow, and luckily Jason Barnett has created a great library for us to use, which is available here: **https://github.com/Boeeerb/PiGlow**.



3.3V	1	2	5V
	3	4	5V
	5	6	GND
	7	8	
GND	9	10	
	11	12	
	13	14	GND
	15	16	
3.3V	17	18	
	19	20	GND
	21	22	
	23	24	
GND	25	26	

Pin diagram for Model B Raspberry Pi.

For this project, **piglow.py** will need to be in the same directory as our **beacon.py** code. With these files downloaded, try out some of the examples to ensure that your PiGlow board is working correctly.

Our final requirement is PiFM, a library of code that we can easily drop in to our project to add an FM transmitter. You can download the library from **www.icrobotics.co.uk/wiki/index.php/Turning_the_Raspberry_Pi_Into_an_FM_Transmitter**. Extract the files to the same directory as your **beacon.py** and **piglow.py** files. I kept the example audio file – the Star Wars theme – as the audio to play over the airwaves. You could also use any 16-bit mono WAV file.

Coding the project

You can download the code for this project from my GitHub repository: **https://github.com/lesp/PiBeacon**.

We coded this project in Python 2.7 due to its mature collection of libraries and documentation. Libraries enable us to reuse code that other people have written. I used four libraries in my code: **PiFM** to control the radio transmitter; **RPI.GPIO** for GPIO access; **time** to add a delay function to my code; and **PiGlow** to control the PiGlow LED board.

Import the libraries into our code like so:

```
import PiFm
import RPi.GPIO as GPIO
from piglow import PiGlow
from time import *
```

LV PRO TIP

Project files for the PiBeacon are available at https://github.com/lesp/PiBeacon .

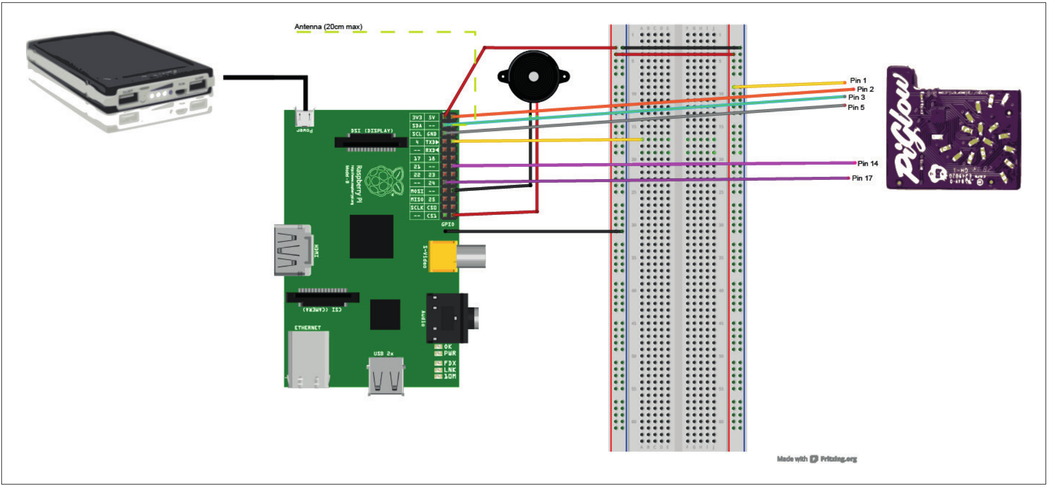


Diagram of the completed setup. Remember to pay careful attention to the GPIO pins for PiGlow.

Next I created two variables: **button_pin** and **buzzer**, and in each one I stored the value of the GPIO pin used for each, respectively 8 and 26. Variables are great, as they enable our program to retain information and act as a data storage system. Variables are used to replace hard coded values in our code. For example I could've used the integers 8 and 26 throughout my code, but if I wanted to change those numbers to something else, then I would have to go through every line of code to make the change. Because we're using a variable, we can simply change the value of that variable once and that change is reflected whenever we refer to the variable name.

"Variables enable our program to retain information and act as a data storage system."

In order to use the GPIO we need to tell Python how we want to use it:

```
GPIO.setmode(GPIO.BOARD)
```

This tells the Pi that I wish to use the numbering as per the earlier diagram.

```
GPIO.setup(button_pin , GPIO.IN)
GPIO.setup(buzzer , GPIO.OUT)
```

These two lines tell the Pi that our button, attached to pin 8, is an input and that our buzzer on pin 26 is an output. Remember that the variables **button_pin** and **buzzer** both contain the pin reference for each.

To make it easier for me to use the PiGlow function, **PiGlow()**, I next create a variable called **piglow**:

```
piglow = PiGlow()
```

Later on I use the code

```
piglow.all(128)
```

to set all of the LEDs to half brightness, but I'll cover that in more detail later.

Now we come to the main part of the program. In order to control the program we use an infinite loop, which in Python is '**while True:**'. This is the simplest kind of loop, and for the purpose of this project, is the most practical. Any code contained in this loop will run over and over until it is stopped.

The next line is a conditional statement that checks to see if the button has been pressed. This, coupled with our infinite loop, enables the program to constantly check for user input via the button:

```
while True:
  if GPIO.input(button_pin)==1:
```

So now that we have a conditional statement, what do we want it to do if the condition is true? Well firstly I want it to print "Button Pressed", for debugging purposes, so that I can see that the code has worked. Then I want the code to start PiFm and play the Star Wars theme. The code is as so:

```
          print("BUTTON PRESSED")
          PiFm.play_sound("/home/pi/sound.wav")
```

Once PiFm has finished playing the audio I want to then start a loop that iterates three times. Inside this loop I want the buzzer and PiGlow to provide output in the form of Morse code – more specifically the internationally recognised SOS message (... - - - ...).

To create the iterated loop I used a 'for' loop with a range that starts at zero and ends before three, so it goes 0,1,2. A 'for' loop is a loop that will iterate through a list, range or tuple until complete, giving us a the limited number of loops that we require. This gives us the three iterations that we require. Here's the code:

```
for i in range(0,3):
```

You might be wondering where the **i** came from? Well, this is a variable that we've declared "on the fly". You could replace **i** with **x**, **y** or **z** if you wished. The **range(0,3)** bit instructs the for loop to start at 0 and count to 2, as 3 is the limit of our range. By counting from 0 to 2 we have 3 loops.

Send signals

Now to make the buzzer and PiGlow come to life. We have to tell the GPIO to send electricity to the buzzer, and to do that we use the Boolean term "True" to say that we want to turn the power on. Remember I earlier set up the GPIO pin 26 as an output and used a variable called **buzzer** to represent this. So now to send the power to the pin I use the following code.

```
GPIO.output(buzzer, True)
```

To turn the buzzer off I change the **True** to **False**.

For PiGlow it is a little bit different but by no means a challenge. To illuminate all of the LED on the board I use **piglow.all**. Now as you will see in the code there is a number contained in brackets. This number is the brightness of the LED, with 0 being off and 255 being full brightness. I used 128, which is the halfway point between the two. A word of warning: PiGlow is extremely bright, so be careful with your eyes. Here's how to turn the LED on.

```
piglow.all(128)
```

Radio transmissions

This project uses a Python library called PiFM, which is available from **www.icrobotics.co.uk/wiki/index.php/Turning_the_Raspberry_Pi_Into_an_FM_Transmitter**. This library is what powers the PiBeacon's radio transmissions. It's very versatile, with extra functionality such as broadcasting in stereo and using a microphone connected to your Pi to broadcast live audio over the airwaves.

Transmitting radio signals is not to be taken lightly, and great care should be taken when using this project. Make sure that you are not operating on any frequencies that are reserved for emergency services or aviation, otherwise you will get in trouble with the authorities. Please refer to the official guidance available from **http://stakeholders.ofcom.org.uk/enforcement/spectrum-enforcement/law**, as there are certain regulations that must be followed when using radio transmitters.

The FM transmitter is also very powerful – so powerful in fact that if used incorrectly it can cause interference. Best practice would be to reduce the length of wire used in the build so that the effect is localised. The use of SOS audio messages or SOS Morse code is also not to be broadcast on the radio spectrum, so please just play the theme from Star Wars or Transformers and save the emergency for the real thing.

If you are still unsure then the best resource to use is your local amateur radio group (basically a LUG for those interested in radio related topics). A quick Google search will find your local group, who will be able to answer any questions that you may have. Remember: hack responsibly.

And to turn off the LED we create a new line, which is identical to before but with the (128) changed to (0).

To control which letter is being communicated in Morse I used a delay function, which in Python is called **sleep()**. To create a dot, which is a short beep in Morse I kept the delay to a minimum and set it to 0.5, which is half a second. To create a dash, which is a longer sound, I used a delay of 1, which is 1 second. In code the delays look like this.

```
sleep(0.5) # For a DOT
sleep(1) # For a DASH
```

The last section of code is the else statement. When using a conditional such as **if**, we can use an **else** statement to capture any unexpected conditions. In this case the **else** statement is used when no user input is detected, it will print "Waiting for input" over and over. As soon as user input is detected, the **else** condition is no longer true and the **if** condition, when the button is pressed, is now true.

Before you test your project it would be prudent to check all of the connections and wiring before you start the program. Once you're happy that everything is as it should be, run your code. You can do this in Idle via the Run > Run Module menu item.

Grab your radio and tune in to 103.3MHz FM, which is the default frequency that we will be using for this project. You should now see the shell printing "Waiting for input" so go ahead and press the button. A moment later you should hear the theme from Star Wars playing through your FM radio. A few minutes later, once the music has finished, your buzzer and PiGlow will start emitting a message in Morse code. Congratulations: you have built a working PiBeacon!

Bonus points – change your message

In this project we use **sleep()** to control the delay for our beeps and flashes, with half a second for a dot and one second for a dash. So using just dots and dashes we can communicate text and numbers.

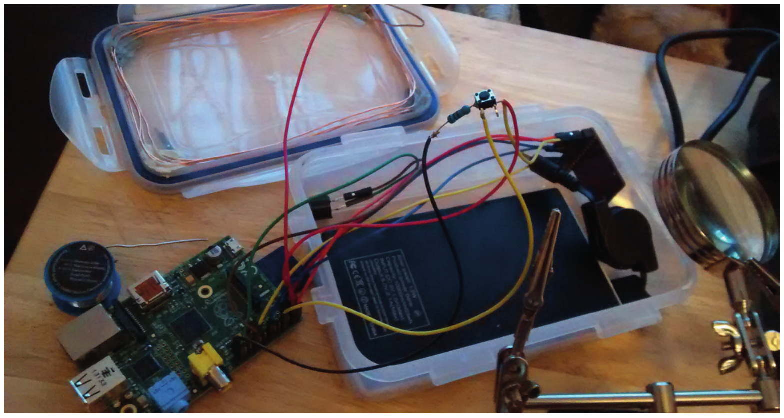
Instead of broadcasting SOS, let's say "Linux Voice". First of all we'll refer to a chart of Morse Code.

L	**DOT DASH DOT DOT**
I	**DOT DOT**
N	**DASH DOT**
U	**DOT DOT DASH**
X	**DASH DOT DOT DASH**
V	**DOT DOT DOT DASH**
O	**DASH DASH DASH**
I	**DOT DOT**
C	**DASH DOT DASH DOT**
E	**DOT**

Why don't you try altering the example code to output this message instead?

Here's how to write L in Morse using Python

```
#The letter L in Morse code.
#DOT
GPIO.output(buzzer, True)
piglow.all(128)
sleep(0.5)
GPIO.output(buzzer, False)
piglow.all(0)
#End of DOT, now a 1 second pause
sleep(1)
#DASH
GPIO.output(buzzer, True)
piglow.all(128)
sleep(1)
GPIO.output(buzzer, False)
piglow.all(0)
sleep(1)
#End of DASH, now a 1 second pause
#DOT
GPIO.output(buzzer, True)
piglow.all(128)
sleep(0.5)
GPIO.output(buzzer, False)
piglow.all(0)
#End of DOT, now a 1 second pause
sleep(1)
#DOT
GPIO.output(buzzer, True)
piglow.all(128)
sleep(0.5)
GPIO.output(buzzer, False)
piglow.all(0)
#End of DOT, now a 1 second pause
sleep(1)
```


Assembling the final prototype and soldering the connections was essential to qualify for the PA Consulting competition.

So what have we accomplished here?

- We have built the hardware that powers our project.
- Using Python and libraries from external sources we have created the code that controls the components in the beacon.

We also used programming concepts such as Loops, to control the flow of our program and to repeat repetitive tasks; variables, to store the values of GPIO pins in one section of code, enabling us to quickly make changes to one value that are reflected throughout the program; and conditionals to control the flow of our program by using logic. The next step is to play with the lights on the PiGlow – you could even create an animation.

Les Pounder is a maker and hacker specialising in the Raspberry Pi and Arduino. Les travels the UK training teachers in the new computing curriculum and Raspberry Pi.