

Python Radio 23: Bluetooth Low Energy

Simon Quellen Field

Simon Quellen Field

Follow

6 min read

.

Sep 10, 2024

Listen

Share

More

Designed by committee...

Press enter or click to view image in full size

MidJourney

Wifi is not too difficult to set up. Micropython is well supplied with classes built-in to handle Wifi. A web server may be a fair chunk of code, but it is easy to set up and configure. Bluetooth is another matter. Bluetooth was designed to handle a large number of use cases, and as a result is quite complex, and doing something simple is often quite tedious.

If we wanted to send temperature data from one computer to another using the HC-12, we would simply open up a UART and send the data.

This is what you have to do with Bluetooth:

```
Import bluetooth
```

```
Import random
```

```
Import struct
```

```
Import time
```

```
From ble_advertising import advertising_payload
```

```
From micropython import const
```

```
_IRQ_CENTRAL_CONNECT = const(1)
```

```
_IRQ_CENTRAL_DISCONNECT = const(2)
```

```
_IRQ_GATTS_INDICATE_DONE = const(20)
```

```
_FLAG_READ = const(0x0002)
```

```
_FLAG_NOTIFY = const(0x0010)
```

```
_FLAG_INDICATE = const(0x0020)
```

```
# org.bluetooth.service.environmental_sensing
```

```
_ENV_SENSE_UUID = bluetooth.UUID(0x181A)
```

```
# org.bluetooth.characteristic.temperature
```

```
_TEMP_CHAR = (
```

```
Bluetooth.UUID(0x2A6E),
```

```
_FLAG_READ | _FLAG_NOTIFY | _FLAG_INDICATE,
```

```
)
```

```
_ENV_SENSE_SERVICE = (
```

```
_ENV_SENSE_UUID,
```

```
(_TEMP_CHAR,),
```

```
)
```

```
# org.bluetooth.characteristic.gap.appearance.xml
```

```
_ADV_APPEARANCE_GENERIC_THERMOMETER = const(768)
```

```
Class BLETemperature:
```

```
Def __init__(self, ble, name="mpy-temp"):
```

```
Self._ble = ble
```

```
Self._ble.active(True)
```

```
Self._ble.irq(self._irq)
```

```

((self._handle,)) = self._ble.gatts_register_services((_ENV_SENSE_SERVICE,))
Self._connections = set()
Self._payload = advertising_payload(
Name=name, services=[_ENV_SENSE_UUID], appearance=_ADV_APPEARANCE_GENERIC_THER
)
Self._advertise()
Def _irq(self, event, data):
# Track connections so we can send notifications.
If event == _IRQ_CENTRAL_CONNECT:
Conn_handle, _, _ = data
Self._connections.add(conn_handle)
Elif event == _IRQ_CENTRAL_DISCONNECT:
Conn_handle, _, _ = data
Self._connections.remove(conn_handle)
# Start advertising again to allow a new connection.
Self._advertise()
Elif event == _IRQ_GATTS_INDICATE_DONE:
Conn_handle, value_handle, status = data
Def set_temperature(self, temp_deg_c, notify=False, indicate=False):
# Data is sint16 in degrees Celsius with a resolution of 0.01 degrees Celsius.
# Write the local value, ready for a central to read.
Self._ble.gatts_write(self._handle, struct.pack("<h", int(temp_deg_c * 100)))
If notify or indicate:
For conn_handle in self._connections:
If notify:
# Notify connected centrals.
Self._ble.gatts_notify(conn_handle, self._handle)
If indicate:
# Indicate connected centrals.
Self._ble.gatts_indicate(conn_handle, self._handle)
Def _advertise(self, interval_us=500000):
Self._ble.gap_advertise(interval_us, adv_data=self._payload)
We have to create a service and advertise that it is a sensor and returns temperature data. We must
indicate that it can be read, it can notify, and it can indicate.
We give it a name "mpy-temp" so that other devices can find its advertisements and pair with it. We
have to track connections and disconnections.
In the micropython library is a package called aioble which makes much of the BLE processing a
little simpler. You can find it at .
Here is the temperature sensor example from that library:
Import sys
Sys.path.append("")
From micropython import const
Import uasyncio as asyncio
Import aioble
Import bluetooth
Import random
Import struct
# org.bluetooth.service.environmental_sensing
_ENV_SENSE_UUID = bluetooth.UUID(0x181A)
# org.bluetooth.characteristic.temperature

```

```

_ENV_SENSE_TEMP_UUID = bluetooth.UUID(0x2A6E)
# org.bluetooth.characteristic.gap.appearance.xml
_ADV_APPEARANCE_GENERIC_THERMOMETER = const(768)
# How frequently to send advertising beacons.
_ADV_INTERVAL_MS = 250_000
# Register GATT server.
Temp_service = aioble.Service(_ENV_SENSE_UUID)
Temp_characteristic = aioble.Characteristic(
Temp_service, _ENV_SENSE_TEMP_UUID, read=True, notify=True
)
Aioble.register_services(temp_service)
# Helper to encode the temperature characteristic encoding (sint16, hundredths of a degree).
Def _encode_temperature(temp_deg_c):
Return struct.pack("<h", int(temp_deg_c * 100))
# This would be periodically polling a hardware sensor.
Async def sensor_task():
T = 24.5
While True:
Temp_characteristic.write(_encode_temperature(t))
T += random.uniform(-0.5, 0.5)
Await asyncio.sleep_ms(1000)
# Serially wait for connections. Don't advertise while a central is
# connected.
Async def peripheral_task():
While True:
Async with await aioble.advertise(
_ADV_INTERVAL_MS,
Name="mpy-temp",
Services=[_ENV_SENSE_UUID],
Appearance=_ADV_APPEARANCE_GENERIC_THERMOMETER,
) as connection:
Print("Connection from", connection.device)
Await connection.disconnected()
# Run both tasks.
Async def main():
T1 = asyncio.create_task(sensor_task())
T2 = asyncio.create_task(peripheral_task())
Await asyncio.gather(t1, t2)
Asyncio.run(main())

```

It isn't a lot smaller, but it is perhaps a little easier to understand and modify.

The aioble library has an example file server as well, so your devices can store and serve files to one another, or to phones, tablets, laptops, or other bluetooth clients.

To get the aioble library onto your ESP32, run this small main.py program:

```

From mip import install
From network import WLAN, STA_IF
Def main():
Wlan = WLAN(STA_IF)
Wlan.active(True)
Wlan.connect("BirdfarmOffice2","12345678") # SSID and password
Print(wlan.isconnected())

```

```
Install("aioble")
```

```
Main()
```

You would use your own Wifi SSID and password of course, instead of mine.

Or, you can run the following version:

```
From connect import Connect
```

```
From mip import install
```

```
Def main():
```

```
Con = Connect("Get Library")
```

```
Con.reconnect()
```

```
Install("aioble")
```

```
Main()
```

It uses the connect.py module from the previous project and the network.cfg file from there. It connects to the Internet using your local Wifi, so that it can then download the library from micropython.org.

There are many libraries there that you can download.

Once a library is downloaded, it resides on your flash filesystem until you erase the chip, so you only need to run that program once.

Suppose we wish to know when a particular Bluetooth-enabled device is in the vicinity. This probably won't work to track people, or tell you when someone special has arrived home or left home, as phones and watches change their addresses often to prevent this. But two of our own devices can find each other this way and tell roughly how far away they are from one another. A program to do that would look like this:

```
From time import time, localtime
```

```
From ntptime import settime
```

```
From connect import Connect
```

```
From uasyncio import get_event_loop, sleep
```

```
TIMEZONE = -8 * 60 * 60
```

```
Devices = {}
```

```
Async def lookie():
```

```
From aioble import scan
```

```
Async with scan(duration_ms=5000, interval_us=30000, window_us=30000, active=True) as scanner:
```

```
Async for result in scanner:
```

```
My_time = localtime(time() + TIMEZONE)
```

```
Time_str = my_time
```

```
Devices[str(result.device.addr_hex())] = [result.rssi, result.device.addr_hex(), time(),  
result.name()]
```

```
Lookfor = "5a:5d:bb:ef:0c:64"
```

```
Async def printem():
```

```
For x in devices:
```

```
If x == lookfor:
```

```
Print("* ", end="")
```

```
Else:
```

```
Print(" ", end="")
```

```
Print(x, end=": ")
```

```
Count = 0
```

```
For y in devices[x]:
```

```
If count == 1:      # address already printed
```

```
Pass
```

```
Elif count == 2:    # The timestamp
```

```
Diff = time() - y
```

```

If diff < 10:
Print("just now", end=", ")
Elif diff < 60:
Print(str(diff) + " seconds ago", end=", ")
Elif diff < 120:
Print("a minute ago", end=", ")
Elif diff < 3600:
Print(str(round(diff/60)) + " minutes ago", end=", ")
Elif diff < 7200:
Print("an hour ago", end=", ")
Elif diff < 86400:
Print(str(round(diff/3600)) + " hours ago", end=", ")
Else:
Print("long ago")
Elif count == 3:
Print(y)
Else:
Print(y, end=", ")
Count += 1
Print()
Def main():
Con = Connect("BLE_Scanner")
Con.reconnect()
# from mip import install
# install("aioble")
# import mip
Settime()
Def scan_task():
While True:
Await lookie()
Def print_task():
While True:
Await printem()
Await sleep(10)
Loop = get_event_loop()
Loop.create_task(scan_task())
Loop.create_task(print_task())
Loop.run_forever()
Loop.close()
Main()

```

The main program sets up two tasks that will run in parallel. One task scans for Bluetooth devices and records them in a dictionary.

The other task reads the dictionary and prints out the information every ten seconds.

If the target address is found, it puts an asterisk in front of the line.

The output looks like this:

```

79:3c:0c:8d:f9:a2: -74, 4 minutes ago, None
54:e9:b0:b9:8d:37: -78, a minute ago, None
55:77:11:96:f3:4a: -69, 3 minutes ago, None
45:0d:0b:7f:77:15: -77, 5 minutes ago, None
F2:69:bb:f8:fe:ba: -64, just now, Two_feba

```

F8:04:2e:86:42:ab: -50, just now, [TV] office
5a:ad:96:0f:f0:e0: -71, just now, None
50:68:34:35:88:61: -35, 3 minutes ago, None
46:3c:d9:72:c8:eb: -36, 2 minutes ago, None
54:e5:88:49:66:d1: -78, 10 minutes ago, None
D5:ef:31:87:5b:cb: -49, just now, Andy_5bcb
5b:58:f6:85:2c:49: -66, 3 minutes ago, None
76:79:7d:da:b5:e1: -69, 47 seconds ago, None
76:66:86:81:bb:3a: -78, 13 minutes ago, None
Ec:85:48:29:a0:1f: -56, just now, Echo_a01f
4e:78:cb:24:0e:f0: -74, 2 minutes ago, Galaxy Watch6 (MHAZ)
78:25:33:e4:e5:c6: -67, 37 seconds ago, None
66:03:6d:61:85:22: -36, 10 seconds ago, None
50:e7:a6:ba:00:29: -67, 2 minutes ago, None
47:e6:6c:33:0a:12: -62, 13 minutes ago, None
5a:38:26:63:55:fd: -59, 4 minutes ago, None
5b:a6:45:35:eb:1d: -78, 3 minutes ago, None
49:c9:69:59:9e:2e: -76, 4 minutes ago, None

Notice that the Galaxy Watch announces itself by name, making it easy to tell if the only person in the house with a Galaxy Watch happens to be at home. The Pixel phone and Pixel Watch do not, and the program returns None for the name.

You can find out what the address of a device is by simply holding it close to our ESP32 while we are scanning. The RSSI (Received Signal Strength Indicator) value will rise (become less negative) as the device nears.

Our program needs an Internet connection to get the correct time by way of NTP (the Network Time Protocol). We import settime() from the module ntptime to set our clock properly. Being able to do this can be quite useful to synchronize two radios, so that one will transmit to the other at an exact time, making decoding easier in some protocols.

Ble

Bluetooth Low Energy

Python Programming

Esp32

Python Radio 31: There Goes Your Money!

Using Python to see how contactless transactions work.

Simon Quellen Field

Simon Quellen Field

Follow

24 min read

.

Jan 2, 2025

Listen

Share

More

Press enter or click to view image in full size

Money flying away after a cellphone transaction.

MidJourney

Radio Frequency Identification (RFID) is all around you. Your passport has a chip in it that works without any batteries to communicate with customs and security at airports. Credit cards have them. Smartphones have them.

Android phones have Near Field Communication chips that can read and write to these devices.

When you are paying at a grocery store, you want to be sure you aren't accidentally paying for someone else's purchase, just because the register picked up a signal from your phone or card. NFC relies on Near Field Communication, which works only from a couple of centimetres away. It won't read your card unless you almost press it onto the reader.

How can it do this without a battery?

The reader emits a signal at 13.56 megahertz into a coil of wire. If a similar coil of wire is close enough, energy from the one coil will couple to the other.

The second coil is attached to a tiny chip that gets power when it is close to the first coil.

However the chip has no way to store enough power to transmit a signal. Instead, it uses a neat trick. It periodically puts a load on the power, in a bit stream that carries data.

The sending coil sees this drain on its power and reads that signal to identify the card and see what information is stored there. It's a bit like sending data to someone giving you water in a hose by opening and closing the nozzle in Morse code. They can feel the pressure change at their end. Writing is easier. The device with power just sends a signal to the chip, which then stores the information in non-volatile memory. This is typically good for over a million writes.

NFC cards are used to access locked doors at places like gyms or hotel rooms. Identity information and access permission codes are written into the cards at the front desk and the user now has free access to the facility.

There are a number of cards and tags that you can buy. As you might expect, these have to be cheap and are often sold in bulk. So I have a bunch of them:

[Press enter or click to view image in full size](#)

A selection of NFC tags and cards.

Photo by author.

My favourites are the little transparent ones, but I had trouble writing to them (except when using my smartphone). In this article, we will be using the credit card size.

For the reader, I chose this nice little board from Adafruit — the RFID-RC522.

[Press enter or click to view image in full size](#)

The RFID-RC522 module from Adafruit.

Photo by author.

We will talk to it using an ESP32 we have not used before, the Lolin S2 Mini.

I started this project using the Wemos D1 Mini (an ESP8266 board), but the driver required more RAM than it had. But the S2 Mini has a similar form factor and many advantages. It is an ESP32, so much bigger and faster. It has a USB-C port, which is much easier to deal with than the old Micro-USB. It has twice as many I/O pins. Which is great, but to fit them in, they had to make two rows, so it is not breadboard-friendly anymore.

It comes with rows of header pins you can solder on, but since that won't help breadboarding, it seems of only marginal use. Instead, I brought out my own set of socket headers (shown at the upper right). Now I can just plug in Dupont rainbow ribbon cable connectors just like on a breadboard.

[Press enter or click to view image in full size](#)

The Lolin S2 Mini ESP32 computer board.

Photo by author.

The result turned out like this:

[Press enter or click to view image in full size](#)

The RFID-RC522 connected to the Lolin S2 Mini.

Photo by author.

ESP Pin "3V3" (red) connected to 522 Pin "3.3V"

ESP Pin "17" ((green) connected to 522 Pin "MISO"

ESP Pin "11" (white) connected to 522 Pin "IRQ"

ESP pin "14" (orange) connected to 522 Pin "SCK"

ESP Pin "13" (yellow) connected to 522 Pin "MOSI"

ESP Pin "15" (blue) connected to 522 Pin "SDA"
ESP Pin "GND" (black) connected to 522 Pin "GND"
ESP Pin "17" (brown) connected to 522 Pin "RST"

The S2 Mini has 2 whole megabytes of RAM. That sounds laughable to those of us used to demanding tens of gigabytes just to surf the web, but it is a real luxury after spending weeks trying to cram a big driver into the ESP8266.

But that means you have to choose the version of Micropython that supports external RAM. See this page for the download. Hold the 0 button while powering up to get it into programming mode. The rest of the instructions for loading Micropython can be found earlier in this series, or at micropython.org.

Before launching into code, let's look at the program running for a bit.

Hit the ENTER key when the program starts, and you get:

R for Read, F for Format, E for Erase, W for Write:

Place the card on the reader and type r and ENTER.

Please place card on reader

Card 0XDA8A0EC3 [0xC3, 0x0E, 0x8A, 0xDA]

READ

```
00 S00 B0: Len 16 C3 0E 8A DA 9D 08 04 00 62 63 64 65 66 67 68 69 .....bcdefghi
01 S00 B1: Len 16 7E 01 03 E1 03 E1 03 E1 03 E1 03 E1 03 E1 ~.....
02 S00 B2: Len 16 7E 01 03 E1 03 E1 03 E1 03 E1 03 E1 03 E1 ~.....
03 S00 B3: Len 16 00 00 00 00 00 00 00 78 77 88 FF 00 00 00 00 00 .....xw.....
04 S01 B0: Len 16 68 74 74 70 73 3A 2F 2F 73 63 69 74 6F 79 73 2E .
05 S01 B1: Len 16 63 6F 6D 00 00 00 00 00 00 00 00 00 00 00 00 com.....
06 S01 B2: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
07 S01 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
08 S02 B0: Len 16 68 74 74 70 73 3A 2F 2F 70 75 72 65 66 69 78 69
09 S02 B1: Len 16 6F 6E 2E 63 6F 6D 00 00 00 00 00 00 00 00 00 on.com.....
10 S02 B2: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
11 S02 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
12 S03 B0: Len 16 68 74 74 70 73 3A 2F 2F 6E 65 74 72 6F 67 6C 79
13 S03 B1: Len 16 63 65 72 69 6E 65 2E 63 6F 6D 00 00 00 00 00 00 cerine.com.....
14 S03 B2: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
15 S03 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
16 S04 B0: Len 16 68 74 74 70 73 3A 2F 2F 62 69 72 64 66 61 72 6D
17 S04 B1: Len 16 2E 6F 72 67 00 00 00 00 00 00 00 00 00 00 00 .org.....
18 S04 B2: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
19 S04 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
20 S05 B0: Len 16 54 72 75 74 68 20 69 73 20 61 20 73 68 61 64 6F Truth.is.a.shado
21 S05 B1: Len 16 77 2C 20 68 69 73 20 70 61 73 73 69 6F 6E 20 61 w,.his.passion.a
22 S05 B2: Len 16 20 6B 69 73 73 2C 20 77 68 65 6E 20 6B 6E 6F 77 .kiss,.when.know
23 S05 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
24 S06 B0: Len 16 6C 65 64 67 65 20 69 73 20 70 6F 77 65 72 20 61 ledge.is.power.a
25 S06 B1: Len 16 6E 64 20 69 67 6E 6F 72 61 6E 63 65 20 62 6C 69 nd.ignorance.bli
26 S06 B2: Len 16 73 73 2E 00 00 00 00 00 00 00 00 00 00 00 00 ss.....
27 S06 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
28 S07 B0: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
29 S07 B1: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
30 S07 B2: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
31 S07 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
32 S08 B0: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```



```

33 S08 B1: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
34 S08 B2: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
35 S08 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
36 S09 B0: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
37 S09 B1: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
38 S09 B2: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
39 S09 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
40 S10 B0: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
41 S10 B1: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
42 S10 B2: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
43 S10 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
44 S11 B0: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
45 S11 B1: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
46 S11 B2: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
47 S11 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
48 S12 B0: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
49 S12 B1: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
50 S12 B2: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
51 S12 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
52 S13 B0: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
53 S13 B1: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
54 S13 B2: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
55 S13 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
56 S14 B0: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
57 S14 B1: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
58 S14 B2: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
59 S14 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....
60 S15 B0: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
61 S15 B1: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
62 S15 B2: Len 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
63 S15 B3: Len 16 00 00 00 00 00 00 00 7F 07 88 FF 00 00 00 00 00 .....

```

You get a dump of all the data on the card. If a character is printable, it shows up on the right, otherwise, there's just a period. Send me a note if you see the poem.

Next, we'll try formatting the card, then erasing the card, and finally writing five different records of varying lengths:

R for Read, F for Format, E for Erase, W for Write:

F

Please place card on reader

Card 0XDA8A0EC3 [0xC3, 0x0E, 0x8A, 0xDA]

FORMAT

Writing MAD sector failed.. Try erasing the card

Can't format

E

Please place card on reader

Card 0XDA8A0EC3 [0xC3, 0x0E, 0x8A, 0xDA]

ERASE

Erased

F

Please place card on reader

Card 0XDA8A0EC3 [0xC3, 0x0E, 0x8A, 0xDA]

FORMAT

.....Done

W

Please place card on reader

Card 0XDA8A0EC3 [0xC3, 0x0E, 0x8A, 0xDA]

WRITE

Please place card on reader

Card 0XDA8A0EC3 [0xC3, 0x0E, 0x8A, 0xDA]

WRITE

Please place card on reader

Card 0XDA8A0EC3 [0xC3, 0x0E, 0x8A, 0xDA]

WRITE

Please place card on reader

Card 0XDA8A0EC3 [0xC3, 0x0E, 0x8A, 0xDA]

WRITE

Please place card on reader

Card 0XDA8A0EC3 [0xC3, 0x0E, 0x8A, 0xDA]

WRITE

Note that the card has access permissions that prevent us from formatting at first. We have to erase all the data before we can write to certain places (and reformatting wants to write to a place the original format had protected).

We'll see a little bit about that when we look at our main.py file.

That big driver I talked about is in a file called mfrc522.py. We'll just look at how the class it defines is constructed before diving way in. The instance of that class is constructed in main.py.

The class itself is MFRC522:

Class Ndef:

READ = const(0)

ERASE = const(1)

WRITE = const(2)

FORMAT = const(3)

ISNTAG = const(4)

DUMPNTAG = const(5)

MAX_BLOCKS = const(64)

Def __init__(self):

From mfrc522 import MFRC522

Self.reader = MFRC522(sck = 14, mosi = 13, miso = 12, rst = 17, cs = 15, spi_id = 1)

Def checksum(self, data):

Crc = 0xc7

For byte in data:

Crc ^= byte

For _ in range(8):

Msb = crc & 0x80

Crc = (crc << 1) & 0xff

If msb:

Crc ^= 0x1d

Return crc

Def access_setup(self):

From RfidAccess import RfidAccess

Return RfidAccess()

Def Process(self, cmd, data = None, sect = None, blk = None):

```

From utime import sleep_ms
defaultKey = [0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF]
firstSectorKey = [0xA0, 0xA1, 0xA2, 0xA3, 0xA4, 0xA5]
nextSectorKey = [0xD3, 0xF7, 0xD3, 0xF7, 0xD3, 0xF7]
print("Please place card on reader")
for i in range(50):
    self.reader.init()
    access = self.access_setup()
    (stat, tag_type) = self.reader.request(self.reader.REQIDL)
    If stat == self.reader.OK:
        (stat, uid) = self.reader.SelectTagSN()
        If stat == self.reader.OK:
            Uid_str = hex(int.from_bytes(bytes(uid), "little", False)).upper()
            Uid_hex = self.reader.tohexstring(uid)
            Print("Card {} {}".format(uid_str, uid_hex))
        If cmd == READ:
            Print("READ")
            #read MAD sector (first sector)
            If self.reader.MFRC522_DumpClassic1K(uid, Start = 0, End = 4, keyA = firstSectorKey) == self.reader.OK:
                #read the rest of the card
                Self.reader.MFRC522_DumpClassic1K(uid, Start = 4, End = 64, keyA = nextSectorKey)
            Else:
                If self.reader.MFRC522_DumpClassic1K(uid, Start = 0, End = 4, keyA = defaultKey) == self.reader.OK:
                    #read the rest of the card
                    Self.reader.MFRC522_DumpClassic1K(uid, Start = 4, End = 64, keyA = defaultKey)
            Return True
        Elif cmd == WRITE:
            Print("WRITE")
            While len(data):
                Buf = bytearray(data[0:16], "utf-8")
                While len(buf) < 16:
                    Buf.append(0)
                If self.reader.writeSectorBlock(uid, sect, blk, buf, keyB = defaultKey) == self.reader.ERR:
                    Print("Write failed sector", sect, " ", block, blk)
                Data = data[16:]
                Blk += 1
                If blk == 3:
                    Sect += 1
                    Blk = 0
            Return True
        Elif cmd == ERASE:
            Print("ERASE")
            # set default access
            Access.decodeAccess(0xff, 0x07, 0x80)
            Block3 = access.fillBlock3(keyA = defaultKey, keyB = defaultKey)
            # print("Reset Mad Sector (first sector)")
            #reset first sector
            Self.reader.writeSectorBlock(uid, 0, 3, block3, keyB = defaultKey)
            #erase block1 and 2

```

```

Datablock = 16 * [0]
Self.reader.writeSectorBlock(uid, 0, 1, datablock, keyB = defaultKey)
Self.reader.writeSectorBlock(uid, 0, 2, datablock, keyB = defaultKey)
#reset all other sectors
For s in range(1, 16):
# permission to default
# print("Reset sector ", s)
Self.reader.writeSectorBlock(uid, s, 3, block3, keyB = defaultKey)
For b in range(3):
# put all data to zero block 0, 1 and 2
Self.reader.writeSectorBlock(uid, s, b, datablock, keyB = defaultKey)
Print("Erased")
Return True
Elif cmd == FORMAT:
Print("FORMAT")
#set MAD sector
# first fill block permission
Access.setTrailerAccess(keyA_Write = access.KEYB, access_Read = access.KEYAB, access_Write =
access.KEYB,
keyB_Read = access.NEVER, keyB_Write = access.KEYB)
access.setBlockAccess(access.ALLBLOCK, access_Read = access.KEYAB, access_Write = access.K
access_Inc = access.NEVER, access_Dec = access.NEVER)
block3 = access.fillBlock3(keyA = firstSectorKey, keyB = defaultKey)
# print("Len of block3 is", len(block3))
#Write the sector access
If self.reader.writeSectorBlock(uid, 0, 3, block3, keyA = defaultKey) == self.reader.ERR:
If self.reader.writeSectorBlock(uid, 0, 3, block3, keyA = firstSectorKey) == self.reader.ERR:
Print("Writing MAD sector failed.. Try erasing the card")
Return False
Else:
Print(".", end = "")
B1 = [0x14, 0x01, 0x03, 0xE1, 0x03, 0xE1, 0x03, 0xE1, 0x03, 0xE1, 0x03, 0xE1, 0x03, 0xE1, 0x03,
0xE1]
B1[0] = self.checksum(b1[1:]) # I know this is already ok but just to demonstrate the CRC
Self.reader.writeSectorBlock(uid, 0, 1, b1, keyB = defaultKey)
# b2 = [0x03, 0xE1, 0x03, 0xE1, 0x03, 0xE1, 0x03, 0xE1, 0x03, 0xE1, 0x03, 0xE1, 0x03, 0xE1, 0x03,
0xE1]
Self.reader.writeSectorBlock(uid, 0, 2, b1, keyB = defaultKey)
#set permission for all other sectors
Access.setTrailerAccess(keyA_Write = access.KEYB, access_Read = access.KEYAB, access_Write =
access.KEYB,
keyB_Read = access.NEVER, keyB_Write = access.KEYB)
access.setBlockAccess(access.ALLBLOCK, access_Read = access.KEYAB, access_Write = access.K
access_Inc = access.KEYAB, access_Dec = access.KEYAB)
block3 = access.fillBlock3(keyA = nextSectorKey, keyB = defaultKey)
#Write all next sectors access
For sector in range(1, 16):
If self.reader.writeSectorBlock(uid, sector, 3, block3, keyA = defaultKey) == self.reader.ERR:
Print("\nWriting to sector ", sector, " Failed!")
Return False

```

```
Else:  
Print(".", end = "")  
#force sector 1 to be 1 record empty  
Block = 16 * [0]  
Block[2] = 0xfe  
If self.reader.writeSectorBlock(uid, 1, 0, block, keyB = defaultKey) == self.reader.ERR:  
Print("Unable to set first NDEF record!")  
Return False
```

```
Elif cmd == ISNTAG:  
Print("Is NTAG" if self.reader.IsNTAG() else "Not NTAG")
```

```
Elif cmd == DUMPNTAG:
```

Python Radio 34: Immaculate Reception

Simon Quellen Field

Simon Quellen Field

Follow

19 min read

.

Feb 5, 2025

Listen

Share

More

FM Radio with Digital Data

Press enter or click to view image in full size

Immaculate Reception

MidJourney

We built an FM transmitter in the previous project. Now we will build an FM receiver.

Our receiver will not just play music. It will decode the RDS data to show us the station, the frequency, and which song is playing, along with other interesting data such as the signal strength, the type of music the station plays, and more.

We will also add controls so that we can seek to the next strong station, enter a frequency, control the volume, mute the audio, and even draw a graph of every channel by signal strength. That way, we can easily pick a frequency that has no competing signal for our transmitter.

The RDS data for many stations includes a digital time record, so we can set the microprocessor's real-time clock to the correct time (accurate to a tenth of a second). This is quite useful when our processor has no Internet connection and no GPS reception. We can make a self-setting clock and place it anywhere.

The chip we will be using is the Si4703 FM radio receiver. It is a very tiny chip, but thankfully there are breakout boards with it already soldered on:

Press enter or click to view image in full size

The hardware for the FM receiver

Photo by the author.

In the photo above, we connected the breakout board to a Wemos D1 Mini ESP8266 board. There are connections:

D1 Mini 3.3-volt power → 3.3V on the breakout board

D1 Mini Ground → Ground

D1 Mini pin D2 → SDIO

D1 Mini pin D1 → SCLK

D1 Mini pin D3 → RST

The breakout board has a 1/8-inch (3.5 mm) phone jack for earphones. You can use the earphones as the antenna, or (as I did) you can attach a wire to the earphone sleeve and another to the D1 Mini's

ground, and attach those to an external FM antenna. This helps for weak stations. The hardware goes together quickly. The software took a lot longer to create. But all you need to do is copy and paste.

Unlike the ESP32 or the RP2040, the ESP8266 only has 16 megabytes of memory. We pack a lot of features into our radio, and they don't all fit at once. But Micropython has a trick. If we move functions into their own files on the flash memory and import them in a function, any memory they take up gets freed when the function returns.

Each time I ran into memory constraints when writing the code, I moved some code into another file and imported it. So instead of one (big) main.py file, there are now seven. But we are able to cram all of the goodies onto the D1 Mini and not have to pay for a more expensive microcontroller. The main.py file is still the largest and is mostly the driver for the Si4703:

```

From time import sleep
Class xmit:
I2C_ADDRESS    = 0x10
# The Si4703 reads the registers starting at 0xA and wraps around to 0x0 at 0xF
# This enables code to save time by reading only a few status registers and ignore the control
registers.
DEVICEID       = 0x00 # Never changes
CHIPID         = 0x01 # Never changes
POWERCFG       = 0x02 # Control registers start here
CHANNEL        = 0x03
SYSCONFIG1     = 0x04
SYSCONFIG2     = 0x05
SYSCONFIG3     = 0x06
OSCILLATOR     = 0x07
# 8 and 9 are reserved
STATUSRSSI     = 0x0A # Status registers start here
READCHAN       = 0x0B
RDSA           = 0x0C
RDSB           = 0x0D
RDSC           = 0x0E
RDSD           = 0x0F
START_OSC      = 0x8100
VOLUME_LOW     = 0
EXTEND_RANGE   = 0x100
ENABLE_UNMUTE  = 0x4001
FREQ_DEFAULT   = 88_500_000
# Register 0x02 – POWERCFG
SMUTE          = (1 << 15)
DMUTE          = (1 << 14)
MONO           = (1 << 13)
# bit 12 is zero
RDSM           = (1 << 11)
SEEK_MODE      = (1 << 10)
SEEKUP         = (1 << 9)
SEEK           = (1 << 8)
# bit 7 is zero
DISABLE        = (1 << 6)
SEEKDN         = (1 << 1)
ENABLE         = (1 << 0)

```

```

# Register 0x03 – CHANNEL
TUNE      = (1 << 15)
# Low 9 bits are the channel
# Register 0x04 – SYSCONFIG1
RDSIEN    = (1 << 15)
STCIEN    = (1 << 14)
# Bit 13 is zero
ENABLE_RDS = (1 << 12)
DE        = (1 << 11)
AGCD      = (1 << 10)
# Bits 9 and 8 are zero
# The four below are all 2 bits wide
BLNDADJ   = (1 << 6)
GPIO3     = (1 << 4)
GPIO2     = (1 << 2)
GPIO1     = (1 << 0)
# Register 0x05 – SYSCONFIG2
SEEKTH    = 8          # The top 8 bits
BAND      = 6          # The next 2 bits
SPACE1    = 5
SPACE0    = 4
VOLUME_MASK = 0x000F   # The bottom 4 bits
# Register 0x06 – SYSCONFIG3
SMUTER    = 14         # 2 bits
SMUTEA    = 12         # 2 bits
# Bits 11 and 10 are unassigned
RDSPRF    = 9          # RDS performance bit
VOLEXT    = 8          # Extended volume range
SEEK_SNR   = 4          # 4 bits wide
SEEK_CNT   = 0          # 4 bits wide
# Register 0x07 – TEST1
AHIZEN    = (1 << 14)
XOSCEN    = (1 << 15)
# Register 0x0A – STATUSRSSI
RDSR      = (1 << 15)
STC        = (1 << 14)
TUNING_READY = (1 << 14)
SFBL      = (1 << 13)
AFCRL     = (1 << 12)
RDSS      = (1 << 11)
BLERA     = (1 << 9)    # 2 bits wide
STEREO    = (1 << 8)
RSSI      = (1 << 0)    # Bottom 8 bits
# Register 0x0B – READCHAN
BLERB     = (1 << 14)   # 2 bits wide
BLERC     = (1 << 12)   # 2 bits wide
BLERD     = (1 << 10)   # 2 bits wide
READCHANBITS = (1 << 9) # Low 9 bits
# RDS Variables
# Register RDSB

```

```

GROUPTYPE      = 11
GT_MASK        = 0x1F
TP             = 10
TA             = 4
MS             = 3
TYPE0_MASK     = 0x0003
TYPE2_MASK     = 0x000F
Def __init__(self, bus, pin_reset, i2c_address = I2C_ADDRESS):
From machine import RTC, Pin
Self.LED = Pin(2)
Self.LED.off()
Self._bus = bus
Self._i2c_address = i2c_address
Self._pin_reset = pin_reset
Self.regs = [0] * 16
Self.chan = 5
Self.threshold = 25
Self.rtc = RTC()
Self.rds_list = []
Self.flush_RDS()
Self._assert_reset()
Self.read_all_registers()
Self.regs[self.OSCILLATOR] = self.START_OSC
Self.write_all_registers()
Sleep(1)
Self.read_all_registers()
Self.regs[self.POWERCFG] = self.ENABLE_UNMUTE
Self.regs[self.SYSCONFIG1] |= self.ENABLE_RDS
Self.regs[self.SYSCONFIG2] |= self.VOLUME_LOW
Self.regs[self.SYSCONFIG2] |= (0x19 << self.SEEKTH)
Self.regs[self.SYSCONFIG2] &= 0xFFFF0          # Clear volume bits
Self.regs[self.SYSCONFIG2] |= 1                # Lowest volume
# self.regs[self.SYSCONFIG3] |= self.EXTEND_RANGE
# self.regs[self.SYSCONFIG3] |= (0x4 << self.SEEK_SNR)
# self.regs[self.SYSCONFIG3] |= (0x8 << self.SEEK_CNT)
Self.write_all_registers()
Sleep(0.2)
Def flush_RDS(self):
Self.pi = 0
Self.ps = bytearray(' ' * 8, 'utf-8')
Self.rt = bytearray(' ' * 64, 'utf-8')
Self.rt_index = 0
Self.ps_buf = [0] * 4
Self.ps_cnt = 0
Self.rt_buf = [0] * 32
Self.rt_cnt = 0
Self.old_rt_index = 0
Self.debug_str = ""
Self.program_service = ""
Self.RDS_text = ""

```



```

Self.other = ""
Self.time_str = ""
Self.has_time = False
Def _assert_reset(self):
Self._pin_reset.on()
Sleep(0.01)
Self._pin_reset.off()
Sleep(0.01)
Self._pin_reset.on()
Def has_RDS(self):
Self.read_all_registers()
If self.regs[self.STATUSRSSI] & self.RDSR: # We received some RDS data
Return True
Return False
Def get_group_type(self, gt):
From group_types import group_types
Type_str = group_types[gt]
Return type_str
Def get_program_type(self, pt):
From type_strings import rbds_types # Change this to rds_types if not in North America
Type_str = rbds_types[pt]
Return type_str
Def read_RDS(self):
Self.read_all_registers()
Self.new_data = False
Self.old_ps_AB = ""
Self.old_rt_AB = ""
If self.regs[self.STATUSRSSI] & self.RDSR: # We received some RDS data
From read_RDS import read
Read(self)
Return self.new_data
Def clear_RDS(self):
Self.ps = bytearray(' ' * 10, 'utf-8')
Self.rt = bytearray(' ' * 65, 'utf-8')
Def RDS_performance(self, yes_no):
If yes_no:
Self.bit_set(self.SYSCONFIG3, self.RDSPRF)
Else:
Self.bit_clr(self.SYSCONFIG3, self.RDSPRF)
Def frequency(self):
Self.chan = (self.read_register(self.READCHAN) & 0x3FF)
Freq = self.chan * 200_000 + 87_500_000
Return freq
Def get_volume(self):
Return self.read_register(self.SYSCONFIG2)
Def set_volume(self, v):
Self.read_all_registers()
Self.regs[self.SYSCONFIG2] &= 0xFFFF0
Self.regs[self.SYSCONFIG2] |= v & 0xF
Self.write_all_registers()

```

```

Sleep(0.2)
Def set_frequency(self, freq):
# Channels in the U.S. start at 87.5 MHz and have 0.2 MHz spacing
Self.flush_RDS()
Chan = int((freq - 87_500_000) / 200_000)
Self.read_all_registers()
Self.regs[self.CHANNEL] &= 0xFE00    # Clear channel bits
Self.regs[self.CHANNEL] |= chan | self.TUNE
Self.write_all_registers()
Sleep(0.2)
While self.read_register(self.STATUSRSSI) & self.TUNING_READY == 0:
Pass
Self.stereo = self.regs[self.STATUSRSSI] & self.STEREO != 0
Self.regs[self.CHANNEL] &= ~self.TUNE  # Clear the tune bit
Self.write_all_registers()
Sleep(0.2)
Def seek(self, direction, first_time = True):
Self.flush_RDS()
Self.read_all_registers()
Self.regs[self.POWERCFG] |= self.SEEK_MODE
If direction == True:
Self.regs[self.POWERCFG] &= ~self.SEEKDN
Self.regs[self.POWERCFG] |= self.SEEKUP
Else:
Self.regs[self.POWERCFG] &= ~self.SEEKUP
Self.regs[self.POWERCFG] |= self.SEEKDN
Self.regs[self.POWERCFG] |= self.SEEK
Self.write_all_registers()
While self.read_register(self.STATUSRSSI) & self.TUNING_READY == 0:
Pass
Self.stereo = self.regs[self.STATUSRSSI] & self.STEREO != 0
Sfbl = self.regs[self.STATUSRSSI] & self.SFBL
If sfbl:          # We have reached the band limit – double check RSSI
Rssi = self.regs[self.STATUSRSSI] & 0xFF
If rssi < self.threshold and first_time == True:
If direction == True:
Self.set_frequency(self.frequency() + 200_000)
Else:
Self.set_frequency(self.frequency() - 200_000)
Return self.seek(direction, False)
Self.regs[self.POWERCFG] &= ~self.SEEK
Self.regs[self.POWERCFG] &= ~self.SEEK_MODE
Self.write_all_registers()
Def show_seek_config(self):
Self.read_all_registers()
Th = (self.regs[self.SYSCONFIG2] >> 8) & 0xFF
Snr = (self.regs[self.SYSCONFIG3] >> 4) & 0xF
Cnt = self.regs[self.SYSCONFIG3] & 0xF
Snrdb = self.regs[self.SYSCONFIG3] & 0xF
Print(th, snr, cnt)

```

```

Def seek_config(self, threshold, snr, cnt):
    Self.threshold = threshold
    Self.read_all_registers()
    Self.regs[self.SYSCONFIG2] &= ~0xFF00
    Self.regs[self.SYSCONFIG2] |= (threshold & 0xFF) << 8
    Self.regs[self.SYSCONFIG3] &= ~0xFF
    Self.regs[self.SYSCONFIG3] |= (snr & 0xF) << 4
    Self.regs[self.SYSCONFIG3] |= cnt & 0xF
    Self.write_all_registers()
    Def rssi(self):
        Self.stereo = self.regs[self.STATUSRSSI] & self.STEREO != 0
        Return self.read_register(self.STATUSRSSI) & 0xFF
    Def read_register(self, reg_address):
        Self.read_all_registers()
        Return self.regs[reg_address]
    Def bit_clr(self, reg_address, bits):
        Self.read_all_registers()
        Self.regs[reg_address] &= ~bits
        Self.write_all_registers()
    Def bit_set(self, reg_address, bits):
        Self.read_all_registers()
        Self.regs[reg_address] |= bits
        Self.write_all_registers()
    Def write_register(self, reg_address, value):
        Self.regs[reg_address] = value
        Self.write_all_registers()
    Def write_all_registers(self):
        # Writing to the Si4703 starts at register 2 (since the other registers are read-only?)
        # So we only need a list that holds 2 through 7: 6 words or 12 bytes
        Buf = [0] * 12
        For i in range(0, 6):
            Buf[i*2], buf[(i*2)+1] = divmod(self.regs[i+2], 0x100)
            Self._bus.writeto(self._i2c_address, bytearray(buf, 'utf-8'))
    Def read_all_registers(self):
        R = self._bus.readfrom(self._i2c_address, 32)
        l = 0
        While i < len(r):
            lnd = int((i/2) + 10) % 16)
            Self.regs[lnd] = (r[i] << 8) | r[i+1]
            l += 2
    Def dump_registers(self):
        Print("DVID CHIP PWRC CHAN SYS1 SYS2 SYS3 OSC          RSSI RCHN RDSA RDSB RDSC RDS")
        For x in range(16):
            Print("{:04X}".format(self.regs[x]), end=" ")
        Print()
    Def histogram(self):
        From histogram import histogram
        Histogram(self)
        From machine import Pin, I2C
        D1 = Pin(5)

```

```

D2 = Pin(4)
D3 = Pin(0)
Def show_RDS(x, how_many):
x.RDS_performance(True)
old_s = ""
for cnt in range(how_many):
sleep(0.04)                # Records come in 11.4 times per second
if x.read_RDS():
r = x.rssi()
s = x.program_service
s += " {:.3f} - {:.04x} - {:.s} {:.s} {:.3d} ".format(x.frequency() / 1_000_000, x.pi, x.ms,
x.program_type, r)
s += x.RDS_text
s += " [" + x.other + "]"
# s += " [" + x.time_str + "]"
If x.has_time:
T = x.rtc.datetime()
Yr = t[0]
Mo = t[1]
Dy = t[2]
Wd = t[3]
Hr = t[4]
Mn = t[5]
Sc = t[6]
S += " [{:2d}/{:02d}/{:02d} {:02d}:{:02d}:{:02d}].format(mo, dy, yr, hr, mn, sc)
# s += x.debug_str          # Remove this if you don't want debug info
S += '                      \r' # The <CR> prevents scrolling. Lines write over old lines.
If s != old_s:
Print(s, end="")
# print()                  # Remove this if you don't want scrolling
Old_s = s
x.RDS_performance(False)
def demo(x, times, slp):
from tune_demo import tune_demo
tune_demo(x, times, slp)
def chip_dump(x):
from chip_info import show_data
show_data(x)
def main():
i2c = I2C(scl=D1, sda=D2, freq=100_000)
x = xmit(i2c, D3)
sleep(1)
chip_dump(x)
x.set_frequency(98_500_000)
x.show_seek_config()
x.dump_registers()
x.set_volume(7)
sleep(5)
# x.histogram()
Print("{:d} stations have RDS data".format(len(x.rds_list)))

```

```
For f in x.rds_list:
x.set_frequency(f)
show_RDS(x, 10_000)
x.set_frequency(98_500_000)
show_RDS(x, 10_000)
x.set_frequency(104_900_000)
show_RDS(x, 10_000)
x.set_frequency(105_700_000)
show_RDS(x, 10_000)
x.set_frequency(99_100_000)
show_RDS(x, 10_000)
x.set_frequency(100_300_000)
show_RDS(x, 10_000)
demo(x, 10, 10)
main()
```

The first part of the xmit class defines the registers and the bits we need to read or set.

The `__init__()` method starts the receiver chip receiving using some default settings.

Several methods deal with reading the RDS data. Then there are several methods for controlling the frequency and volume.

The chip deals with reading and writing the registers in a peculiar way that is designed to be more efficient. There is no way to select the register you want to read or write to. Instead, you read or write some number of bytes.

There are six control registers, starting at register 2. Since all the other registers are read-only, when you write to the chip, the writing starts at register 2. This is one peculiarity, but it makes a little sense.

The status registers come after that, starting at register 10 (0x0A). When you read from the chip, the reading starts at register 10. The assumption is that you already know what you put in the control registers, so you never need to read them. But just in case, if you read more than 12 bytes, the addresses wrap around, and you start reading register 0 and on up.

The first register read is register 10, which is also the register most often polled, so you can simply read 2 bytes (the registers are all 16 bits wide), and get register 10.

Register 0 and 1 never change, and are read-only. They describe the chip manufacturer and revisions of silicon and software, and which version of the chip you have.

All of this means that we keep a shadow copy of the registers in memory, and read into that or write from it, after making changes to the shadow copies.

That's it for the driver.

The `main()` routine is a demonstration platform for the chip and our code. This is the code you will most likely be changing to pare it down to the features you care about and add your own user interface.

We set up the I2C connection to the chip and create an instance of the xmit class. The `chip_dump()` routine decodes and prints out the data in registers 0 and 1, and it is just for show.

Next, we set the frequency and volume, and show some debugging info (the state of all the registers).

The histogram takes several seconds to create, as it has to tune into every FM channel from 87.5 to 108 and report the signal strength (called RSSI for Received Signal Strength Indicator). Because it takes so long, the demo has it commented out. But it's fun to see, and it finds the quietest channel for our transmitter.

If the histogram method was run it populated the `rds_list`, which has the frequencies of all of the stations that it found that were sending out RDS data. Since the RDS data is sent out at a lower power than the music, only strong stations will show up on the list.

Next is a set of stations in my area that I know have RDS data, and I tune to them and report the data they send.

Lastly, the demo() routine shows how to control the seek behavior of the chip.

The chip_info.py module looks like this:

```
Def show_data(x):
x.read_all_registers()
print("\033[2J")          # Clear the screen in case a reset printed garbage
dev_id = x.regs[x.DEVICEID]
if dev_id & 0xFFF == 0x242:
print("Manufacturer: Skyworks Solutions")
part_family = (dev_id >> 12) & 0xF
if part_family == 1:
print("Part family Si4702/3")
chip_id = int(x.regs[x.CHIPID])
rev = (chip_id >> 10) & 0x3F
print("Silicon revision:", rev)
dev = (chip_id >> 6) & 0xF
if dev == 0:
print("Device: Si4700")
if dev == 1:
print("Device: Si4702")
if dev == 8:
print("Device: Si4701")
if dev == 9:
print("Device: Si4703")
firmware = chip_id & 0x3F
print("Firmware revision:", firmware)
# dev_id was 0x1242
# chip_id was 0x1253
# RRRR RRcc ccFF FFFF
# 0001 0010 0101 0011
```

Since the demo expects to be run from a terminal emulator, and the D1 Mini p

The Weatherman: How NOAA Satellites Beam Forecasts to Earth

Investigator515

Investigator515

Follow

6 min read

.

Jun 4, 2025

Listen

Share

More

Use your laptop and SDR to collect live weather imagery from space

When you talk about radio, most people think of it as a way to communicate via voice. Amateur Radio operators use FM to talk on local repeaters, and your car radio uses the same modulation type to deliver entertainment while you're on the road. But, you won't have to look at the spectrum for long to realise that there is a whole bunch of activity that has nothing to do with voice communications. One of the most interesting parts of the spectrum to monitor is the Space Communications allocation. Here, we'll find telemetry and beacon data as well as a variety of other signals. With many of them being digital modes, we'll need to use plugins to help our SDR properly decode the transmissions.

One project that can be fun for beginners is downloading weather imagery. A good first step into the world of satellites, these signals are relatively strong and, due to their orbit pattern, provide good coverage globally. Let's take a look at what the options are for DIY space weather.

[Press enter or click to view image in full size](#)

The NOAA fleet helps predict global weather by sending imagery back to Earth. This is NOAA-19.

Source: Wikipedia

NOAA Satellites

The NOAA fleet has been a workhorse of data collection, collecting and sending data back to Earth with each orbit. Using the Automatic Picture Transmission mode (APT), satellites beam their data back to Earth in the Very High Frequency (VHF) band.

Some simple software allows the RTL-SDR to detect these transmissions, then apply some decoding magic to turn them into usable images.

The NOAA fleet has a strong and consistent signal, and its choice of a VHF downlink makes it a great candidate for beginners, as it's more tolerant of error correction and antenna choice.

This fleet is, however, composed of aging platforms. The venerable NOAA-15, for instance, was originally launched in 1988, and it's fair to say that, depending on your location, there are more modern options available to choose from.

[Press enter or click to view image in full size](#)

APT imagery of North America. Source: Wikipedia

Still, the remaining NOAA birds can be fun to explore if you haven't looked at satellite data before, and it's worth taking a look at the APT transmission mode before it's phased out entirely.

[Press enter or click to view image in full size](#)

This is the second generation, GOES-8. Source: Wikipedia.

GOES

Known as the Geostationary Operational Environmental Satellites (GOES) this program is also operated by NOAA, however, it comes with a few distinct twists.

Using a fleet of Geostationary satellites flying high above the Equator, GOES satellites provide near real-time, high-resolution imagery and data for weather forecasting, severe storm tracking, and space weather monitoring across the Western Hemisphere.

GOES provides stunning imagery due to its extreme orbit. This is one of the first images from GOES-1. Source: Wikipedia.

A multi-generation program, the GOES fleet started with the launch of GOES-1 way back in 1975, and over the years, the fleet has grown with the addition of new satellites.

The most modern satellites of the fourth generation were launched in 2016, meaning that they provide clear, modern imagery. Due to their extreme orbital distances though, you'll need to work a little harder to successfully capture and decode the downlink.

[Press enter or click to view image in full size](#)

The Meteor M-2 fleet originated in the old USSR as part of the Soviet weather program. This is the first M2. Source: Wikipedia.

Meteor M2

Originally developed by the Soviet Union in the early 1960s, the Meteor spacecraft are weather observation satellites launched by the Soviet Union and Russia since the Cold War.

Like the American program, the Meteor is a multi-generation fleet that's been refined with each future iteration, and while the program relies on a single class of satellite, each bird is packed full of useful technology.

While the modern fleet was originally planned to consist of four satellites, budgetary restraints meant that the Meteor-M project was significantly curtailed in comparison to its earlier goals.

The Meteor program would be plagued with issues in comparison to its equivalent NOAA projects. In 2017, Meteor-M 2.1 would be lost at launch, while in 2019, satellite 2.2 would cease transmitting imagery outright.

Despite these issues, 3 more Meteor-M satellites are currently being prepared for orbit.

Software & Trackers

If you're interested in trying to collect some imagery, you'll need to use some software to help you along the way, along with a tracker to help you calculate where the satellite is and when it will be there.

The web-based HeavensAbove is useful if you don't have a platform or don't want to install dedicated trackers on your device. If you're able though, the open-source Gpredict is a simple tracker available for all platforms that can be set up quickly and easily.

Linux users can find it in the APT repository, meaning you can install it using the following command

```
Apt install gpredict
```

While the original Wxtolmg was popular with space weather enthusiasts, it now suffers from a lack of developer support. So while you'll find plenty of tutorials that recommend it, we'd suggest straying from the crowd on this one.

The open-source Satdump program has a broad range of SDR support and is pre-configured to collect data from multiple platforms. While you'll have to install it from source, downloading the imagery is quick and easy once it's configured.

We'll be covering Satdump in a future tutorial, but for now, you can find out all you need to know by taking a look at the Satdump repository on GitHub.

Press enter or click to view image in full size

No Radio, No Problem

For some people, an interest in space weather may still not provide enough motivation to figure out antennas, receivers and decoders. There's no denying that collecting the weather using this approach has a decidedly heavy tech focus that some users might find difficult to navigate.

While we'd encourage you to mess about with your SDR a give it a shot, in some instances you may need data quickly and easily, and as you'd imagine, in the information age, finding this stuff can be pretty easy.

The first place to look is the US Government's NESDIS website. Hosting both NOAA & GOES imagery, can download up-to-date imagery that's relevant to your area.

If you'd like to try with radio but don't have your own station, you can try using the WebSDR project's remote receivers to capture your own data without needing a station.

Or you can bypass all the noise by downloading and analysing imagery from social media, thanks to the NOAASatellites X account. With more than 1.5 million followers, they regularly post imagery that's been collected from space.

There's never been a better time to try to be your own weatherman.

Medium has recently made some algorithm changes to improve the discoverability of articles like this one. These changes are designed to ensure that high-quality content reaches a wider audience, and your engagement plays a crucial role in making that happen.

If you found this article insightful, informative, or entertaining, we kindly encourage you to show your support. Clapping for this article not only lets the author know that their work is appreciated but also helps boost its visibility to others who might benefit from it.

■ We're now on Bluesky!

Articles we think you'll like:

What The Tech?! Space Shuttles

Shodan: A Map of the Internet

✉ ■ Want more content like this? Sign up for email updates

Space

Satellite

Radio

Software Defined Radio

Weather

Python Radio 2: Antennas

Follow

5 min read

.

Aug 20, 2024

Listen

Share

More

How to get more range

Press enter or click to view image in full size

Photo by the author

In the previous article in this series, we built a transmitter and receiver for 433.92 megahertz.

Python Radio: Simple Beginnings

The simplest digital radio mode

medium.com

It had a simple piece of wire for an antenna. We didn't worry about how long the wire was or what shape we bent it into.

However, our transmitter only emits 10 milliwatts of radio power, and with low-power transmitters like this, it makes sense to get the most out of it.

Ten milliwatts sounds tiny (and it is). But radio amateurs have awards for communicating 1,000 miles per watt and more:

From our point of view, people who get millions of miles per watt have an advantage. They are using the ionosphere to bend their signals around the earth. But 1,000 miles per watt is a mile per milliwatt, so with perfect conditions and the right antenna, we might see ten miles.

One of the ways to improve our antenna is to make it resonate at the frequency we use. We call this tuning the antenna. Each wave crest meets the echo of the previous one at exactly the right time and they reinforce one another.

To make our antenna resonate, we make the length of the antenna a quarter of a wavelength.

To calculate the wavelength, we ask Google to . It tells us the wavelength is 69.0893386 centimeters. Dividing that by four . We add another millimeter to account for the amount of wire we stick in the hole before soldering it in place.

Now we have antennas, but we can do better. Adding a ground plane is like putting a mirror under the antenna. It doubles the effectiveness. A ground plane can be as simple as a few wires soldered to the ground pin of the transmitter or receiver. We will use three wires. They should be just a little bit longer than a quarter wavelength. We use 12% longer, or about 19 centimeters.

We can optimize further now that we have an antenna and a ground plane. If the ground plane wires are perpendicular to the antenna, the whole antenna system presents 35 ohms of impedance to the transmitter. But our FS1000A is designed to feed a 50-ohm impedance (as are most of the transmitters we will find). If we bend the ground wires down to an angle of 42 degrees from perpendicular, the impedance changes to 50 ohms. If we bend them all the way down, we have now converted our system a half-wave dipole antenna, which has an impedance of 73 ohms.

If the transmitter's output impedance is different from the antenna's input impedance, some of the power gets reflected back into the transmitter, and is not available to the antenna to be radiated into space as radio waves. If the impedances are the same, we get the best power transfer.

Our FS1000A can only put out about 10 milliwatts of power, so we want the transfer to be as good as we can make it. With more powerful transmitters that put out multiple watts, having a big impedance mismatch causes so much power to be reflected back into the transmitter that the output transistor overheats and is destroyed. This can happen if we forget to attach the antenna, or if there is a short circuit to ground. But we don't have to worry when we are dealing with only 10 milliwatts.

Our transmitter is now raised up on the tripod formed by the ground plane, so we connect it to the ESP8266 using three wires:

Photo by the author

Likewise with the receiver:

Photo by the author

A closeup of the transmitter shows the connection:

Press enter or click to view image in full size

Photo by the author

And the receiver closeup:

Photo by the author

With the ground plane antennas on the receiver and the transmitter, I was easily able to send and receive Morse code at well over half a kilometer before I ran out of road. These are omnidirectional antennas, so the power is sent out in all directions. If we built directional antennas, such as a Yagi-Uda, we could get much greater distances.

A directional antenna focuses the transmitter power in one direction, much like a lens on a flashlight. Likewise, at the receiver, a directional antenna acts like a telescope, only hearing in one direction, and ignoring interfering noise from other directions. We have improved our signal-to-noise ratio.

There are A three-element antenna can be made by taping three wires to a piece of cardboard since the whole antenna is only about 9 inches long:

Press enter or click to view image in full size

With three elements we get over 5 dBd of gain. With six elements, we get over 9 dBd of gain. You can go nuts and use 20 elements (making the antenna 14 feet long) and get over 15 dBd of gain. That's like having a 15x telescope at each end of the link.

433.92 megahertz is in the UHF part of the radio spectrum. UHF signals are line-of-sight. They do not bend around the earth. But with the right antennas, you might find that if you can see it, you can signal it. Even with only 10 milliwatts of power. If this is surprising, consider that a 10-milliwatt laser pointer is so bright that you can see it for over 100 miles on a dark night.

Python

Micropython

Radio

From Concept to Antenna

How I design antennas for my devices.

Armando Rodrigues

Armando Rodrigues

Follow

6 min read

.

Dec 18, 2024

Listen

Share

More

Press enter or click to view image in full size

Myself climbing an antenna truss tower back in my college days.

Antenna design is a fascinating intersection of engineering and creativity. As an RF Engineer working on wireless automotive technologies, I have often found myself marveling at the ingenuity of a well-designed antenna. From enabling seamless connectivity in smartphones to ensuring reliable communication in spacecraft, antennas are the silent heroes of modern technology. In this article, I'll take you through the journey of designing antennas — from the initial concept to the final product — sharing insights, methods, and tools that have shaped my approach.

Press enter or click to view image in full size

My first ever self-made antenna: an 8-element Yagi-Uda antenna designed for analog video transmission. You can see I used very cheap materials for this project. But it did work perfectly anyway!

Why antenna design matters

The role of the antenna in any wireless communication system cannot be overstated. But, since RF is commonly seen as black magic for most electronics hobbyists, it is quite often overlooked. Antenna design directly impacts the size, aesthetics, cost, and battery life of devices. For example, a compact, highly efficient antenna can enable smaller devices with longer battery life, while a poorly designed one might increase device size and require excessive power for transmission, reducing battery life. This is why antenna design is critical at the concept stage. Instead of slamming a random antenna in your projects, you should always aim to select an antenna with the necessary characteristics in terms of gain, radiation pattern, efficiency, etc... Or design your own antenna if you are up to the challenge!

Selecting the right antenna type

Choosing the right antenna type is the first major decision in the design process. This choice depends on several factors, such as the operating frequency, the environment in which the device will function, and the desired radiation pattern.

For instance, monopole antennas, such as the ones shown below, might be ideal for compact devices that require omnidirectional coverage, while a patch antenna is better suited for applications needing a low-profile design and directional radiation pattern. Loop antennas are great for compact, low-frequency applications, whereas Yagi antennas, such as the one I showed earlier, excel in high-gain, point-to-point communications.

Press enter or click to view image in full size

Different types of antennas based on the classic quarter-wave monopole.

Understanding the trade-offs is key. Higher gain antennas offer better range but are almost inevitably more directional, making them less effective in environments requiring wide coverage.

Balancing these requirements ensures the selected antenna aligns with the overall device objectives.

If you don't know where to start you can consider the following list to spark some Google searches:

Whip Monopole Antenna

Advantages: Compact, omnidirectional, easy to deploy, and low-cost. Does not need a concrete ground plane to work (usually the device it is attached to functions as a ground plane).

Trade-offs: Limited bandwidth, ground plane quality can impact efficient operation.

Helical Antenna

Advantages: Supports circular and linear polarization (depending on the spacing between turns), offers moderate gain, and has a wide bandwidth.

Trade-offs: Usually presents lower radiation efficiency when compared to typical monopoles.

Patch Antenna

Advantages: Low profile, lightweight, and suitable for integration into PCBs or planar structures.

Trade-offs: Narrow bandwidth, lower efficiency, hard to tune.

Yagi-Uda Antenna

Advantages: High directional gain, excellent for long-distance communication and relatively simple design.

Trade-offs: Requires precise alignment and can be bulky for low-frequency use.

Printed Circular Monopole Antenna

Advantages: Wide bandwidth, compact size, and easy to fabricate on PCB.

Trade-offs: Lower efficiency compared to larger designs and can be sensitive to substrate material.

Planar Inverted-F Antenna (PIFA)

Advantages: Compact, low-profile, highly suitable for mobile and embedded devices.

Trade-offs: Narrow bandwidth, limited gain, and requires precise impedance matching.

Dipole Antenna

Advantages: Simple, robust design, omnidirectional in the horizontal plane, and easy to construct.

Trade-offs: Moderate size for low frequencies and less directional control.

Parabolic Dish Antenna

Advantages: Extremely high gain, highly directional, ideal for satellite and long-distance communication.

Trade-offs: Expensive, large, and requires precise mechanical alignment.

Horn Antenna

Advantages: High gain, excellent for high-frequency applications, and easy impedance matching.

Trade-offs: Bulky and expensive, primarily used for specialized purposes like radar and satellite communication.

Loop Antenna

Advantages: Compact, suitable for low-frequency applications, and robust against detuning.

Trade-offs: Low efficiency, low gain, and not ideal for high-frequency applications.

Understanding how the antenna works

An antenna isn't just a piece of metal — it's a finely tuned component that must resonate at specific frequencies to perform optimally. To design an effective antenna, you need to understand the principles of resonance, impedance matching, and polarization. You also need to understand how the physical characteristics of the antenna topology correlate with these important parameters. Usually, these relations between antenna topology and fundamental parameters (for example antenna length and operating frequency) are not linear, making the design and tuning process quite challenging.

Below you can see how the different physical aspects of a PIFA design affect its fundamental parameters such as bandwidth and input impedance.

Press enter or click to view image in full size

Main tuning parameter of a planar inverted-F antenna.

Refining the antenna: The cheap way

Prototyping antennas on a budget involves clever use of tools and resources. A good starting point is simulation software, which allows you to model and test antenna designs virtually. Open-source tools like NEC or free versions of commercial software can provide valuable insights.

Once a design shows promise in simulations, physical prototyping can begin. Using inexpensive materials like copper tape or PCB boards, you can create functional prototypes to test in real-world scenarios. Simple tools like vector network analyzers (VNAs) can measure parameters like return loss and impedance, ensuring the design performs as expected. Nowadays it is not hard to find cheap VNAs in AliExpress or similar websites that, although clearly not machines of precision, can still handle any hobbyist project.

Iterative refinement is key at this stage. Small adjustments to dimensions, spacing, or materials can have significant impacts on performance. This approach keeps costs low while allowing for substantial design improvements.

Refining the antenna: The expensive way

For more advanced designs or when budgets allow, high-end tools and facilities can take antenna refinement to the next level. Professional-grade simulation software, such as CST Microwave Studio or ANSYS HFSS, provides highly accurate modelling, including 3D electromagnetic interactions. Access to anechoic chambers allows precise testing of radiation patterns, gain, and efficiency in a controlled environment. High-quality fabrication methods, like photolithography for PCB antennas or CNC machining for metal elements, ensure prototypes closely match design specifications.

While these methods are costly, they're invaluable for achieving top-tier performance, particularly in demanding applications like automotive or aerospace systems where I work. These investments are obviously meant for companies, not hobbyists.

Conclusion

Antenna design is a blend of science, creativity, and practicality. From understanding the basics of resonance and impedance to selecting the right type and refining the design, each step requires careful consideration and iteration. Whether working on a tight budget or with state-of-the-art tools, the goal remains the same: to create an antenna that performs optimally and has the right characteristics. More gain or bandwidth is not necessarily better, the characteristics your project needs are the ones you should always aim for.

Antenna

Rf

Engineering

Software Defined Radio & Radio Hacking: Part 2

Investigator515

Investigator515

Follow

8 min read

.

Sep 25, 2023

Listen

Share

More

Discussing the Concept and Importance, of Radio Direction Finding

In Part 1 of this article, we looked at the relevance of SDR in cyber security and other roles. Today's article is Part Two of the series and is a focus on direction finding and Geolocation of signals. It's a lesser skill in regards to cyber security but it is an important skill with regards to isolating signal interference and understanding how to track and locate signals as required. It's also still commonly used today, being important in space exploration, search and rescue operations as well as having a valid military applications.

Being based around physics, some of the fundamentals are older yet proven concepts but when that's paired with modern technology we can still have some great fun with it. It'll also be a longer article today as there's several important concepts to impart. Let's get started.

The History:

Discovered in the early 1900's it took wartime innovation to bring direction finding techniques and strategy into mainstream research. Early radio pioneer Guglio Marconi was involved in discovering key fundamental techniques, however like most early technology these were rudimentary and took time to evolve.

An important part of researching the new technology occurred when the British established a chain of direction finding stations on the coast, targeting German U-Boats allowing proper techniques to be used and refined. This technology was further refined during WW2 with the addition of radar. This eventuated in a system known as HF/DF or "Huff Duff". Huff Duff focused on German High Frequency radio transmissions and its installation on to most allied ships caused significant disruption to the U-Boats, with fresh positions being able to located every time a boat would transmit.

FH4 system, affectionately known as Huff Duff. Source: Wikipedia

Direction Finding was also a key part of safe aviation navigation in the days before GPS, with Non Directional Beacons and VHF Omni Range systems allowing aircraft to navigate the skies with surprisingly good accuracy.

Press enter or click to view image in full size

VHF VOR aviation navigation station. Source: Wikipedia.

The Strategies:

While direction finding sounds pretty complicated it's actually far easier than you'd think. It's also a great exercise for those getting started with Radio in general, as to DF something successfully you'll need to understand concepts relating to signal propagation, signal strength and

signal type, as well as the concept of harmonic frequencies and how they work. While professional systems may use some of the methods below, beginners can get started with nothing more than a receiver with a good antenna. It's important to understand the limitations that will come with that however, as usually the less technique you're using the longer it will take to DF your chosen signal. However a beginner can get started with the basics to get a feel for it as needed. Firstly though, we need to understand how your receiver antenna radiates and by default, will receive. If you have an omni directional antenna, the pattern will look like this doughnut on the left side of the diagram.

[Press enter or click to view image in full size](#)

Different Radiation patterns will work to your advantage. Source: Wikipedia

Where as if you have a directional antenna such as a Yagi, your radiation pattern will look like this narrow one on the right instead.

Understanding these patterns are important. This is because as we develop our techniques we can use different types of antennas to DF quickly and more efficiently. While you won't necessarily need them for today, it is important to understand these concepts early.

Triangulation merely relies on plotting bearings on your map. It's the best way to start. Source: Wiki

Triangulation:

Triangulation is one of the fundamental ways to DF a signal, requiring little more than a bearing for your signal and a map to plot it on. Triangulation works best with multiple receivers, however should you only have one you can use a single receiver and change positions to get a fresh bearing instead.

To triangulate you'll tune your receiver to your frequency of interest, and then turn slowly in a 360 degree circle. To find your bearing correctly, use the onboard compass on your smartphone to obtain an appropriate fix. Where the strongest signal is is the most likely bearing for the signal. Plot that on your map. You'll need an additional two bearings to properly triangulate but once you have them, the point on your map where the three lines overlap is where your transmitter is.

[Press enter or click to view image in full size](#)

Kerberos SDR uses 4x RTL as receivers along with smart software for TDOA applications. Source: rtl-sdr.com

Time Delay On Arrival (TDOA)

TDOA is a more advanced version of triangulation that uses time delay of reception between receivers, along with some smart software processing to calculate a fix extremely quickly. Quite common in military applications TDOA is also common in the civilian research, with a number of off the shelf systems available for purchase.

TDOA is also pretty impressive as a system, as in typical systems the delay can be mere milliseconds, with the processing power doing the heavy lifting with regards to isolating the signal and geo location.

Depending on your level of capability with hardware hacks, a TDOA system is a great RTL project, with many systems having been designed and hardware cheap and readily available.

Should you decide to build a TDOA system, you'll gain detailed experience on configuration of RTL-SDR dongles, programming of the GUI to receive and process the signals as well as antenna experience, bringing the whole system together. While it may be out of reach for beginners, experienced hackers and makers can have a great time with a project like this.

Check out this blog post on building a TDOA system, or find a commercial option off the shelf via this link instead. There's also a super interesting thesis on the RTL-SDR as well.

[Press enter or click to view image in full size](#)

There's math for TDOA & Phase. Lots and lots of math. Thankfully, we delegate. Source: Wikipedia Phase Comparison

The last of our common techniques, Phase comparison is similar to TDOA however rather than taking

our measurements at the receiver, we'll apply them at the antenna. Like TDOA, most of the magic is within the software however that processing power enables the rapid geo location of signals with ease.

The basic concept of the method involves measuring the offset between antennas and using this math to apply a bearing to the received signals. Phase comparison is often used in both aviation and military applications and while there's better ways for researchers to DF signals locally, it's still a fundamental concept that should be understood.

Challenges and Limitations:

While this sounds like a lot of information to start with, it's important to realize that TDOA and Phase calculation aren't really the best way to get started for beginners. The best way to get into this type of thing is to use your RTL to explore the spectrum and develop an understanding around triangulation and plotting RF Data.

One thing you will have to deal with as part of this however is isolating interfering signals and more importantly dealing with an overloaded receiver. This is where understanding the spectrum can come into play as we are able to add, and modify devices to deal with this. The obvious rule of thumb where signal strength increases close to antenna proximity means that eventually you'll reach a point where you need to attenuate the signal to continue to close in on the location.

The simple addition of a directional antenna will give a significant improvement to any triangulation DF system as it will enable more precise bearings as well as having a significant "null spot" to the sides allowing for better isolation. Designs exist for cheap, easy home made tape measure antenna's that can allow you to experiment with this some more.

Frequency Harmonics:

Harmonic frequencies refer to unwanted additional frequencies that are generated at integer multiples of the fundamental frequency being transmitted. These harmonics can result from imperfections or nonlinearities in the transmitter's electronic components, such as amplifiers and oscillators.

For instance, if a radio transmitter is intended to broadcast a signal at a specific frequency, say 100 MHz, due to imperfections in its components, it may also unintentionally emit energy at multiples of this frequency, such as 200 MHz (2nd harmonic), 300 MHz (3rd harmonic), and so on. These harmonic frequencies can interfere with other radio communication systems and can potentially violate regulatory limits set by authorities to prevent electromagnetic interference.

For direction finding purposes, we can use these harmonics as a cheap form of attenuation when attempting to DF an elusive signal. Monitoring the weaker second or third harmonic, will allow you to track a signal far closer than tracking it on the fundamental frequency.

Something to Remember:

Before we close today's article it's important to understand that certain parts of radio theory can take significant time to learn and understand. We've tried to break that down in to beginner sized portions, allowing people that may be unfamiliar with the spectrum to get involved and start experimenting. Unfortunately because of this, we've had to select the bits of the theory side to leave out, so the article doesn't turn into a novel.

So you may find alternate ways of locating a signal, or different antenna types relating to Direction Finding and that's entirely okay because like anything, there are many effective ways of doing this. We've simply tried to break it down enough for you to get started. Once you're away, we'd recommend continuing your research so you develop your spectrum knowledge, as large parts of the spectrum remain open to cyber security research. These are often less popular than more common roles meaning there's plenty to discover in the way of exploits and hardware.

Looking to exploit Bluetooth? SDR. Looking to reprogram an electronic billboard, or intercept data from devices in the ISM bands. SDR. Looking to conduct research into automotive systems or IOT devices. SDR. And best of all, do you want to move from passive signal reception to offensive red teaming?

You got it. You'll need an SDR.

Remember, there's a vast array of devices, formats and transmission types in the SDR world and it's up to you, the radio hacker to ensure you continue take the next steps in discovering them.

Part 3 will focus on intercepting Space Communications & Satellites, but if you'd like to read about another specific SDR topic, feel free to nominate one in the comments. There's also a separate article on some simple antenna designs to affordably improve your SDR reception coming in the future as well.

Medium has recently made some algorithm changes to improve the discoverability of articles like this one. These changes are designed to ensure that high-quality content reaches a wider audience, and your engagement plays a crucial role in making that happen.

If you found this article insightful, informative, or entertaining, we kindly encourage you to show your support. Clapping for this article not only lets the author know that their work is appreciated but also helps boost its visibility to others who might benefit from it.

Enjoyed this article? Support our work and join the community!

Support me on Ko-fi: Investigator515

Join our Telegram channel for exclusive updates or.

Follow us on Twitter

Articles we think you'll like:

Software Defined Radio & Radio Hacking

What the Tech?! Personal Computers

✉ Want more content like this? Sign up for email updates here

Hacking

Software

Radio

Python Radio 8: Hacking a Receiver

Simon Quellen Field

Simon Quellen Field

Follow

6 min read

.

Aug 24, 2024

Listen

Share

More

The ESP32 as a local oscillator

Press enter or click to view image in full size

Photo by author

The little XY-MK-5V receiver we have been using (see photo above) is a type known as a regenerative receiver. This is a very old design, invented in 1912 by an undergraduate at Columbia University named Edwin Armstrong.

Some things to notice: the tuning is done with a tiny screwdriver that moves a ferrite slug in the coil in the center. There is one integrated circuit, the LM358 audio amplifier chip, in the upper right. There are two transistors (the 2SC3356s, marked R25), and a number of surface mount resistors and capacitors, in addition to two more coils, the three-turn coil and the big epoxy-coated coil near the center.

There is a lot going on there, out in the open where we can see it.

But there is a better way to build a radio receiver. It was also invented by Edwin Armstrong (in 1917), and separately by the Frenchman Lucien Levy. This radio mixes the incoming radio waves with a locally generated signal to produce "beat frequencies", which are the sum and the difference frequencies between the original signal and the local oscillator signal.

This has many advantages. The amplifier stages now only have to deal with one frequency (called the intermediate frequency). Tuning is done by changing the local oscillator.

One such superheterodyne radio receiver designed to replace the XY-MK-5V is the RXB12.

[Press enter or click to view image in full size](#)

Photo by author

Right away we can see it looks more modern. Almost all of the electronics are in the SYN470R integrated circuit in the center of the board. There are eight surface-mount capacitors around the chip, and one zero-ohm resistor (used as a jumper — more about that later). But there is also a big metal can in the upper right corner (big is relative, the whole board is only 30 millimeters across). That can holds a quartz crystal which sets the frequency to 433.98 megahertz.

The new receiver has more precise tuning (much like a quartz watch keeps better time than one with moving parts), and it is a little more sensitive, giving better range. I got mine for 58 cents from AliExpress, and you can get one with the transmitter for another 12 cents, about the same as what I paid for the XY-MK-5V and the FS1000a.

Get a few of them, since we are going to butcher at least one as we modify it to allow us to tune it to a huge number of frequencies.

The chip uses the crystal to control its local oscillator. But the chip also allows us to connect our own oscillator instead. Our oscillator is inside the ESP32, in the form of its Pulse Width Modulation feature. The ESP32 can change the state of a pin as fast as 40 million times per second using PWM.

We will modify the board by removing the crystal. Hold the board with the crystal can facing down, and place the soldering iron so that both leads of the crystal simultaneously heat up enough to melt the solder. The crystal can will just fall out.

Now we can attach a wire where the crystal used to be. We want the hole closest to the center of the board. The other hole is already connected to ground, so we don't need a wire for it.

While we have the soldering iron hot, we might as well attach the antenna. The holes on the side of the board away from the pins are the antenna (nearest the edge) and the ground (closer to the chip). I chose to solder in an SMA connector there, so I can easily switch antennas.

Photo by author

When we are all finished, and the antenna is connected and the receiver wired up to the ESP32, it looks like this (after I flattened things to improve the focus):

[Press enter or click to view image in full size](#)

Photo by author

For this project, I chose an ESP32 WROOM-32 (about \$2 at AliExpress.com). The receiver gets its power and ground from the +5 volt and GND pins on the board. The data from the receiver goes to pin 36. The board has another ground pin near the middle, and the black wire from the speaker goes to that. The red wire of the speaker goes to pin 19. Lastly, the new pin on the receiver board that we added for tuning goes to pin 18.

The code looks like this:

```
From machine import Timer, Pin, PWM
```

```
Def main():
```

```
From time import sleep_us
```

```
Freqs = [433.98, 315]
```

```
Speaker_pin = Pin(19, Pin.OUT)
```

```
Speaker_pin.value(0)
```

```
In_pin = Pin(36, Pin.IN)
```

```
While True:
```

```
Speaker_pin.value(0)
```

```
For x in freqs:
```

```
F = int(((x + (0.86 * x/315))/64.5) * 1000000)
```

```

Print(x, f)
Tune = PWM(Pin(18), freq=int(f), duty=512)
For count in range(1000000):
Speaker_pin.value(in_pin.value())
Sleep_us(10)
Main()

```

We will be playing with two frequencies, 119 megahertz apart. These are the two frequencies you can buy the little FS1000a transmitters for, so that is convenient. The frequencies are 433.98 megahertz and 315 megahertz.

You can see we have attached the speaker to pin 19, and we will be copying the state of pin 36 to pin 19. We must do this because our little receiver board does not have enough power to drive a speaker (but the ESP32 pin does).

You may have noticed that the crystal can was marked 6.7458 and not 433.98. That is because the superheterodyne chip has a frequency multiplier built in. This allows for the use of crystals such as our 6,745,800 Hertz crystal, but it also allows little computers like the ESP32 to set the frequency, even though they can only reach 40 megahertz, and not the 433.98 required.

The complicated-looking arithmetic is just the way we convert the frequency we want (433.98 MHz) into the frequency the receiver wants. The receiver will then multiply the frequency we give it to the final 433.98 MHz.

We print out the frequency we asked for, and the frequency we send to the receiver.

Next, we set up the Pulse Width Modulator to send out square waves (duty cycle 512) at the right frequency on pin 18.

The loop after that spins a million times, copying the receiver output into the speaker and then waiting for 10 microseconds. The whole loop thus takes 10 seconds, during which time the receiver is listening to the first frequency, and we can hear some noise from the speaker.

In this way, our device will let us hear communications on first 433.98 MHz, and then on 315 MHz, and then back, over and over.

To make the demonstration interesting, we will build two of our little ESP8266 Morse by Keyboard projects. One will run the following code:

```

From morse import Morse
Def main():
PIN_D4 = 2
# tone = int(input("Tone? "))
Tone = 800
Morse = Morse(PIN_D4, tone)
Print("Morse code AM Beacon")
Morse.speed(20)
# str = input("Enter the message to send: ")
Str = "Patrizia knows two or three letters of morse code, but she won't admit to it."
While(True):
Morse.send(str)
Main()

```

The other ESP8266 beacon will be running this code:

```

From morse import Morse
Def main():
PIN_D4 = 2
# tone = int(input("Tone? "))
Tone = 300
Morse = Morse(PIN_D4, tone)
Print("Morse code AM Beacon")

```

```
Morse.speed(20)
```

```
# str = input("Enter the message to send: ")
```

```
Str = "Dave used to know some Morse code, but has forgotten all but a few letters."
```

```
While(True):
```

```
Morse.send(str)
```

```
Main()
```

Since each beacon uses a different tone, it is easy to tell which one we are receiving. As our receiver switches between the two frequencies, we can hear the Morse code change tone and message.

It sounds like there is a conversation going on.

Python Radio 11: Power

Follow

6 min read

.

Aug 28, 2024

Listen

Share

More

Let's get some watts out...

Press enter or click to view image in full size

MidJourney

The Raspberry Pi Pico (RP2040) delivers about 3.8 milliwatts of power to a pin (when transmitting at about 7 megahertz). You can (as we will see later) do some amazing things with only this small amount of power. The ARRL has an award you can win called the 1,000 miles per watt award, as does the NAQCC. This is often won by reducing power to levels the RP2040 can reach. Connect your RP2040 to a cheap end-fed half-wave antenna and contact someone 4 miles away and you win.

But many people would like more power. This is where an amplifier comes in handy.

CW is one of the easiest modes to amplify since it is not picky about fidelity. We can use a Class C amplifier and get efficiencies between 75% and over 90%, making it easier on batteries if we go portable.

To make our amplifier, we will be using an RF power transistor. Specifically, a cheap 2SC2078 transistor.

We will use our transistor as a switch. Our transistor has three pins, called the base, the collector, and the emitter. The collector is connected to the positive side of the battery. The emitter is connected to the negative side of the battery. If we put a small positive voltage on the base (such as the 3.3 volts from an RP2040 pin), the transistor will turn on, and get quite hot as it shorts out the battery. So we won't do that.

Instead, we will use an inductor and a capacitor (or two) to make a resonant circuit tuned to the frequency range we are interested in (in this case, the range of 7.0 MHz to 7.3 MHz, known as the 40-meter band for the length of its waves).

If we send a brief pulse of current into our resonant circuit, it will ring like a bell, making waves at 7 MHz that quickly die out. This is like giving a child on a swing just one push. This will not be the happiest kid in the playground.

However, if we give the swing a push every time it comes back to us, we get a steady rhythm whose frequency is determined mostly by the length of the ropes, but also to some extent by exactly how often we push.

We can set up our circuit like this:

Press enter or click to view image in full size

Image by author

This schematic was taken from the free software package LTSpice from Linear Technologies. It allows us to simulate our circuit on the computer and test our assumptions and fine-tune our understanding.

The coil and two capacitors up at the top should be familiar. That is our low-pass filter. When the transistor is turned on (by getting a 3.3-volt pulse from the RP2040) the capacitors charge and the coil builds up a magnetic field. When the transistor switches off, the field collapses, inducing a current in the coil that is aided by the capacitors that are now discharging.

That current now goes out to the antenna, since the transistor looks like an open switch when its base is at zero volts.

What is interesting is that because the current from the coil and the capacitors is added to the current from the battery, the voltage at the antenna is almost twice the battery voltage. In the simulation, we get 5.6 watts out to the antenna, instead of the 3.8 milliwatts we previously got from the RP2040.

In our previous transmitter, we sent a square wave to the antenna. If we sent a square wave to our amplifier, the transistor would be turned on half the time. The efficiency would only be about 25%, and we would be dissipating almost 20 watts in the transistor, and getting about 11 watts out. Our battery would last seconds, and it and the transistor would get dangerously hot.

Instead, we only turn the transistor on for a quarter of the time. In this case, the period of the pulses is 142.207053 nanoseconds (7032000 hertz), and the pulse width is 35 nanoseconds.

Things are looking good so far. But our waveform doesn't look like the nice clean sine wave we need before we connect our transmitter to an antenna:

[Press enter or click to view image in full size](#)

Image by author

The tops of the peaks are sharp points, and the bottoms are flattened at zero.

The distortion is even easier to see when we look at it in the frequency domain, after doing a fast Fourier transform:

[Press enter or click to view image in full size](#)

Image by author

We want to see a single peak at the fundamental frequency (7032000 hertz), without all those peaks at the harmonic frequencies. People listening to all of those other frequencies don't want to hear us.

Let's see what we can do to fix that.

For one thing, we have an impedance mismatch between the RP2040 and the base of the transistor. The RP2040 has an output impedance of around 50 ohms (I measured 46 ohms, which is close enough). The transistor base would like to see about 11 ohms.

We can use a pi network filter to fix that. There are pi network impedance matching calculators on the web, and they tell us that a 0.66 Henry coil, an 820 picofarad capacitor, and a 1,000 picofarad capacitor will give us the proper match:

[Press enter or click to view image in full size](#)

Image by author

Our output power just jumped from 5.62 watts and 16.86% harmonic distortion to 6.81 watts and 13.89% distortion. Better, but still not something we want on the air. Since our efficiency is still up at almost 90%, it is time to sacrifice some of that output power for spectral purity. We'll aim for about 3 watts, and settle for 70% efficiency. We'll start by looking at the output impedance of the transistor, and match it to the antenna.

The output impedance is the voltage squared divided by twice the output power in watts. The voltage is 9 volts, which is 81 when squared. The output power we said would be 3 watts, so 81 divided by 6 is 13.5 ohms of output impedance. If we use a transformer to match the impedance to the antenna, we would want a 1:3.7 ratio for the inductances in each coil. By running different numbers through the simulation, we find that a 1:2.3 ratio gives us the highest power (3.25 watts) but something like 1:0.6 gives us better spectral purity. So we get to choose, trading off power for beauty.

The FCC is happy if our second harmonic is 43 dB down from the first. If I pick a 1:2 ratio for the transformer and follow it with a 5-pole filter, I get my 3 watts, and my harmonics are over 47 dB

down, giving us room for component values that might be off by 5% or more when we actually build the device.

The simulation shows a beautiful sine wave:

[Press enter or click to view image in full size](#)

[Image by author](#)

and a very clean spectral plot:

[Press enter or click to view image in full size](#)

[Image by author](#)

Our final schematic looks like this:

[Press enter or click to view image in full size](#)

[Image by author](#)

Radio Transmitter

Rp2040

Rf Power Amplifiers

Rf Filters

Python Radio 28: It's Easier than You Think

Understanding how the hardware works.

Simon Quellen Field

Simon Quellen Field

[Follow](#)

13 min read

.

Sep 25, 2024

[Listen](#)

[Share](#)

[More](#)

[Press enter or click to view image in full size](#)

Abstract art

MidJourney

Programmers are smart people. But sometimes they lack confidence when it comes to hardware issues.

Here we will explain how the hardware works and why it works, in a language tailored to programmers.

But first, we will build the hardware.

In the previous article, we built a repeater to get more range from our HC-12 transceivers. But we still used the inefficient little spring antennas shipped with the radios.

With those antennas, we can get about 300 meters of range instead of the 1000 meters claimed for the radios. The quarter wave ground plane antenna we built earlier would get the full kilometer (and a bit more).

We chose the ground plane antenna over a dipole because it is easy to adjust the angle of the ground plane wires to 42 degrees to get the 50-ohm impedance match we wanted. But today we are going to build a dipole and match the impedance using a gadget called a gamma match.

For reasons I will explain shortly, we will build our antenna on a sheet of foam-core board 48 inches by 36 inches.

In garden stores, they sell gummed copper tape for keeping snails and slugs away from flower beds.

This is very convenient stuff to use when building an antenna on foam core. You can solder to it easily. You can tune the antenna easily by folding back a bit of the tape at the ends. This is better than cutting it because you can undo the folding if you have gone too far.

Below is the dipole, after tuning it with the NanoVNA (vector network analyzer, a very convenient tool when building antennas, and available at Amazon.com for fifty dollars).

[Press enter or click to view image in full size](#)

Half-wave dipole.

Half-wave dipole (photo by author)

NanoVNA

NanoVNA (photo by author)

The dip in the yellow curve shows the antenna is tuned to 433.4 MHz. The shallowness of the curve shows that the antenna is wide-band, and does not have a steep resonance at 433.4. This will allow it to work well at the entire frequency range of the HC-12.

The green curve shows that we have matched the impedance to 50 ohms.

We did the impedance match using the long center conductor of the coax. By connecting it far away from the center of the dipole, we can lower the impedance from the dipole's 72 ohms down to 30 ohms or less (should we want to go that low).

In this case, the impedance match was only a few millimeters from the center of the dipole. I soldered it in place, and the impedance jumped to 55 ohms, which is still just fine.

If you want to build this antenna and don't have a NanoVNA (yet) the dipole ended up 26 centimeters long, and the gamma match was 3 millimeters long.

[Press enter or click to view image in full size](#)

Dipole connected to HC-12 and RP2040.

Dipole connected to HC-12 and RP2040

To connect the dipole to the HC-12 we use an SMA female to IPEX cable pigtail. These are available on AliExpress.com for 28 cents. You could just solder the antenna hole on the HC-12 to one leg of the dipole and ground to the other, but I wanted to use the NanoVNA to tune the antenna and it has an SMA input. The HC-12 has a nice tiny IPEX connector on the board.

We will use the following main.py module to send text to the repeater we built in a previous project.

```
From machine import Pin
```

```
From hc12 import HC12
```

```
From time import sleep
```

```
LED = Pin(25, Pin.OUT)
```

```
Def main():
```

```
Radio = HC12(1, 4, 5, 3, 1200)
```

```
Radio.long_distance()
```

```
Radio.command("C001")
```

```
Radio.status()
```

```
Count = 0
```

```
While True:
```

```
Send = "Sending: " + str(count)
```

```
Print(send)
```

```
LED(1)
```

```
Radio.write(send)
```

```
LED(0)
```

```
Sleep(5)
```

```
Count += 1
```

```
Main()
```

I promised earlier that I would explain why we needed so much foam-core. We are going to use it now. Imagine you had a light bulb and you wanted to double the amount of light sent in one direction. You would put a mirror behind the light bulb.

In radio, it goes by the synonym: a reflector.

We could just put it anywhere behind the dipole, but if we put it a quarter wave away, the reflection adds to the dipole's energy in phase, so we get the full effect.

[Press enter or click to view image in full size](#)

2-element Yagi-Uda antenna.

2-element Yagi-Uda antenna (photo by author)

The reflector should be a bit longer than the dipole. About 5% longer works well (I went crazy in the photo above). From the center of the dipole tape to the center of the reflector tape is a quarter wave. Don't worry a lot about millimeters here, the spacing is fairly forgiving. The full sine wave is 70 centimeters, and even if the reflected wave doesn't match up perfectly with the transmitted wave, it will still add a lot. Besides, this antenna works for almost the entire 70-centimeter band, from 424 MHz to 442 MHz, so it will only be absolutely perfect at one tiny spot, but work just fine everywhere else.

The gain of our antenna just went from about 2 to over 4. And when we receive signals, we get far less noise from behind us, so our signal-to-noise ratio almost doubled too.

This design is a 2-element Yagi-Uda antenna, named after the Japanese engineers who invented it. Four times better than an isotropic antenna (one that transmits equally in all directions, like our light bulb) is pretty good. But we can do better.

If we put some slightly shorter conductors in front of the dipole, they act as "parasitic elements". They absorb some of the radio energy and re-radiate it. If they are carefully spaced, they can constructively and destructively interfere with the radio waves, and act like a lens in front of the dipole.

There are many Yagi calculators on the web that will show you the proper spacing. I used one to calculate a 7-element Yagi-Uda antenna, and came up with this:

Press enter or click to view image in full size

7-element Yagi-Uda.

7-element Yagi-Uda (photo by author)

The gain is almost 10 times better than the dipole (9.88 dBd, or about 12 dBi). It performs so well, I have named it Carlos Antenna (Oye como va?).

How the hardware works

I have used several hardware words and concepts without explanation, and here is where I will catch you all up.

Impedance

Most "simple machines" you learn about in elementary physics, such as the lever, the pulley, the inclined plane, and the screw, are impedance-matching devices.

Press enter or click to view image in full size

Simple Machines

Flicker Creative Commons

An impedance-matching device trades off one thing for another. A lever lets you lift a heavy weight with little force, but the tradeoff is the distance your smaller force moves. Likewise with an inclined plane or ramp. You can carry a heavy weight up to a higher place, but you walk farther to do it.

A megaphone (the un-powered cone type) is an impedance matcher. It exchanges pressure for volume. This is most apparent when you use it backward, as in an ear trumpet, or in the toy I used to use in the swimming pool. We taped plastic wrap over the small end of a traffic cone and spoke into the large end. Swimmers underwater could hear us clearly.

When you shout at the water's surface, most of the sound energy bounces off. There is an impedance mismatch between the sound pressure needed in air and that needed in water. Matching the impedance allows more energy to transfer to the water.

This reflection effect happens to radio waves too. If the impedance is not matched, the energy is reflected to the transmitter, sometimes so much that the output transistor overheats and dies.

A transformer is an impedance matcher. It trades off voltage and current. We measure electrical impedance in terms of ohms. Ohms describe the ratio between voltage and current. Ohm's Law is simply that ratio. Voltage divided by current.

In direct current circuits, ohms are that simple. In radio frequencies, there are inductances (think

coils of wire) and capacitors, both of which store energy, so that the voltage in a signal may lead or lag the current. Because this changes the ratio of voltage to current at any instant, it is expressed in ohms.

Voltage Standing Wave Ratio (VSWR)

Voltage standing waves.

Wikimedia Commons

When impedances are not matched, we get reflected energy. We can measure the voltage going in one direction and the voltage going in the reverse direction with a clever use of diodes.

If you tie a rope to a tree and send a pulse wave down the rope by jerking it up and down, the wave will travel to the tree and then bounce back to you. If you keep jerking the rope at a certain frequency, you will get a “standing wave” in the rope. This happens in radio a lot. Measuring the voltage in both directions gives us the voltage standing wave ratio (VSWR), a measure of how well we have matched the impedances.

If the tree was small and absorbed all of the energy in the rope, it would wave in the air and there would be no reflected wave back to us. In a radio circuit, we would see the voltage, but the current in the reverse direction would be negligible. The VSWR would be close to 1, and there would be no reflected energy.

Bandwidth

Impedance matching is a function of wavelength. This is because inductors and capacitors behave differently at different frequencies. Inductors resist changes in current, so the faster the changes happen, the more they resist. Capacitors are open circuits at low frequencies and work better at higher frequencies (the opposite of inductors).

If we plot the VSWR as we increase the frequency, there will be a dip around the resonant frequency of the antenna. We can measure the part of the dip that is below, say a VSWR of 2, and say the width of that area is the band of frequencies that we can effectively transmit using that antenna. This is bandwidth, and how it got its name.

The faster we send information, the more often the signal has to change. This is reflected in the bandwidth. It takes more radio spectrum to send high-speed data.

Antenna Tuning

There are several things we might want to optimize when building an antenna.

If we design the antenna to match the impedance of our transmitter, we can avoid using a transformer to do the impedance match for us. We saw how to do this with the ground plane antenna.

But we can also tune the antenna to the frequency we want to transmit on. A highly resonant antenna rejects frequencies outside of its narrow bandwidth. This can increase the signal-to-noise ratio, allowing for longer-range communication.

On the other hand, we might want to be able to change frequencies a lot, so we can talk or listen to different transmitters or stations. We would tune the antenna to have a wider bandwidth. The log-periodic antennas that used to be the way we all received television signals are an example. They are directional, but extremely wide band, since television channel 2 starts at 54 megahertz, while channel 13 ends up at 216 megahertz.

Radiation pattern

An isotropic antenna sends and receives from all directions, like our light bulb. Most antennas are not isotropic, since sending signals into the ground or into space is not an effective way to communicate. A dipole antenna doesn't send or listen in the direction of the ends of the wire. If we hold it up vertically, little energy is wasted sending it into the ground or space.

Our Yagi-Uda antenna sends and listens much more in one direction than others. A three-dimensional plot of its radiation pattern looks like this:

Press enter or click to view image in full size

Yagi-Uda radiation pattern.

Yagi-Uda radiation pattern (Wikimedia Commons)

It has side lobes caused by diffraction, but most of the energy is in the big lobe in the X direction. Almost no noise comes in from behind the reflector.

Effective Area

There is an important concept in antenna theory called the “effective area”. It is the area blocked by a hypothetical sphere around an isotropic antenna.

[Press enter or click to view image in full size](#)

Antenna effective area.

Wikimedia Commons

The higher the frequency, the smaller this sphere becomes. It is an imaginary sphere, just a concept, but we know its area, and can thus know its radius (divide the area by π). The reason this hypothetical sphere gets smaller is because the radiation resistance of an antenna is a function of frequency. The higher the frequency, the more the antenna resists putting out radio energy. Even though we don't have an isotropic antenna, this concept allows us to calculate the range of our communications.

Once we know the effective area, if we have the transmitter power, the antenna gain, and the distance we want to communicate, we can get the strength of the signal at the receiver.

The power of the HC-12 is 100 milliwatts.

The power falls as the square of the distance. This is the “inverse square law”, and is pretty simple to understand once you remember that the formula for the area of a sphere is 4 times π times the radius squared. The radius in this case is the distance between the transmitter and the receiver. The power is a function of the part of the area of a sphere with that radius that is captured by the receiver's antenna: its effective aperture.

Since we have Python at our disposal, let's encapsulate all of that in a little program.

```
From math import sqrt, pi, log10
```

```
Min_receive_dBm = -117
```

```
Max_transmit_dBm = 20
```

```
Realistic_receive_dBm = -80
```

```
Freq = 433400000
```

```
Tx_gain = 12
```

```
Rx_gain = 12
```

```
Def dBm_to_watts(dBm):
```

```
Return 10 ** ((dBm - 30) / 10)
```

```
Def speed_of_light():
```

```
Return 299792458
```

```
Def wavelength(frequency):
```

```
Return speed_of_light() / frequency
```

```
Def path_loss_dB(distance, frequency):
```

```
Return 20 * log10(distance) + 20 * log10(frequency) + 20 * log10((4 * pi) / speed_of_light())
```

```
Def link_budget(max, min):
```

```
Return max - min
```

```
Def effective_aperture(frequency):
```

```
Return wavelength(frequency) ** 2 / (4 * pi)
```

```
Def received_power(frequency, tx_power, tx_gain, distance):
```

```
Return tx_power / (4 * pi * distance ** 2) * tx_gain * effective_aperture(frequency)
```

```
Def dist(frequency, watts, tx_gain, rx_gain):
```

```
Return (((4 * pi) ** (-1/speed_of_light())) / frequency) * watts * tx_gain * rx_gain
```

```
Def nanowatts(watts):
```

```
Return watts * 1000000000
```

```
Def picowatts(watts):
```

```
Return watts * 1000000000000
```

```

Def femtowatts(watts):
Return watts * 10000000000000000
Print()
Print("Frequency:".ljust(40, ' '), freq / 1000000, "MHz")
Print("Minimum receive power:".ljust(40, ' '), picowatts(dBm_to_watts(realistic_receive_dBm)),
"picowatts")
Print("Maximum transmit power:".ljust(40, ' '), 1000 * dBm_to_watts(max_transmit_dBm), "milliwatts")
Print("Wavelength:".ljust(40, ' '), round(wavelength(freq) * 100, 2), "centimeters")
Print("Effective aperture:".ljust(40, ' '), round(effective_aperture(freq) * 10000, 2), "square
centimeters")
Print("Received power at 1 km:".ljust(40, ' '), round(picowatts(received_power(freq, .1, 1, 1000)),
2), "picowatts")
Print()
Print("Link budget:".ljust(40, ' '), link_budget(max_transmit_dBm, realistic_receive_dBm), "dB")
Z = 0
X = 0
Budget = link_budget(max_transmit_dBm, realistic_receive_dBm)
While z < budget:
Y = 2 ** x
X += 1
Z = path_loss_dB(y, freq)
If z < budget:
Print(("Path loss at " + str(y) + " meters:".ljust(40, ' '), round(z), "dB")
Print()
Print("With 12 dBi antennas at each end:")
Z = 0
X = 0
Budget = link_budget(max_transmit_dBm, realistic_receive_dBm)
While z < budget:
Y = 2 ** x
X += 1
Z = path_loss_dB(y, freq) - tx_gain - rx_gain
If z < budget:
Print(("Path loss at " + str(y) + " meters:".ljust(40, ' '), round(z), "dB")

```

The output for our HC-12 looks like this:

Frequency:	433.4 MHz
Minimum receive power:	10.0 picowatts
Maximum transmit power:	100.0 milliwatts
Wavelength:	69.17 centimeters
Effective aperture:	380.76 square centimeters
Received power at 1 km:	303.0 picowatts
Link budget:	100 dB
Path loss at 1 meters:	25 dB
Path loss at 2 meters:	31 dB
Path loss at 4 meters:	37 dB
Path loss at 8 meters:	43 dB
Path loss at 16 meters:	49 dB
Path loss at 32 meters:	55 dB
Path loss at 64 meters:	61 dB
Path loss at 128 meters:	67 dB

Path loss at 256 meters:	73 dB
Path loss at 512 meters:	79 dB
Path loss at 1024 meters:	85 dB
Path loss at 2048 meters:	91 dB
Path loss at 4096 meters:	97 dB

With 12 dBi antennas at each end:

Path loss at 1 meters:	1 dB
Path loss at 2 meters:	7 dB
Path loss at 4 meters:	13 dB
Path loss at 8 meters:	19 dB
Path loss at 16 meters:	25 dB
Path loss at 32 meters:	31 dB
Path loss at 64 meters:	37 dB
Path loss at 128 meters:	43 dB
Path loss at 256 meters:	49 dB
Path loss at 512 meters:	55 dB
Path loss at 1024 meters:	61 dB
Path loss at 2048 meters:	67 dB
Path loss at 4096 meters:	73 dB
Path loss at 8192 meters:	79 dB
Path loss at 16384 meters:	85 dB
Path loss at 32768 meters:	91 dB
Path loss at 65536 meters:	98 dB

Each 3 dB of gain is a doubling of signal strength. Our 12 dBi antenna is like a 16x telescope (doubling 4 times is 16).

To the transmitter, this looks like multiplying the output power.

To the receiver, it looks like multiplying the received signal strength, but at the same time reducing the noise level, since just like a telescope, the receiver now sees less noise-making landscape.

Our range just went from 4 kilometers to 65 kilometers.

All from a bit of foam core and some copper tape.

Radio

Python Radio 13: Software Defined Radio

Simon Quellen Field

Simon Quellen Field

Follow

6 min read

.

Aug 31, 2024

Listen

Share

More

Python listens to the radio.

Press enter or click to view image in full size

Photo by the author

If you do an Internet search for “SDR receiver price” you will find a large number of items for sale, from about \$15 to \$300. For our purposes (since we want to receive low frequencies as well as those above 30 MHz) we want receivers that can reach at least as low as 100 kilohertz. I have seen one for \$18 that gets as low as 10 kilohertz.

What makes these devices special is the software on your computer that controls them and decodes the

signals.

Another search, “SDR radio software” gets you a large number of software packages, most of them free, with names like SDR#, HDSDR, SDR-RADIO.COM, SDR++, Linrad, GQRX, CubicSDR, and many. There is also a Python library called pyrtlsdr, installed by the command `pip install — upgrade pyrtlsdr[lib]`.

Using this library, the following program will pop up a spectrum display centered around the 7032000-hertz signal from our RP2040 CW transmitter:

```
From pylab import *
From rtlsdr import RtlSdr
Import numpy as np
Import scipy.signal as signal
Import peakdetect
Real_center_freq = 7.032e6
Offset = 200e3
Margin = 10e3
Sdr = RtlSdr()
Sdr.set_direct_sampling(1)
Sdr.sample_rate = 225001
Sdr.center_freq = real_center_freq – offset
Sdr.gain = ‘auto’
Num_samples = sdr.sample_rate
Samples = sdr.read_samples(num_samples)
Power, psd_freq = psd(samples, NFFT=1024, Fs=sdr.sample_rate, Fc=real_center_freq)
Power_db = 10*np.log10(power)
Maxima, minima = peakdetect.peakdetect(power_db, psd_freq, delta=1)
For mx in maxima:
F = mx[0]
dBm = mx[1]
print(“Peak at”, f, “of”, dBm, “dB”)
# Was this peak anywhere near our target frequency?
If f > real_center_freq-margin and f < real_center_freq+margin:
Print(“We see a peak at”, str(f))
Sdr.close()
Show()
```

We set direct sampling to 1 so that we can reach the low frequencies. Then we read a second of samples from the receiver and plot them:

Press enter or click to view image in full size

Image by the author

The output of our program on the console looks like this:

Found Rafael Micro R820T/2 tuner

Enabled direct sampling mode, input 1

Exact sample rate is: 225001.000804 Hz

Peak at 6983879.6687734295 of -77.32257246972893 dB

Peak at 7032000.0 of -70.93046957472468 dB

We see a peak at 7032000.0

We chose the smallest sample rate that the RTL-SDR is capable of. It can sample in two ranges: 225 kHz to 300 kHz and 900 kHz to 2032 kHz (although rates higher than about 2.4 MHz drift unreliably). In the RTL-SDR, the sample rate is also the bandwidth, and for CW we want a narrow bandwidth (usually less than a kilohertz, but 225 kHz will do for now).

We can zoom in on the spectrum to see more detail:

[Press enter or click to view image in full size](#)

[Image by the author](#)

Let's look at some of the aspects of the code that are not immediately obvious.

We set direct sampling because we are looking at the low frequencies (below 30 MHz). Without that, we can't reach those frequencies.

We have an offset of 200,000 hertz. The RTL-SDR samples at the center frequency we give it, and returns us an array of samples centered at zero. This would give us what is called an "artifact", something that looks like a signal but is not actually there in the real world. By offsetting the center frequency by a couple of hundred kilohertz, we can get rid of this artifact.

The actual collecting of the samples is done by `read_samples()`. We could plot the array it returns to get an oscilloscope trace of our signal (we say that data is in the time domain). But we want to see the data in the frequency domain, so we use the "power spectral density" method `psd()` to get a Fast Fourier Transform (FFT) of the data.

Again, we could simply plot that (in fact, we do plot that on the last line of the program). But first, we want to find the signal peaks in the data, to see if the signal from our CW transmitter is there. We use the `peakdetect()` method to do that. It returns arrays of tuples that contain the frequency and the power of each peak.

We scan through that data to locate our target signal and print it out.

Another way to view the signals is with a waterfall display. Here is our RP2040 transmitter sending out Morse code CW signals:

[Press enter or click to view image in full size](#)

[Image by the author](#)

The code to do this is mostly involved with the graph itself, but the radio portion is fairly simple. As we did previously, we get the samples and convert them to a power spectral density plot with the `psd()` method. But then instead of drawing a line graph, we use the data to paint a bar of colors, with the lighter colors indicating higher power. We continue painting these bars of color one after another and when we get to the bottom of the image, it scrolls.

The code is shown below. We use a bandwidth of a megahertz centered around 28 MHz:

```
Import matplotlib.animation as animation
From matplotlib.mlab import psd
Import pylab as pyl
Import numpy as np
Import sys
From rtlSdr import RtlSdr
NFFT = 1024*4
NUM_SAMPLES_PER_SCAN = NFFT*4
NUM_BUFFERED_SWEEPS = 100
Class Waterfall(object):
    Keyboard_buffer = []
    Shift_key_down = False
    Image_buffer = -100*np.ones((NUM_BUFFERED_SWEEPS, NFFT))
    Def __init__(self, sdr):
        Self.fig = pyl.figure()
        Self.sdr = sdr
        Self.init_plot()
    Def init_plot(self):
        Self.ax = self.fig.add_subplot(1,1,1)
        Self.image = self.ax.imshow(self.image_buffer,
        Aspect='auto', interpolation='nearest', vmin=-50, vmax=10)
        Self.ax.set_xlabel('Current frequency (MHz)')
```

```

Self.ax.get_yaxis().set_visible(False)
Def update_plot_labels(self):
Fc = self.sdr.fc
Rs = self.sdr.rs
Freq_range = (fc - rs/2)/1e6, (fc + rs*(0.5))/1e6
Self.image.set_extent(freq_range + (0, 1))
Self.fig.canvas.draw_idle()
Def update(self, *args):
Start_fc = self.sdr.fc
Self.image_buffer = np.roll(self.image_buffer, 1, axis=0)
For scan_num, start_ind in enumerate(range(0, NFFT, NFFT)):
Self.sdr.fc += self.sdr.rs*scan_num
Samples = self.sdr.read_samples(NUM_SAMPLES_PER_SCAN)
Psd_scan, f = psd(samples, NFFT=NFFT)
Pwr = 10 * (np.log2(psd_scan)/np.log2(8))
Self.image_buffer[0, start_ind: start_ind+NFFT] = pwr
Self.image.set_array(self.image_buffer)
Self.sdr.fc = start_fc
Return self.image,
Def start(self):
Self.update_plot_labels()
Ani = animation.FuncAnimation(self.fig, self.update, interval=50, save_count=64*1024, blit=True)
Pyl.show()
Return
Def main():
Sdr = RtlSdr()
Wf = Waterfall(sdr)
Sdr.rs = 1.0e6
Sdr.fc = 28000000
Sdr.gain = 'auto'
Wf.start()
Sdr.close()
If __name__ == '__main__':
Main()

```

We turned off Farnsworth sending in our RP2040 code so that we could more easily see the signals in the waterfall. If we send the CW very slowly (something called QRSS mode), we can use a very narrow bandwidth and receive weak signals very far away, since the narrow bandwidth gives us a better signal-to-noise ratio. Instead of listening to the Morse code, we watch it scroll down the waterfall display:

```

From cwmorse import CWMorse
From time import sleep
Frequency = 28050000
Def main():
Cw = CWMorse(15, frequency)
Cw.farnsworth(False)
Cw.speed(0.1)
Print("CW transmitter")
Msg = "This is AB6NY testing RP2040 as a 40 meter transmitter sending on " + str(frequency) + "
Hertz."
While True:

```

```

Print(msg)
Cw.send(msg)
Sleep(5)
Main()
The cwmorse.py code is modified slightly to allow turning off the Farnsworth method:
From machine import Pin, PWM
Class RP_CW:
Def __init__(self, carrier_pin, freq):
From machine import Pin
From rp2 import PIO, StateMachine, asm_pio
@asm_pio(set_init=PIO.OUT_LOW)
Def square():
Wrap_target()
Set(pins, 1)
Set(pins, 0)
Wrap()
Self.carrier_pin = Pin(carrier_pin, Pin.OUT)
Self.f = freq
Self.sm = StateMachine(0, square, freq=2*self.f, set_base=self.carrier_pin)
Self.sm.active(1)
Def on(self):
Self.sm.active(1)
Print("#", end="")
Def off(self):
Self.sm.active(0)
Print(" ", end="")
Def frequency(self, frq):
Self.f = frq
Class CWMorse:
Character_speed = 20
Def __init__(self, carrier_pin, freq):
Self.cw = RP_CW(carrier_pin, freq)
Self.cw.frequency(freq)
Self.farns = True
Def farnsworth( self, on_off):
Self.farns = on_off
Def speed(self, overall_speed):
Print("Farnsworth is", self.farns)
If overall_speed >= 20 or self.farns == False:
Self.character_speed = overall_speed
Units_per_minute = int(self.character_speed * 50)      # The word PARIS is 50 units of time
OVERHEAD = 2
Self.DOT = int(60000 / units_per_minute) – OVERHEAD
Self.DASH = 3 * self.DOT
Self.CYPHER_SPACE = self.DOT
If overall_speed >= 20 or self.farns == False:
Self.LETTER_SPACE = int(3 * self.DOT) – self.CYPHER_SPACE
Self.WORD_SPACE = int(7 * self.DOT) – self.CYPHER_SPACE
Else:
# Farnsworth timing from

```

```

Farnsworth_spacing = (60000 * self.character_speed - 37200 * overall_speed) / (overall_speed *
self.character_speed)
Farnsworth_spacing *= 60000/68500 # A fudge factor to get the ESP8266 timing closer to correct
Self.LETTER_SPACE = int((3 * farnsworth_spacing) / 19) - self.CYPHER_SPACE
Self.WORD_SPACE = int((7 * farnsworth_spacing) / 19) - self.CYPHER_SPACE
Def send(self, str):
From the_code import code
From time import sleep_ms
For c in str:
If c == ' ':
Self.cw.off()
Sleep_ms(self.WORD_SPACE)
Else:
Cyphers = code[c.upper()]
For x in cyphers:
If x == '.':
Self.cw.on()
Sleep_ms(self.DOT)
Else:
Self.cw.on()
Sleep_ms(self.DOT)
Self.cw.on()
Sleep_ms(self.DOT)
Self.cw.on()
Sleep_ms(self.DOT)
Self.cw.off()
Sleep_ms(self.CYPHER_SPACE)
Self.cw.off()
Sleep_ms(self.LETTER_SPACE)

```

Software Defined Radio

Rp2040

Software Defined Radio & Radio Hacking. Part 1

Follow

10 min read

Aug 28, 2023

Listen

Share

More

Understanding SDR is essential for mastering cyber security disciplines.

Note: The legalities behind the radio spectrum vary from country to country. While generally speaking receiving is okay, in some places this may be still prohibited so you should check your local laws before starting. Transmission of signals outside the licence free ISM band requires a license, again please check your locality for situation specific details. Direction finding can be done in seconds with the correct setup... so play nicely.

If you were around in the early days of the internet you might recall it was a much simpler time.

We had simpler web pages, less security and the proliferation of social media and smart devices hadn't occurred yet, meaning the art of cyber security was much easier to master. This doesn't mean the early internet didn't have good things. Some classic pieces of software like Snort were designed literally decades ago yet remain entirely relevant even today.

However over time as the world and how we communicate changed, we see more importance given to specialized roles that are essential in understanding just how technology works, and more importantly how it can be exploited. Then subsequent to that, how something can be protected. As the internet slowly changed and we saw the growth of IOT devices, how these devices connect and engage with the world also changed. In the private world, we saw cloud based assets grow, and private CCTV networks explode into everyday usage

In the commercial world we saw the development of things like aircraft ADS-B and marine AIS systems bring information to the masses, allowing anyone with a Web connection and a browser to browse traffic globally in an instant. We also saw the development of things like smart electricity meters and point to point links for information sharing, allowing the growth of integrated RF systems to replace the role of many manual jobs.

Because of these changes in our world, we see that some skill sets grow and evolve right along side our technology. Everyday we see solar installers using RF systems to monitor power generation, and garage door installers using RF triggers to open garages at the push of a button. Starlink and Iridium signals fly overhead, transmitting conversations, internet and telemetry the whole way. All of these can be explored with some SDR knowledge.

When we look at all of this in context and try to understand how we analyse the world around us, we start to realize that we need to cultivate certain skill sets to effectively engage with these new forms of technology. Without a doubt one of these essential skills to master, is Software Defined Radio.

In Part 1 of today's article, we'll be looking further into SDR and radio hacking, attempting to understand exactly what it is and why exactly we consider this an essential skill for the cyber security professional. Let's get started.

What Is It:

We've actually covered simple software defined radio before, in this article, as well as discussing the different types of emissions as well in this . However these were more generalized, and today we'll be digging deeper into some of the specifics.

In the interests of providing more clarity, we can say that Software Defined Radio (SDR) is a technology that allows you to use software to control and manipulate radio signals, instead of using traditional hardware components. It provides flexibility and versatility in terms of signal processing and modulation techniques.

In a nutshell, this effectively means that rather than needing a radio to intercept and analyse radio transmissions, we can now use our computer. But more importantly, rather than simply using and relying on voice transmissions, we can now use digital signals and modes linked to our computer. We can also record, analyse and play back or re transmit signals near instantly, whilst also keeping an eye on the spectrum at the same time.

This is a lot to take in for beginners, so let's pause here and ask Chat GPT for a quick summary of some of the benefits of SDR compared to analogue radio.

Input: Please explain some of the benefits of SDR over traditional radio

Output: Sure! Some benefits of Software Defined Radio (SDR) over traditional radios are:

1. Flexibility: SDR allows for easy reconfiguration and adaptation to different communication standards and protocols.
2. Cost-Effectiveness: SDR reduces the need for specialized hardware components, making it more cost-effective to implement and upgrade radio systems.
3. Upgradability: SDR systems can be updated with new features and functionalities through software updates, extending their lifespan and keeping them up-to-date.
4. Spectrum Efficiency: SDR enables efficient utilization of the available spectrum by adapting to changing conditions and optimizing signal processing techniques.
5. Interoperability: SDR can support multiple communication standards, making it easier for different systems to communicate with each other.

6. Development and Experimentation: SDR provides a platform for researchers and developers to experiment with new radio technologies and algorithms.

Overall, SDR offers increased flexibility, cost savings, and the ability to adapt to evolving communication needs.

Okay, so this seems pretty clear cut. Let's look at how we can get started learning about SDR in a cyber security capacity.

Press enter or click to view image in full size

The Radio spectrum, from DC to Daylight. It's a vast playground for those willing to learn to ropes.

Source: FCC

Where to Begin:

Before we begin, it's important to understand your own limitations and learning path here.

Do you wish to have a cursory understanding or something deeper? Do you have a hardware budget, or are there financial limitations that restrict you from obtaining what you need?

While none of these will make your journey impossible, like anything clearly setting and understanding both goals and limitations will allow a clear learning path whilst creating a good environment for said learning.

It's also important to understand before we start that when you pair SDR and then the concept of hacking, we start to see how vast the world is around us while taking a distinct step away from networks and traditional forms of hacking. While it's still present and does overlap into our article today, the point is that mastering SDR gives you far more options in the tool kit and because of this, we move into some rather exciting and specific applications.

Want to hack a satellite?? You'll need SDR experience to master uplinks and down links, as well as dealing with Doppler effect and intercepting telemetry from beacons in space. This is no joke, check out our screen cap from the latest contest in the United States showing different event types.

Press enter or click to view image in full size

Hack a Sat 4. The dream for space enthusiast's. Source: Hack-aSat.com

Want to fly Kismet on a drone? You'll still use Wi-Fi for some parts of this, but you'll need radio frequency spectrum experience to deal with channels, spacing, antennas and optimizing your setup. Perhaps your interest is Geolocation of transmissions and devices? Understanding where to look for different transmissions and how to optimally Geolocate signals means you'll obtain experience with more antennas, feed lines and software that can be used to assist you in your quest.

As you can see, the addition of SDR to our toolkit gives us a vast array of choices beyond traditional networking and computers. And this is before we look at things like Bluetooth devices, wireless camera and ISM band devices and beyond that, spectrum specific functions like location spoofing and other offensive applications. So if you like to push the limits of learning, there's plenty of specialties for you to focus on should you choose.

Press enter or click to view image in full size

A visual look at Broadband GPS Jamming conducted in a lab environment. Source: twitter.com

Now we've broken that down, it's time to see how we start our journey. Lets look at some non hardware options first as that's the simplest way to get started.

Online Tools: Web SDR

Proliferating thanks to Ham Radio operators the world over, Web SDR relies on web portals streaming online to provide an open access system that basically anyone can use. While you'll find traditional Web SDR portals have good hardware, you'll typically find they tend to focus on amateur radio bands. While some of this can be considered to be menial "chit chat", the amateur bands also included dedicated space communications bandwidth. This allows for the interception and transmission of both voice and data modes by anyone with an appropriate license and equipment.

Press enter or click to view image in full size

Web SDR Dashboard for the Qatari Es'hail-2 Satellite. Source: WebSDR.org

While the Ham Bands do come with limitations, they can be a great way to get started with SDR, with

users able to learn how to tune frequencies, record transmissions, adjust bandwidth, IF and other essential SDR skills.

If we've got your attention, then you should check out [this](#) to get started with the non hardware approach.

Hardware Methods:

When looking at our Hardware approach we see a two different options available to us. We have receive only and receive / transmit capable devices.

Receive only devices come in a number of variants, with probably the RTL-SDR being the cheapest and most prolific of these devices. Based on a modified TV tuner, these devices unlocked software defined radio to the masses due to the fact they are able to be procured extremely cheaply.

The other benefit of RTL devices is the proliferation of free, open source software to power it.

This means that many digital modes are able to be decoded with a simple software patch, allowing many signals that are outside the scope of an analogue radio become trivial to intercept.

RTL-SDR powered by the R2832U Chipset. Source: Wikipedia.

The down side of these devices is that due to their cheap nature and typically plastic housing, they can be finicky, subject to interference and signal drift. While they are still a useful tool, they can be frustrating for beginners and any serious professional should be considering devices that come with a metal housing, and a crystal oscillator to provide legitimate frequency stability.

[Press enter or click to view image in full size](#)

One of the biggest SDR benefits is seeing large chunks of spectrum along with multiple signals. This is the HF amateur band showing many different types of signals. Source: RTL-SDR.com

The next step up from these devices are systems like the HackRF or Blade RF. Fully transmit and receive capable across a wide range of the spectrum these are the types of devices our radio hacker will find useful, providing both interception and re transmission capability in a rugged housing with plenty of bandwidth as well as the much needed frequency stability that many of the cheaper RTL devices lack.

[Press enter or click to view image in full size](#)

Hack RF one. A half duplex, transmit capable SDR. All will be explained by part 2. Source: Wikipedia.

While we'll leave the utility of this device for part two of this series what we can say about these types of SDR units is that they are capable of re transmitting many different types of signals, including many well known open source types. This might include Video, Loran or a Spoofed GPS Signal. There's also propitiatory signals like digital radio, car and garage remotes and much more. If you'd like a look at some of the signals you might see while conducting this type of research, you should check out the [wiki](#) for both visual and audio captures of common transmission types.

[Press enter or click to view image in full size](#)

More Resources

So at this point in our article, you now know where to source Signal Information as well as what type of hardware to select to get your journey started. You also know where to get started with online SDR right now. In Part 2 of this article, we'll break out our shiny new SDR and use it for some Signals Intelligence, focusing on direction finding and intercepting some basic space communications. We'll also show you how to build some simple antennas for your new system and discuss the importance of antennas in making this whole thing work. Antenna Theory is probably worth it's own specific article!

However if you'd like to supercharge your learning journey in between reads, we'd love to provide you with some extra reading and video resources to send you along your way.

OccupyTheWeb has a full course on SDR for Hackers that's a brilliant learning resource for those interested in taking their learning to the next level. Check it, along with many other free resources at his website

OTW should be a familiar shadow in cybersecurity. Source: Amazon.com

has interviewed OccupyTheWeb, as well as providing many useful video based, SDR resources on his YouTube channel. Check it out [here](#).

Press enter or click to view image in full size

David's YouTube is a goldmine of useful information. Source: [YouTube.com](#)

The RTL-SDR website has a blog, new and a vast array of SDR based articles based on many different disciplines. Find them [here](#).

And of course we'll continue releasing our own tutorials, which you can receive by following our socials and subscribing for email updates on your platform of choice. See all our relevant details [below](#).

■ Enjoyed this article? Support our work and join the community! ■

■ Support me on Ko-fi:

■ Join our channel for exclusive updates or.

■ Follow us on

■ Articles we think you'll like:

OSINT Unleashed: 5 Essential Tools for Cyber

What the Tech?! GPS

✉ ■ Want more content like this? Sign up for email updates

Sdr

Radio

Technology

Python Radio 6: Field Strength Meter

Simon Quellen Field

Simon Quellen Field

Follow

4 min read

.

Aug 23, 2024

Listen

Share

More

How strong is my signal?

Press enter or click to view image in full size

Photo by author

A helpful tool for debugging radio projects is a simple field strength meter. It is just an antenna, a diode, and a meter:

Press enter or click to view image in full size

Photo by author

In this case, the antenna is a loop of wire attached to the meter at one end and connected to a diode at the other end. The diode is then attached to the other terminal of the meter. The cathode of the diode (the side with the band) is on the side connected to the meter so that when a radio signal arrives, the meter swings in the positive direction.

Press enter or click to view image in full size

Photo by author

We use a germanium 1N34A diode because it is more sensitive than a silicon diode. Even so, our little 10-milliwatt radio transmitter must be very close to the wire loop to move the needle.

But our little computer has an analog input pin that can read voltage levels and convert them into numbers. So we can replace the analog meter with our computer and print the numbers out on the screen.

We remove the wire loop and its diode from the meter and connect the diode to the A0 pin of the computer. We connect the other end of the wire to the ground pin.

Our program is once again very simple:

```
From machine import ADC
```

```
Pwr = ADC(0)
```

```
While True:
```

```
    Avg = 0
```

```
    For count in range(20):
```

```
        Avg += pwr.read()
```

```
    Avg /= 20
```

```
    Print(avg)
```

We import the ADC (Analog to Digital Converter) class from the module machine.

Our little computer only has one analog input, so we will always use ADC zero. Later, on beefier machines, we can use more. Then we loop forever, averaging 20 readings and printing them. Feel free to average 100 or 1000, I picked the number 20 out of a hat.

Now as we move our transmitter closer to the receiver, we can watch the numbers get bigger.

Instead of numbers, let's make a bar graph out of LED lights to indicate the signal strength. We can get strips of Red/Green/Blue LED lights called NeoPixels by the yard, and we can cut them into any length we wish. I chose to use 16 of them for this project, but you can use three or a hundred if you wish.

NeoPixels are very easy to use. They have only three things to connect, no matter how many lights you use. The connections are power (the +5 volts from the computer), ground, and data. We will use pin D4 as the data pin (GPIO 2).

MicroPython has a convenient NeoPixel module built in. Our code now looks like this:

```
From machine import Pin, freq, ADC
```

```
From time import sleep, sleep_ms
```

```
From neopixel import NeoPixel
```

```
PIN_D4 = 2
```

```
Def main():
```

```
    Freq(160000000)                                # 160 MHz
```

```
    Pwr = ADC(0)
```

```
    Np = NeoPixel(Pin(PIN_D4), 16)
```

```
    While True:
```

```
        Avg = 0
```

```
        For count in range(20):
```

```
            Avg += pwr.read()
```

```
        Avg /= 20
```

```
        How_many_lights = avg * 16 / 1024
```

```
        For i in range(16):
```

```
            If i < how_many_lights:
```

```
                Np[i] = (0, 2, 0)
```

```
            Else:
```

```
                Np[i] = (0, 0, 0)
```

```
        Np.write()
```

```
    Main()
```

Just for fun, I doubled the clock speed to 160 megahertz to show you it can be done. It uses the battery up twice as fast at this rate, and our program doesn't actually benefit from the extra speed, so feel free to omit that.

The program is basically the same since the only addition is the NeoPixel code.

We import the NeoPixel class, and then create an instance of it called np, telling it to use pin D4, and that there will be a string of 16 lights.

We convert the average reading (which would be between 0 and 1023) into a number between 0 and 1

and call that `how_many_lights`.

Then we loop through the 16 lights, assigning them either (0, 2, 0) [a dim green] or (0, 0, 0) [completely dark], based on how many lights we want in our bar graph. I chose green, but you can play with any colors you wish. The lights can be very bright, which is why I chose the number 2, which is less blinding. Using the number 1 might even be better, and would allow longer battery life when running on batteries.

The end result looks like this:

[Press enter or click to view image in full size](#)

Photo by author

The schematic for the circuit looks like this:

[Press enter or click to view image in full size](#)

Image by author

What The Tech?! Microwave Emissions

Investigator515

Investigator515

Following

6 min read

.

Aug 17, 2025

[Listen](#)

[Share](#)

[More](#)

The GigaHertz bands would help us to achieve true, cable-free, high-speed data exchange.

If you aren't a Medium member, you can read with no paywall via Substack

Today's world is a world of connectivity. We stream our music from remote servers, access our documents via the cloud and even have the capacity to share our work and be based remotely. And, while fibre optic systems would carry the bulk of the load regarding these technological shifts, it would take microwave systems to be able to implement the true, high-speed, wireless connectivity that we'd need to flourish. Today, the GHz spectrum is the star of this What The Tech!?, and it has little to do with the microwave oven. Let's take a look!

The Beginnings

As we began to discover the radio spectrum, the ramifications would be global. Most of us have heard of the Titanic, but what's less known is that the proper use of radio could potentially have averted a crisis and changed the course of history.

Early research would focus on the High Frequency band. Here, the antennas were big, and so were the wavelengths, but thanks to the ionosphere, we'd get global propagation potential. This was a huge deal, considering the fixed wire Telegraph was about as good as it got at that point.

People were so stuck on HF that the higher frequencies were deemed to be useless. So much so that they would initially be assigned to radio amateurs (hams) for experimentation purposes.

While it would take some time for things to unfold and play out to fruition, this would be a decision that would have lasting consequences on our societies.

It's worth mentioning that at this point, no one person would be entirely responsible for the discovery of the microwave bands. However, we'd see many influential names that would help contribute much to our knowledge bank. Maxwell, Marconi, Hertz and Bose are all familiar names in today's world, and all these people would help us to understand the radio spectrum as intimately as we do today. Our modern communications systems owe a debt of gratitude to all these early pioneers.

The cavity magnetron would be revolutionary, but it would also take time to be brought to life.

Source: Wikipedia.

The Cavity Magnetron

Like most pivotal inventions, it would take time for the manufacturing technology to catch up to the

research. And, for microwaves, it would be the invention of the Cavity Magnetron that would turn the concept on its head. A critical component in early radar systems, the magnetron would be designed and implemented thanks to research done during World War 2.

Effectively an oscillator, the magnetron generates microwave frequency radio waves thanks to a magnetic field and a set of resonant cavities.

While vacuum tube technology was not new at this point, the technological shift would come from two different aspects. First, there was the overall stability of the magnetron. Previous vacuum-based, RF designs were known for frequency drift as they heated up, which would tend to get worse as you moved higher up the spectrum. For the first time, the magnetron would provide a stable source of RF that would not drift over time.

[Press enter or click to view image in full size](#)

It's small, compact size would allow the magnetron to take flight. Here's an early Soviet 9GHz unit from an early fighter jet. Source: Wikipedia.

The other aspect would be its actual size. Radio systems at the time were bulky, cumbersome systems that were most certainly not portable. The magnetron would change this, giving a compact unit that was capable of hundreds of watts or even kilowatts of output. This would make it perfect for airborne radar systems and other military applications. However, it wouldn't take long for many of these applications to filter down into the civilian world.

[Press enter or click to view image in full size](#)

When we said "compact with high power", it was true! This 1947 commercial radar magnetron puts out over 20,000 watts in the 9 Gigahertz band. Don't stand in front of one. Source: Wikipedia.

Modern Evolutions

We've spoken much about the technology bonanza that the world would benefit from in the aftermath of WW2, and communications were no different. The world would see the benefits of technology like portable radio and aircraft radar, and as such, it didn't take long for it to start being applied in the civilian world.

This was a time of immense change across the board. In aviation, for instance, the North American P-51 Mustang would be a cutting-edge, top-tier fighter aircraft. Yet, by the time of the Korean War just a few years later, it would be deemed mostly obsolete, replaced by Generation 1 fighter jets like the Gloster Meteor and the North American F-86 Sabre.

The F-86 Sabre would introduce airborne radar thanks to the Magnetron and Microwaves. Source: Wikipedia.

While jet engines would be one component that would help make the distinction between the different types of aircraft, the other would be the addition of a radar-based, electronic gun sight. The introduction of the AN/APG-30 radar system would lay the foundations for the modern fast jets that are so familiar to us today.

Where microwaves would really shine, though would be in the application of television technology. The delivery of signals to the end user would rely on cheaper and easier to implement VHF and UHF systems, but for uplink systems and content sharing, microwaves were king, and as such, were implemented in huge numbers by the time the 1950s came around.

This would become easier to implement thanks to the 1947 Atlanta International Telecommunications Union conference. Here, all participants recognised the value of the microwave bands and would make significant inroads in standardising that part of the spectrum, making it much easier to use. They did such a good job at this that even today, many spectrum allocations that are still used would come from that meeting.

[Press enter or click to view image in full size](#)

It just keeps climbing. Now, 90 GigaHertz and beyond are regularly being used for all kinds of applications. Source: Wikipedia.

The Future

Thanks to the evolution of microelectronics and GPS, it's now much easier to get stable microwave

equipment for all sorts of applications. As such, we'd see more and more of the microwave spectrum being unlocked for use. While the ubiquitous smartphone would be responsible for pushing much of this development, the reality is that all sorts of applications now exist within this chunk of the spectrum.

For much of this, satellite communications is king, but we are also seeing applications of things like mm wave radars, radio astronomy and even broadcast applications.

Here's the thing, though. The full self-driving automobile that everyone is working towards?

Creating projects like this would be a lot harder without microwaves, as GHz transmissions are also perfect for applications like vehicle LIDAR.

The usage of the microwave bands is going nowhere any time soon.

Medium has recently made some algorithm changes to improve the discoverability of articles like this one. These changes are designed to ensure that high-quality content reaches a wider audience, and your engagement plays a crucial role in making that happen.

If you found this article insightful, informative, or entertaining, we kindly encourage you to show your support. Clapping for this article not only lets the author know that their work is appreciated but also helps boost its visibility to others who might benefit from it.

Enjoyed this article? Join the community!

Join our OSINT Telegram channel for exclusive updates or

Follow our crypto Telegram for the latest giveaways

Follow us on Twitter and

■ We're now on Bluesky!

Articles we think you'll like:

What The Tech?! Space Shuttles

Shodan: A Map of the Internet

✉ ■ Want more content like this? Sign up for email updates

Whatthetech

Python Radio 33: FM — No Static At All

Simon Quellen Field

Simon Quellen Field

Follow

9 min read

.

Jan 26, 2025

Listen

Share

More

Building a wide-band radio transmitter

Press enter or click to view image in full size

Screenshot of FM transmitter signal

Screenshot by the author

The bane of radio communication in the AM band is noise, often called “static” because much of it is caused by static electric sparks and lightning.

The man responsible for much of the success of AM radio through his inventions of regenerative amplifiers and superheterodyne receivers worked for years on the task of eliminating static.

The man was Edwin Howard Armstrong.

Removing Static

When signals get sent by modulating amplitude, any extra energy at that frequency shows up as noise in the receiver. This is true for both AM and CW (continuous wave, or Morse code), both of which were used when Armstrong was experimenting.

When an AM signal is over-amplified, the sound in the receiver suffers distortion. The tops and

bottoms of the sine wave are “clipped” and look flattened. Amplified enough, all that remains is a square wave.

This would remove all of the information from the AM signal. But the CW signal is unaffected. The difference between a sine wave and a square wave to the ear of the operator is just a matter of tone.

One important effect, however, is that not only is the AM information gone, but so is the static. We can hear the nice tone, and the crackle, pop, and snap of radio noise is gone.

An FM signal can be amplified until all the noise is gone, and we can still see the frequency shift around as the music and voices are transmitted. This is one of the key benefits of FM.

Another benefit is that an amplifier that does that is more efficient than a linear amplifier needed for AM. The output is either on or off, whereas in AM the average power is half of the maximum, even when there is silence.

Wide Band FM

In the 1930s, AM radio reduced static by using narrow-band transmission. AM stations limited the signal to about 10 kilohertz of bandwidth. The human ear can hear up to 20 kilohertz, and much of the static noise is above 10 kilohertz.

Transmitting FM signals limited to 10 kilohertz was not much of an improvement over AM radio. But Armstrong showed that if the radio frequency bandwidth was much wider than the audio bandwidth (by factor of 10), not only was the noise removed, but the full range of human hearing came through, even beyond 20 kilohertz.

Instead of 10 kilohertz bandwidth, FM stations have 200 kilohertz of bandwidth. This is the other reason (besides the removal of static) that FM radio sounds better than AM radio.

Building Our Transmitter

With a fast enough processor, we can send FM radio through software alone. But such a processor costs \$35, and we are cheap.

But with a \$2 Wemos D1 Mini and a \$3 CJMCU-4713 FM transmitter breakout board, we can have an FM radio station for five bucks. And it transmits farther than the Raspberry Pi software transmitter.

Ours will send high-fidelity signals about 300 feet, more than enough to cover the house and driveway, and possibly a neighbor or three.

[Press enter or click to view image in full size](#)

Our FM Transmitter

Photo by author

Six wires is all it takes.

The transmitter board has two ways to get power. It accepts 5 volts on the Vin pin and 3.3 volts on the 3vo pin (the pin we have used since the D1 Mini also uses 3.3 volts).

Besides power and ground, the board has SCL, SDA, and RST connections.

FM GND → D1 G

FM 3vo → D1 3v3

FM RST → D1 D4

FM SCL → D1 D1

FM SDA → D1 D2

Lastly, we solder a 3-foot length of wire to the transmitter’s Ant pin as an antenna.

The Software

As usual here, the hardware is the easy part.

The chip on the board is the Si4713. This chip has a wonderfully rich set of features. We can select any frequency in the FM band, we can control the power we transmit, we can mute the audio, we can attenuate the incoming audio (not quite enough, however, as we will see), and it can do two important digital things:

It can send RDS data, so you can show your station ID and what song you are playing on a radio with a screen, such as the one in your car. Or you can use this feature just to send arbitrary text.

It can accept digital data instead of audio, using I2S. This means that we don't need the I2S DAC board we used in the previous project. The transmitter chip has it built-in. We can read WAV files and send them straight to the chip. Let's look at the software before we get to the bad news.

See

From time import sleep

From array import array

Class xmit:

I2C_ADDRESS = 0x63

REGISTERS_ADDRESSES = (1, 257, 259, 513, 514, 8448, 8449, 8450, 8451, 8452, 8453, 8454, 8455, 8705, 8706,

8707, 8708, 8709, 8960, 8961, 8962, 8963, 8964, 11264, 11265, 11266, 11267, 11268, 11269, 11270, 11271)

DEFAULT_REGISTERS_VALUES = ((1, 199), (257, 0), (259, 0), (513, 32768), (514, 1), (8448, 3), (8449, 6625),

(8450, 675), (8451, 0), (8452, 190), (8453, 0), (8454, 0), (8455, 19000), (8704, 3),

(8705, 65496), (8706, 2), (8707, 4), (8708, 15), (8709, 13), (8960, 7), (8961, 206),

(8962, 10000), (8963, 236), (8964, 5000), (11264, 0), (11265, 0), (11266, 0),

(11267, 0), (11268, 0), (11269, 0), (11270, 0), (11271, 0))

FREQ_UNIT = int(10e3)

FREQ_MIN = int(76e6)

FREQ_MAX = int(108e6)

FREQ_DEFAULT = 88.8e6

POWER_DEFAULT = 115

MAX_LINE_INPUT_LEVELS_mV_pk = {0: 190, 1: 301, 2: 416, 3: 636}

Def __init__(self, bus, pin_reset, i2c_address = I2C_ADDRESS,

Freq = FREQ_DEFAULT, tx_power = POWER_DEFAULT, stereo = True):

Self._bus = bus

Self._i2c_address = i2c_address

Self._pin_reset = pin_reset

Self.init()

Self.set_frequency(freq)

Self.set_power(tx_power)

Self.stereo = stereo

Def init(self):

Self.power_up()

Self.write_all_registers(self.DEFAULT_REGISTERS_VALUES)

Self.set_frequency(self.FREQ_DEFAULT)

Self.set_power(self.POWER_DEFAULT)

Def reset(self):

Self.init()

Def power_up(self, analog_audio_inputs = True):

Self._assert_reset()

Self._write_bytes(array('B', [0x01, 0x12, 0x50 if analog_audio_inputs else 0x0F]))

Sleep(0.2) # need 110ms to power up.

Def power_down(self):

Self._write_bytes(array('B', [0x11]))

Def _assert_reset(self):

Self._pin_reset.on()

Sleep(0.01)

Self._pin_reset.off()

```

Sleep(0.01)
Self._pin_reset.on()
@property
Def frequency(self):
Self._write_bytes(array('B', [0x33, 1]))
Bytes_array = self._read_bytes(8)
Freq = bytes_array[2] << 8 | bytes_array[3]
Self._frequency = freq * 10e3
Return self._frequency
Def set_frequency(self, freq):
Assert self.FREQ_MIN <= freq <= self.FREQ_MAX
Assert (freq // 1e3) % 50 == 0
Self._frequency = freq
Freq = round(freq // self.FREQ_UNIT)
Self._write_bytes(array('B', [0x30, 0x00, freq >> 8 & 0xFF, freq & 0xFF]))
Sleep(0.2) # need 100ms
@property
Def tx_power(self):
Self._write_bytes(array('B', [0x33, 1]))
Bytes_array = self._read_bytes(8)
Self._tx_power = bytes_array[5]
Return self._tx_power
Def set_power(self, power):
Assert power == 0 or (88 <= power <= 115)
Self._tx_power = round(power)
Self._write_bytes(array('B', [0x31, 0, 0, self._tx_power, 0]))
Sleep(0.2) # need 100ms
Def mute(self, value = True):
If value:
Self._write_bytes(array('B', [0x31, 0, 0, 0, 0]))
Else:
Self.set_power(self._tx_power)
Def mute_line_input(self, value = True):
Self.write_register(0x2105, 0x03 if value else 0x00)
Def set_line_input_level(self, attenuation_level = 3, line_level = None):
Line_level = self.MAX_LINE_INPUT_LEVELS_mV_pk[attenuation_level] if line_level is None else
line_level
Self.write_register(0x2104, ((attenuation_level & 0x03) << 12) | (line_level & 0x3FF))
@property
Def stereo(self):
Return (self.read_register(0x2100) & 0x03) == 0x03
@stereo.setter
Def stereo(self, value = True):
Current_value = self.read_register(0x2100)
Self.write_register(0x2100, current_value & ~3 | (3 if value else 0))
Sleep(0.01) # status: 0x84
Def enable(self, value = True):
Self.mute(not value)
# ===== data access =====
Def _get_element_value(self, reg_address, idx_lowest_bit, n_bits):

```

```

Reg_value = self.read_register(reg_address)
Mask = 2 ** n_bits - 1
Return reg_value >> idx_lowest_bit & mask
Def _set_element_value(self, reg_address, idx_lowest_bit, n_bits, element_value, ):
Reg_value = self.read_register(reg_address)
Mask = 2 ** n_bits - 1
Return reg_value & ~(mask << idx_lowest_bit) | ((element_value & mask) << idx_lowest_bit)
# ===== data access =====
Def _read_bytes(self, n_bytes):
Return self._bus.readfrom(self._i2c_address, n_bytes)
Def _write_bytes(self, bytes_array):
Self._bus.writeto(self._i2c_address, bytes_array)
Sleep(0.01) # wait for CTS
Return self._read_bytes(1)[0]
Def _set_property(self, address, value):
Self._write_bytes(array('B', [0x12, 0,
Address >> 8 & 0xFF, address & 0xFF,
Value >> 8 & 0xFF, value & 0xFF]))
Sleep(0.01) # set_property takes 10ms
Def _get_property(self, address):
Self._write_bytes(array('B', [0x13, 0, address >> 8 & 0xFF, address & 0xFF]))
Bytes_array = self._read_bytes(4)
Return bytes_array[2] << 8 | bytes_array[3]
Def read_register(self, reg_address):
Return self._get_property(reg_address)
Def write_register(self, reg_address, value):
Return self._set_property(reg_address, value)
Def read_all_registers(self):
Addressed_values = []
For address in self.REGISTERS_ADDRESSES:
Try:
Value = self.read_register(address)
Addressed_values.append((address, value))
Except:
Pass
Return addressed_values
Def write_all_registers(self, addressed_values):
For (address, value) in addressed_values:
Try:
Self.write_register(address, value)
Except:
Pass
Def status(self, status):
S = ""
If status & 0x80:
S += "CTS "
If status & 0x40:
S += "ERR "
If status & 4:
S += "RDSINT "

```

```

If status & 2:
S += "ASQINT "
If status & 1:
S += "STCINT "
Return s
Def set_ps(self, ps_id, a1, a2, a3, a4):
Status = self._write_bytes(array('B', [0x36, ps_id, a1, a2, a3, a4]))
Print("Status is", hex(status), self.status(status))
Def set_ps_string(self, s):
For x in range(4-len(s)%4):
S = s + " "
For x in range(len(s) / 4):
Self.set_ps(x, s[0], s[1], s[2], s[3])
S = s[4:]
Def set_rds_buff(self, location, a1, a2, a3, a4):
If location == 0:
First = 6
Else:
First = 4
Status = self._write_bytes(array('B', [0x35, first, 0x20, location, a1, a2, a3, a4]))
Print("Status is", hex(status), self.status(status))
Def set_rds_buff_string(self, s):
For x in range(4-len(s)%4):
S = s + " "
For x in range(len(s) / 4):
Self.set_rds_buff(x, s[0], s[1], s[2], s[3])
S = s[4:]
Def enable_rds(self):
Self.write_register(0x2100, 7)
Self.write_register(0x2103, 200)
Self.write_register(0x2C01, 0x40A7)
Self.write_register(0x2C02, 3)
Self.write_register(0x2C03, 0x1008)
Self.write_register(0x2C04, 3)
Self.write_register(0x2C05, 1)
Self.write_register(0x2C07, 8)
From machine import Pin, I2C
D1 = Pin(5)
D2 = Pin(4)
D4 = Pin(2)
Def main():
I2c = I2C( scl=D1, sda=D2, freq=100_000)
X = xmit(i2c, D4)
x.enable()
x.set_frequency(90_100_000)
x.set_power(115)
x.set_line_input_level(3, None)
x.enable_rds()
x.set_ps_string(b"Birdfarm")
x.set_rds_buff_string(b"This is a test of the RDS system.")

```

main()

The chip uses I2C to talk to the D1 Mini. The driver (the class xmit) mostly consists of code for getting data in and out of the chip.

In main(), we set up the I2C link and initialize xmit. Then we enable transmitting, set the frequency to 90.1 Mhz, set the power to the maximum 115, and set the line-in attenuator to the maximum since the chip can only handle about half a volt, and we have a 3.3-volt signal coming.

The Bad News

Why do we care about the audio in level? Won't we just be sending I2S digital data to the chip?

Here's the bad news. All of the breakout boards I have found for the Si4713 seem to be clones of the board from Adafruit. For some reason, Adafruit decided that the I2S feature of the chip was not useful, and the board layout ties the digital clock and digital data lines to ground. This completely eliminates one of the nicest features of the board.

I was highly annoyed to find this out, as I had spent days writing code to support the feature so that the transmitter could read a list of files from a microSD card, and send them to the chip along with the meta-data for the WAV file so you could read it on the receiver's RDS screen.

Our transmitter thus needs an audio feed coming in either through the 3.5 mm jack, or the Rin, Lin, and GND pins of the board. This can come from your laptop, your phone, an MP3 player, or any other audio source.

I chose to use the Old-Time Radio project in Python Radio 32 as the audio source since it was sitting there on my desk.

As I mentioned, that project had audio levels too high for the FM transmitter chip. So I added a 68k ohm resistor between the L pin of the source and the LIN pin of the FM transmitter. This cleaned up the audio immensely, and the receiver output sounded like a professional FM station.

The old-time radio source was mono, so only the left side is needed. If you want stereo, add a similar 68k ohm resistor between R and RIN.

Normally we would also connect the ground pins, but since both computers use the same USB hub, the grounds were already connected.

Receiving RDS data

The screenshot at the top of this story shows the FM transmitter waveform in the SDR# program. That program also decodes the RDS data for you. I called my station Birdfarm (since I live on a 20-acre parrot farm in the mountains above Silicon Valley), and I sent a string of text as well. You can see that at the top of the screenshot.

The 40A7 between the two strings is the station ID number. I left it as the chip default, as I do not have a registered ID number.

The chip accepts the RDS strings four bytes at a time. To send a longer string, we need to loop through the string, sending four-byte chunks.

At the top of the main.py file is a comment leading to the 320-page document describing how to program the chip. That's where you can find the meaning of all the cryptic hexadecimal numbers in the driver code.

The end result of all of this is a very clean FM signal that goes surprisingly far. The sound quality is excellent. If your neighbors are far enough away not to be bothered, you can extend the range by adding a better antenna. Anything over a meter long can get you in trouble with the FCC, but you are unlikely to see any complaints, especially if you choose a quiet portion of the dial.

Python

Radio

Radio Hackers

Fm Transmitter

Micropython

Python Radio 25: UDP networking

Simon Quellen Field

Simon Quellen Field

Follow

7 min read

.

Sep 16, 2024

Listen

Share

More

One Datagram at a time

Press enter or click to view image in full size

Photo by the author

On a recent hike, a friend of mine surprised me. She is not someone you would expect to be interested in either radio or Python programming, but she had casually asked me what I had been up to lately, and I described one of the Morse code projects in this series.

She thought having Morse code radios to call her son to dinner would be fun, as he frequently ignored her texts. His room is in a separate building about a hundred feet from the kitchen.

Any of the little transceiver modules I have written about in this series would work fine. Still, I took this as an opportunity to explore an aspect of networking I had not used much before.

Connectionless data packets using UDP over Wi-Fi.

Most Wi-Fi communication involves setting up a connection between two computers. When you read your email or visit a web page, the two computers set up a protocol using TCP/IP. This allows them to resend data packets if they get lost or damaged, and involves two-directional communication where the receiver responds to let the sender know the data was received properly.

With UDP, packets are just sent. It is simple and quick, and if a packet is lost or corrupted along the way, the assumption is that this is tolerable.

In the Morse code radio I will be describing here, several things combine to make such unreliable communication work just fine. The first one is that there is a person at each end who can ask the other to repeat anything that didn't come through properly. This is quite common in CW radio sent by hand with Morse code keys.

Another thing in our favour here is that switches bounce.

When a person presses a switch or a button, there is a very short period when the contacts touch and then bounce back open, often several times. This is usually a problem, and code is written to wait until the bounces settle down before testing whether the switch is actually open or closed.

In our application here, we ignore that. Every time we notice the switch has made contact, we send a data packet telling the other radio to beep. We might send a dozen or more packets in a few milliseconds. At the other end, any tiny interruption in the sound only happens in the first millisecond or two and is scarcely noticeable.

When the switch is opened, there is also some bouncing. We thus have redundancy. We send several packets for ON, and several for OFF, and if a few are missed, it is no big deal.

There is a chance that we will send an ON signal and the OFF signals will be missed. The receiver would keep making noise. But our code makes noise for the sender as she presses the key, so if her receiver is stuck, she can just tap the key to stop it. The son would hear a quick beep at his end, but that would not be interpreted as a Morse message.

So now I needed to build two transceivers. I have a number of nice (but expensive) Morse code keys:

Press enter or click to view image in full size

Photo by the author

For this project, however, I wanted something as cheap as I am. I found some six-dollar keys on Amazon.com made from strips of metal on a plastic base that fit the bill nicely:

Press enter or click to view image in full size

Photo by the author

The little Wemos D1 Mini is the perfect computer for this project. I get them ten at a time from Aliexpress.com for less than two dollars each. They can send reliably over Wi-Fi for over 350 feet, which is more than enough range for the dinner bell.

Lastly, we need a cheap little speaker. I have a number of these little 2-inch speakers around, and they are perfect, and probably cost me fifty cents each a few years back. AliExpress has some like it for eighty-eight cents each. Since we will be flashing the LED on the D1 Mini as well as making beeps, even the speaker is optional if you are cash-strapped.

There are only four places where we need to solder something. The wires from the key connect to pins D8 and D7 on the D1 Mini. The speaker wires connect to D3 and D7. Rather than try to solder two wires to D7, I connected the key to the speaker terminal that was connected to D7, since the solder pad there was bigger.

The result looks like this:

Press enter or click to view image in full size

Photo by the author

All that's left is to write the code. Here it is:

From name import name, receiver, address

From network import WLAN, STA_IF, AP_IF, AUTH_OPEN

Try:

Import usocket as socket

Except:

Import socket

From machine import Pin, PWM

Import network

Ip = 0

Dgram_socket = 0

SEND_ON = "on"

SEND_OFF = "off"

UDP_PORT = 5001

LED = Pin(2, Pin.OUT)

LED.value(1) # Start with the LED off

D3 = Pin(0, Pin.OUT)

D7 = Pin(13, Pin.OUT)

D8 = Pin(15, Pin.IN, Pin.PULL_UP)

ANY_CHANGE = const(3)

KEY = D8

SPEAKER = D3

Pwm = PWM(SPEAKER, freq=800, duty=1023)

D7.value(1)

Def key_changed(pin):

Global LED, pwm, ip, dgram_socket

If pin.value():

LED.value(0)

Pwm.duty(512)

If ip and dgram_socket:

Dgram_socket.sendto(SEND_ON, (ip, UDP_PORT))

Else:

LED.value(1)

Pwm.duty(1023)

If ip and dgram_socket:

Dgram_socket.sendto(SEND_OFF, (ip, UDP_PORT))


```

KEY.irq(trigger=ANY_CHANGE, handler=key_changed)
Def say_OK():
From time import sleep
For x in range(6):
Pwm.freq(500 + x * 100)
Pwm.duty(512)
Sleep(.1)
Pwm.duty(1023)
Sleep(.1)
Pwm.freq(800)
Def connect():
From time import sleep
Global name, ip, address
Ssid = ""
Station = WLAN(STA_IF)
Station.active(True)
Try:
Network_list = station.scan()
Except Exception as e:
Station.disconnect()
Station.active(False)
Station = WLAN(STA_IF)
Station.active(True)
Network_list = station.scan()
Print("Available networks:")
For net in network_list:
Ssid_name = net[0].decode("utf-8")
Rssi = net[3]
If ssid_name.find(receiver, 0, len(receiver)) == 0:
Print(" -> ", ssid_name, rssi)
Ssid = ssid_name
Else:
Print("   ", ssid_name, rssi)
Ap = WLAN(AP_IF)
Ap.ifconfig((address, "255.255.255.0", "10.90.20.1", "1.1.1.1"))
Ap.active(True)
Ap.config(essid=name, authmode=AUTH_OPEN)
While station.isconnected() == False:
Print("Connecting to: ", ssid)
Station.connect(ssid, "")
For x in range(50):
Sleep(0.1)
Print()
Ip = station.ifconfig()[0]
Print(ip)
Ap.config(essid=name, authmode=AUTH_OPEN)
Print("Connected to", ssid)
Say_OK()
Def main():
Global dgram_socket

```

```

Connect()
Dgram_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
Dgram_socket.bind(("", UDP_PORT))
While True:
Packet = dgram_socket.recv(128)
If packet == b"":
Connect()
Dgram_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
Dgram_socket.bind(("", UDP_PORT))
Else:
Msg = packet.decode("utf-8")
If msg == "on":
LED.value(0)
Pwm.duty(512)
Else:
LED.value(1)
Pwm.duty(0)
Main()

```

Let's walk through it.

The little blue LED on the D1 Mini is connected to pin 2.

We set up D3 and D7 as outputs, and D8 as an input for the key. We set D7 to 1 so that when the key is pressed, D8 will also read as 1.

The `key_changed()` function is called whenever D8 changes state. If D8 reads as a 1, we turn on the LED (by sending a 0 to pin 2) and we beep by setting the duty cycle on the PWM to 512 (to generate a square wave).

Lastly, the function sends a packet with either "on" or "off" as the data.

It takes each radio a second or three to boot up and find the other radio's Wi-Fi SSID. When it is ready, it emits six quick beeps each 100 Hertz higher than the last. This lets the user know that communication can now take place, as both ends are ready.

The `connect()` function scans the Wi-Fi SSIDs it can see, looking for the SSID of its partner. It connects to that as a "station". It then sets up its own SSID and sets up as an access point. Each radio connects to the other radio's access point using its "station" connection. The IP address of the other radio is stored in the variable "ip" to be used when sending packets in the `key_changed()` function.

The `main()` function calls `connect()` and then creates a socket for the UDP datagrams. It listens to port 5001 (`UDP_PORT`) for packets sent to that port by the other radio. When it sees a packet, it turns on or off the LED, and makes the speaker beep or be quiet.

This project also works as a remote doorbell, or as a notifier when the postman opens and closes the mailbox. It can alert you when someone steps on a doormat, opens a refrigerator, or when the bird feeder is empty.

You can also combine this with the Morse text-to-code program to send messages to either radio from your laptop, phone, or desktop computer.

In addition to the `main.py` program shown above, there is a `name.py` module for each radio. The one for Iris looks like this:

```

Name = "Iris"
Receiver = "Ryan"
Address = "10.10.10.10"

```

The one for her son Ryan looks like this:

```

Name = "Ryan"
Receiver = "Iris"

```

Address = "10.10.10.10"

Having the names in the name.py file allows the main.py file to be the same for both radios. This was very convenient during the programming and debugging phases, as I could load main.py onto each machine after making changes, and not have to have a different main.py for each radio.

Radio Hackers

Python

Udp

Python Radio 32: Old-Time Radio

Build a Time Machine

Simon Quellen Field

Simon Quellen Field

Follow

6 min read

.

Jan 5, 2025

Listen

Share

More

Press enter or click to view image in full size

Old-Time Radio Time Machine

MidJourney

In the days before television, electronic entertainment meant radio.

Not just music and news, but comedy shows, mysteries, and adventure came streaming into homes through receivers that were often furniture. Fancy polished wood cabinets were centerpieces in the living room, with chairs arranged around them so the family could listen together.

Those were the days of Abbott and Costello, Jack Benny, Burns and Allen, Amos and Andy, and Fibber McGee and Molly. There was the CBS Radio Mystery Theater. There were dramas like Our Miss Brooks and Sherlock Holmes.

Thousands of these old radio shows are now collected on the Internet to be downloaded for free, as they are now out of copyright. Orson Welles' War of the Worlds caused a panic when people believed it was a true news program. The Lone Ranger brought the old West into living rooms.

Many of the older shows from the 1930s are poorly recorded, but by the 1940s the technology had improved greatly, and the sound quality is quite good.

My Receiver

I have a replica of an old-time radio. It has a nice polished wood cabinet and sits on the counter in the living room. Under it is a drawer where I have hidden an A.M. radio transmitter playing a mix of those old-time radio shows.

Visitors find this quite amusing.

My Transmitter

In keeping with the theme of this series, the transmitter is an ESP32 Lolin S2 Mini connected to an SD card reader and a PCM5102 stereo Digital to Analog Converter (DAC) board.

The ESP32 sends out a 3.3-volt 625-kilohertz square wave that acts as the carrier. A diode that amplitude modulates the carrier mixes it with the audio from the DAC.

Press enter or click to view image in full size

The Transmitter.

Photo by the author

The connections for the SD Card Reader are as follows:

ESP32 Pin 5 → SD Card Reader CS

ESP32 Pin 18 → SD Card Reader SCK

ESP32 Pin 12 → SD Card Reader MOSI

ESP32 Pin 11 → SD Card Reader MISO

ESP32 Pin 3.3V → SD Card Reader VCC

ESP32 Pin GND → SD Card Reader GND

[Press enter or click to view image in full size](#)

The SD Card Reader

Photo by the author

The connections for the PCM51002 DAC are:

ESP32 Pin 40 → DAC BCK

ESP32 Pin 38 → DAC DIN

ESP32 Pin 36 → DAC LRCK

ESP32 Pin 3.3V → DAC VIN

ESP32 Pin GND → DAC GND

[Press enter or click to view image in full size](#)

Photo by the author

Since the ESP32 only brings out one pin for 3.3 volts, I made a Y connector to power both of the other boards.

The PWM output for the carrier wave comes out on pin 3. This goes to a 1k ohm resistor. The audio output from the DAC comes out from the left output pin (marked L on one side of the board and LROUT on the other). This goes to another 1k ohm resistor.

The two resistors connect to one end of the diode (any silicon diode will work).

[Press enter or click to view image in full size](#)

The A.M. modulator

Photo by the author

The remaining end of the diode connects to a loop of wire whose other end connects to the analog ground pin on the DAC. This loop of wire is our antenna. You can make it as long as you like.

How the Hardware Works

We have two signal sources — the 625-kilohertz carrier and the audio signal, which varies between perhaps 100 hertz up to at most about 20 kilohertz.

To get an amplitude-modulated signal from these, we need to multiply them together. You might think we would use the computer to do that, but instead, we use a five-cent diode.

How can a diode do multiplication?

When we apply a voltage to a diode, the current going through the diode rises exponentially with the applied voltage. The graph looks like this:

[Press enter or click to view image in full size](#)

Voltage/current graph for a diode

Mmerevise.co.uk

When we add exponents, we get multiplication. We put our two signals in at one end, and their voltages add in the diode. The result is the product of the two. Since the signals are made up of waves, they interfere, and we get our amplitude-modulated signal out.

What, No Filter?

Our carrier wave is a square wave. It consists of the fundamental sine wave, plus all of the odd harmonics of that wave, in decreasing amplitude as they increase.

This means we could hear our old-time radio at 1875 kilohertz, but only one-third as far. We could hear the signal at 3125 kilohertz, but only 1/5th as far.

We could build a low-pass filter to remove these harmonics. But our signal is 3.3 volts, at about 10 milliamperes, into an antenna that is ten inches long. The wavelength at 625 kilohertz is over 1,500 feet long so that antenna is far from resonant.

How far does our signal go?

The best I've seen is about a foot, after careful fiddling with the position of the antenna and the receiver.

The receiver picks up a beautifully clear signal when it is a few inches above the hidden transmitter. That signal and clarity would be the envy of any 1940s radio listener. But move the receiver a few inches, and the signal degrades quickly.

The Software

Our SD card reader can't handle the 64-gigabyte microSD cards I first tried. It could probably handle 32-gigabyte cards, but I found a one-gigabyte card in a drawer that I'm sure no one will miss. It will hold over 300 radio shows of about ten minutes each. That's two days of constant programming.

We will play WAV files since micropython has modules to do that. Many of the radio programs are in MP3 format, so we convert them to WAV before storing them on the microSD card.

Because the language has support for both boards, the program is actually fairly short:

```
From machine import I2S, SPI, Pin, PWM
From micropython import schedule
From os import mount, umount
Spi = SPI(2, baudrate=100000, polarity=0, phase=0, sck=18, mosi=12, miso=11)
From sdcard import SDCard
Sdcard = SDCard(spi, cs=Pin(5))
Mount(sdcard, "/sd")
Pin = Pin(3, Pin.OUT)
Pwm = PWM(pin, freq=10, duty=512)
SCK_PIN = 40
SD_PIN = 38
WS_PIN = 36
I2S_ID = 0
BUFFER_LENGTH_IN_BYTES = 40000
Wav = open("/sd/{0}.format("WhosOnFirstAudio.Wav"), "rb")
Audio_out = I2S(I2S_ID,
Sck=Pin(SCK_PIN), ws=Pin(WS_PIN), sd=Pin(SD_PIN),
Mode=I2S.TX, bits=16, format=I2S.MONO, rate=16000, ibuf=40000)
Silence = bytearray(1000)
Wav_samples = bytearray(10000)
Wav_samples_mv = memoryview(wav_samples)
Def i2s_callback(arg):
Num_read = wav.readinto(wav_samples_mv)
If num_read == 0:
Pos = wav.seek(44)
Audio_out.write(silence)
Else:
Audio_out.write(wav_samples_mv[:num_read])
Def freqs():
Print()
Print("Convenient reachable frequencies in the A.M. band:")
Guess = 0
For f in range(540, 1700):
Try:
Pwm.freq(f * 1000)
Actual = pwm.freq()
If actual != guess and actual % 1000 == 0:
Print(str(actual / 1000) + " kHz")
Guess = actual
```

Except ValueError as verr:

Pass

Print()

Def main():

Pwm.freq(625 * 1000)

Audio_out.irq(i2s_callback)

Wav.seek(44) # advance to first byte of Data section in WAV file

Audio_out.write(silence)

Freqs()

Main()

It would be even shorter, but I threw in a little routine to calculate convenient frequencies in the A.M. band that the ESP32 can reach using its PWM module.

The SD card reader talks to the ESP32 over the SPI bus. We mount it as a file system called /sd.

The PCM51002 DAC talks to the ESP32 over I2S, a streaming protocol we have not used before in this series. We read samples from the WAV file and stream them to the DAC using I2S. The hardware can thus do all of the heavy lifting, and our program is simple.

We start out by sending a buffer full of zeros for a moment of silence. This “primes the pump”, and we get an interrupt when the I2S module wants more data. That’s when we send the WAV bytes.

At the end of the file, we send more silence, rewind to the beginning, and start over.

In this program, I just play Abbott and Costello’s Who’s On First comedy routine over and over. But you can concatenate days’ worth of old-time radio shows into one big WAV file if you like. Or you can read the files off the microSD card individually.

Programming

Python

Radio

RF Survival Guide - Part 1

Basic Antenna Concepts

Follow

6 min read

.

Mar 12, 2025

Listen

Share

More

Working as a Radio-Frequency (RF) Engineer, I know first hand that the world of RF can seem like a maze of jargon and complex concepts. Whether you’re an amateur radio enthusiast or just someone curious about how wireless communication works, understanding some of the basic terminology can feel like a daunting task. In this guide, I’ll break down some of the most common RF concepts and explain them in plain language, making it easier to navigate this crazy and complex world of electromagnetic waves and to ease the understanding of much of the RF content posted here on Medium.

Knowledge Disclaimer: Although this article is directed to RF newbies, I assume the reader is well versed in basic electrical concepts such as impedance, resistance and reactance.

Press enter or click to view image in full size

Photo by Gontran Isnard on Unsplash

Resonant Frequency

Frequency is the number of times a wave oscillates per second. It is measured in Hertz (Hz). Radio frequencies officially go up to 3THz, but, truly, this is a kind of arbitrary value defined by the Institute of Electrical and Electronics Engineers (IEEE) as terahertz wireless communications are currently in the early research and development stage rather than widespread commercial deployment. Frequencies higher than 3THz are considered to be in the infrared part of the electromagnetic

spectrum and beyond. Keep in mind that these hard limits are just human-created categories. The resonant frequency of an antenna is the frequency at which the antenna naturally oscillates (electrically speaking, of course) and efficiently radiates or receives electromagnetic waves. At this frequency, the antenna has minimal reactance (imaginary impedance is close to zero and only real impedance remains), meaning it is well-matched to the system and minimizes power loss (at this frequency, the antenna does not store any energy due to minimal reactance and radiates all the energy it is fed).

Antenna Bandwidth

Antenna Bandwidth, not to be confused with network bandwidth which measures the capacity of a communication channel (and is the most common colloquial use of the word), is the range of frequencies within which an antenna works well. That is, the frequency range over which the antenna is considered to be impedance matched to the feed line, the gain is within expected values and the radiation pattern is not distorted. When operating outside the bandwidth it was designed to, an antenna can work differently than expected. For example, directional antennas might start to behave more omnidirectionally, with less gain and start to reflect power back to the transmitter due to poor impedance matching, with risk of damaging the device.

Bell (B) and Decibel (dB)

The Bell (B) is not a unit. It is the base 10 logarithm of a ratio between two values (typically a measured value and a reference value). It is rarely used in practice because it is too large for most applications. A Decibel (dB) is one-tenth of a Bell and much more commonly used. Decibels are widely used in the RF context because many measured values span several orders of magnitude. The decibel scale compresses large numbers into a manageable range and simplifies calculations (because in dB, multiplication becomes addition, making it easier to compute gains and losses in cascaded systems).

Power Values (dBm)

As I explained before, in the RF world, power values can span several orders of magnitude. For example, radiated power in a certain direction decays quadratically with distance (i.e. it changes a lot even for relatively small distance variations). As a result, the use of decibels is recommended to compress results into a manageable range. This is also true for many other antenna-related quantities.

dBm is a unit of power relative to 1 milliwatt. So it is ten times the logarithm of the ratio between a certain value and 1 milliwatt. In other words, dBm refers to milliwatts in decibel scale.

Radiation Efficiency and Total Efficiency

Radiation efficiency is the ratio of the power radiated by an antenna to the total input power supplied to it, considering only conductor and dielectric losses. Total efficiency includes both radiation efficiency and mismatch losses due to impedance mismatches, such as between the antenna and the feed line. Much more could be said about this topic (And any of the others, really), including explaining radiation resistance. But I don't want to make this a dense article. I would say this is enough for any beginner regarding antenna efficiency. Many times, when RF engineers talk about efficiency they do not specify which one, contributing to the terminology mess we have in the field.

Directivity, Gain and Realized Gain

Directivity measures how well an antenna focuses energy in a specific direction compared to an isotropic radiator (an imaginary antenna that equally radiates in all directions). It is a theoretical property and does not consider losses.

Gain represents the effective radiated power in a given direction. It includes radiation efficiency. Realized Gain includes both radiation efficiency and impedance matching losses. It is the most practical of the three.

Many times, when RF engineers say "gain" without specifying a direction we typically mean maximum gain (in the strongest radiation direction). We also typically mean realized maximum gain, simply

because that's the one we actually can measure (gain and directivity are only given by electromagnetic simulation software). This just adds up to the maze of terminology jargon I talked about in the beginning.

The terms covered in this section are useful to characterize how well an antenna focuses energy in a specific direction. Some we want to radiate everywhere, others quite the opposite.

Gain Values (dBi, dBd and dBc)

Gain values are expressed in different reference units: dBi, dBd, and dBc, each comparing antenna performance to a different standard.

dBi (decibels relative to an isotropic antenna) is the most common and compares gain to a theoretical isotropic radiator, making it useful for general antenna characterization.

dBd (decibels relative to a dipole) compares gain to a half-wave dipole antenna.

dBc is used for circularly polarized antennas, referenced to an ideal isotropic circularly polarized radiator.

An antenna with a gain of 12 dBi means it radiates 12 dB more power in its main direction compared to an ideal isotropic antenna that radiates equally in all directions.

Polarization and Polarization Loss Factor

Polarization refers to the electric field's orientation in the electromagnetic waves that are transmitted by the antenna. It is defined for a given direction, however, when the direction is not stated, the polarization is taken to be the polarization in the direction of maximum gain. The orientation of the electric field can be constant or change over time. Orienting an antenna made of linear conductors (linear antenna) vertically will result in vertical polarization, while orienting the same antenna horizontally will result in horizontal polarization. For non-linear antennas or for arrays of linear antennas that are not aligned, the orientation of the electric field changes as the wave travels through space. If the electric field vector movement in time describes an ellipse or circle, polarization is considered elliptical and circular, respectively. Antenna polarization is very important, because when the receiving antenna's polarization matches the one of the transmitted wave, there is optimum reception. Otherwise, especially for linear antennas that are orthogonally oriented, reception will be poor. For circularly polarized antennas, signals with either vertical or horizontal polarization can be received, at the cost of a 3 dB reduction in signal strength (and vice-versa). The Polarization Loss Factor (PLF) represents the amount of energy lost due to the mismatch between the polarization vector of a receiving antenna and the polarization vector of an incident wave.

That should be enough for a beginner to explore for days! I could really see a series coming out of this article, especially when we have so much RF-related stuff to talk about, such as Signal-to-Noise Ratio, modulations, S-Parameters, etc... Let me know of your interest in the comments

Radio Frequency

Antennas Installations

Electronics

RF Survival Guide — Part 2

S-Parameters Demystified

Follow

5 min read

.

Apr 16, 2025

Listen

Share

More

Press enter or click to view image in full size

Generic input reflection coefficient measurements for LoRa antennas.

Welcome back, RF hobbyists!

In Part 1 of the RF Survival Guide, we explored the chaotic beauty of RF jargon. Today, we dive into a deceptively simple yet powerful tool in any RF engineer's toolbox: scattering parameters. If you've ever used a VNA (Vector Network Analyzer) spitting out a bunch of curves labeled S11, S21, and thought, "this must be sorcery," you're not alone! But fear not — by the end of this guide, you'll see that S-parameters are just a clever way of describing how RF components behave. What Are S-Parameters?

S-parameters, or scattering parameters, tell you how signals scatter when they hit a device. In plain terms:

How much of each input signal gets reflected back?

How much of each input signal gets transmitted through to each output?

For a 2-port device (like an amplifier, filter, or cable), you typically see:

S11: Input reflection coefficient at port 1 (input return loss);

S21: Transmission coefficient from port 1 to port 2 (forward gain/forward loss/isolation)

S12 : Transmission coefficient from port 2 to port 1 (reverse gain/reverse loss/reverse isolation)

S22: Output reflection coefficient at port 2 (output return loss)

Using S-Parameters to Check Antenna Matching

Let's say you just designed a tiny antenna for your latest IoT project. You hook it up to a Vector Network Analyzer (VNA), and it gives you a plot of S11 across frequencies. So, how do you know if the antenna is matched?

S11 represents how much of the power sent into port 1 is reflected back. So:

If $|S_{11}|$ is close to 0 (or in dB, -10 dB or lower), that means very little is being reflected, which is what we want. It means the antenna is well matched to the system impedance (typically 50 ohms) at that frequency.

If $|S_{11}|$ is close to 1 (or 0 dB), it means almost all the power is bouncing back. Not great!

But where does the power go if it isn't reflected?

In a simple case (like an antenna directly connected to the VNA), any power that isn't reflected is either radiated into space or lost as heat. Since antennas are meant to radiate, a low S11 typically means most power is radiating effectively. Also, if your antenna goes through a matching network, part of that power might be lost in resistive components or reflected back by mismatched stages. S11 doesn't tell you where the power goes — only how much is reflected. The rest could be radiated, lost, or absorbed. That's why we combine S-parameter analysis with efficiency or gain measurements when testing antennas. When we measure the realized gain of an antenna, we can actually know how much of that unreflected power is actually radiated.

A Few Examples

In the images shown below, you can see the difference in S11 between a narrow-band Printed Inverted-F Antenna, tuned for 868MHz, with around 50MHz bandwidth and a sharp resonance:

And a very wideband log-periodic antenna that can work well in the complete frequency range that is shown, having at least 500MHz bandwidth:

Using S-Parameters to Get Cable Loss

Whether you're working with short interconnects inside a device or long coaxial runs in automotive or aerospace systems, knowing how much signal you're losing is crucial. Fortunately, S-parameters can give us that information directly — no need for guesswork or rule-of-thumb estimations.

To measure cable loss, you typically look at S21, the forward transmission coefficient. In an ideal, lossless cable, S21 would be 0 dB across your frequency of interest, meaning all your power gets through. But, real cables introduce attenuation, and that shows up as a drop in the S21 magnitude. The difference from 0 dB gives you the insertion loss introduced by the cable.

You can also use S11 to check for mismatches or reflections. A well-matched cable should have an S11 close to -20 dB or lower. High reflection could indicate poor connectors, improper termination, or even physical damage to the cable.

One Example

Looking at the S21 of this random cable, you can see it has an insertion loss of around 0.3–0.5 dB:

Using S-Parameters to Analyze Amplifiers

Amplifiers are one of the most common — and most critical — components in RF systems. Whether you're boosting a signal before transmission or cleaning up a weak one after reception, you want to be sure your amplifier is doing what it's supposed to do. S-parameters can help you check some of the most important amplifier parameters:

S21 (Gain): This is the star of the show. The magnitude of S21 (in dB) tells you how much gain your amplifier provides across the frequency range. A flat and high S21 is usually what you're after.

S11 (Input Match): This shows how well the amplifier input is matched to the system impedance, typically 50 ohms. Poor input match leads to reflections, which can reduce system efficiency or even cause instability.

S22 (Output Match): Same concept, but at the output. If the amplifier is going to drive another stage or a transmission line, you want minimal reflection here too.

S12 (Reverse Isolation): In a perfect world, an amplifier should be unilateral, meaning no signal should leak from output back to input. S12 gives you an idea of how much reverse signal is getting through. Low S12 (-30 dB) is ideal and helps prevent oscillations in cascaded systems.

When working with amplifiers, one should look at stability circles, gain compression, intermodulation and noise figure, but S-parameters alone are not enough to check all of these parameters. They help me flag issues early, long before building or testing anything physical. If you're designing LNA stages or evaluating power amplifiers in a transmission chain, starting with S-parameters gives you a solid foundation. For more details about RF amplifiers, make sure you stick around for future parts of this RF survival guide!

Rf

Engineering

Electronics

RF Survival Guide — Part 3

PCB Transmission Lines

Follow

3 min read

.

Aug 1, 2025

Listen

Share

More

Unless your antenna is very, very close to your RF feed source, you will always need a transmission line to carry high-frequency signals from one place to another. Outside PCB boards, you can use already-made cables and waveguides, which are specifically designed for that purpose. But if working with PCBs, you need to design these transmission lines yourself!

This article covers some basic types of PCB transmission lines and the pros and cons of each, to help you decide on which to use for your next RF PCB project!

Coplanar Waveguides

Press enter or click to view image in full size

CPWG Example -> H: Substrate Thickness; ϵ_r : Relative Permittivity; G: Trace-GND Gap; W: Trace width; T: Copper Thickness

Coplanar waveguides are so common that they can even be implemented at an IC level!

A Coplanar Waveguide (CPW) is a type of transmission line where the signal conductor and the return paths (ground planes) are all printed on the same side of the PCB. The signal trace runs in the middle, flanked by two ground planes with a consistent gap. Beneath the substrate, an optional ground plane can be added to improve shielding and lower loss, forming what's often referred to as a coplanar waveguide with ground (CPWG).

The characteristic impedance is determined by the trace width and thickness, substrate height and dielectric constant and the gap to the ground planes.

Stripline

[Press enter or click to view image in full size](#)

Very simple sketch I made, describing a stripline implementation on a PCB.

A stripline consists of a conductor sandwiched between two ground planes. The characteristic impedance is determined by the strip width, the substrate thickness, and the substrate's relative permittivity.

Like in many other PCB transmission lines, via stitching can be used to short the two ground planes along the edges of the planes, avoiding propagation of unwanted modes. Striplines are more expensive than other transmission lines, as they require 3-layer PCBs, but they provide better isolation due to the conductor being encapsulated inside metallic planes.

Microstrip

[Press enter or click to view image in full size](#)

Simple visualization of a microstrip transmission line.

A microstrip does not necessarily need to be implemented on a PCB, but it most commonly is. It is a type of transmission line where a conductor is placed on top of a ground plane and separated by a dielectric. By carefully designing the thickness of the dielectric and the conductor, the width of the conductor and the dielectric constant of the dielectric, one can tune the transmission line to have the intended characteristic impedance.

Micro strips only require 2-layer PCBs, making them more compact, lighter and cheaper than stripline technology. However, due to not being enclosed, microstrips are more susceptible to crosstalk and can easily radiate if they are close in length to the guided wavelength.

Designing a PCB Transmission Line

[Press enter or click to view image in full size](#)

All tx line types available at PCB Way's online calculator.

While professional tools like Keysight ADS or Ansys HFSS offer advanced simulation and tuning capabilities, most hobbyist and low-complexity projects can rely on online calculators or tools provided by PCB manufacturers, such as [from PCBWay](#).

Rf

Pcb Design

Pcb