Collected RF Reader

Explorations in Radio Frequency

Core & Basics

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

_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)

_ENV_SENSE_UUID = bluetooth.UUID(0x181A)

_TEMP_CHAR = (

Bluetooth.UUID(0x2A6E),

_FLAG_READ | _FLAG_NOTIFY | _FLAG_INDICATE,

_ENV_SENSE_SERVICE = (

_ENV_SENSE_UUID,

(_TEMP_CHAR,),

_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_THERMOMETER

Self._advertise()

Def _irq(self, event, data):

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)

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):

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:

Self._ble.gatts_notify(conn_handle, self._handle)

If indicate:

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(”)

Import uasyncio as asyncio

Import aioble

Import bluetooth

Import random

Import struct

_ENV_SENSE_UUID = bluetooth.UUID(0x181A)

_ENV_SENSE_TEMP_UUID = bluetooth.UUID(0x2A6E)

_ADV_APPEARANCE_GENERIC_THERMOMETER = const(768)

_ADV_INTERVAL_MS = 250_000

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)

Def _encode_temperature(temp_deg_c):

Return struct.pack(”<h”, int(temp_deg_c * 100))

T = 24.5

While True:

Temp_characteristic.write(_encode_temperature(t))

T += random.uniform(-0.5, 0.5)

Await asyncio.sleep_ms(1000)

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()

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:

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:

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:

TIMEZONE = -8 * 60 * 60

Async with scan(duration_ms=5000, interval_us=30000, window_us=30000, active=True) as 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”

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()
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 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

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.

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.

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.

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

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 03 E1   ~...............

02 S00 B2: Len 16 7E 01 03 E1 03 E1 03 E1 03 E1 03 E1 03 E1 03 E1   ~...............

03 S00 B3: Len 16 00 00 00 00 00 00 78 77 88 FF 00 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 00 com.............

06 S01 B2: Len 16 00 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 7F 07 88 FF 00 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 00 on.com..........

10 S02 B2: Len 16 00 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 7F 07 88 FF 00 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 00 ....

15 S03 B3: Len 16 00 00 00 00 00 00 7F 07 88 FF 00 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 00 .org............

18 S04 B2: Len 16 00 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 7F 07 88 FF 00 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 7F 07 88 FF 00 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 00 ss.............

27 S06 B3: Len 16 00 00 00 00 00 00 7F 07 88 FF 00 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 00 ....

29 S07 B1: Len 16 00 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 00 ....

31 S07 B3: Len 16 00 00 00 00 00 00 7F 07 88 FF 00 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 00 ....

33 S08 B1: Len 16 00 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 00 ....

35 S08 B3: Len 16 00 00 00 00 00 00 7F 07 88 FF 00 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 00 ....

37 S09 B1: Len 16 00 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 00 ....

39 S09 B3: Len 16 00 00 00 00 00 00 7F 07 88 FF 00 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 00 ....

41 S10 B1: Len 16 00 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 00 ....

43 S10 B3: Len 16 00 00 00 00 00 00 7F 07 88 FF 00 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 00 ....

45 S11 B1: Len 16 00 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 00 ....

47 S11 B3: Len 16 00 00 00 00 00 00 7F 07 88 FF 00 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 00 ....

49 S12 B1: Len 16 00 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 00 ....

51 S12 B3: Len 16 00 00 00 00 00 00 7F 07 88 FF 00 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 00 ....

53 S13 B1: Len 16 00 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 00 ....

55 S13 B3: Len 16 00 00 00 00 00 00 7F 07 88 FF 00 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 00 ....

57 S14 B1: Len 16 00 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 00 ....

59 S14 B3: Len 16 00 00 00 00 00 00 7F 07 88 FF 00 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 00 ....

61 S15 B1: Len 16 00 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 00 ....

63 S15 B3: Len 16 00 00 00 00 00 00 7F 07 88 FF 00 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:

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.

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):

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):
Return RfidAccess()
Def Process(self, cmd, data = None, sect = None, blk = None):
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")
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)
If cmd == READ:
Print("READ")
If self.reader.MFRC522_DumpClassic1K(uid, Start = 0, End = 4, keyA = firstSectorKey) == self.reader.OK:
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:
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”)

Access.decodeAccess(0xff, 0x07, 0x80)

Block3 = access.fillBlock3(keyA = defaultKey, keyB = defaultKey)

Self.reader.writeSectorBlock(uid, 0, 3, block3, keyB = defaultKey)

Datablock = 16 * [0]

Self.reader.writeSectorBlock(uid, 0, 1, datablock, keyB = defaultKey)

Self.reader.writeSectorBlock(uid, 0, 2, datablock, keyB = defaultKey)

For s in range(1, 16):

Self.reader.writeSectorBlock(uid, s, 3, block3, keyB = defaultKey)

For b in range(3):

Self.reader.writeSectorBlock(uid, s, b, datablock, keyB = defaultKey)

Print(“Erased”)

Return True

Elif cmd == FORMAT:

Print(“FORMAT”)

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.KEYB,

access_Inc = access.NEVER, access_Dec = access.NEVER)

block3 = access.fillBlock3(keyA = firstSectorKey, keyB = defaultKey)

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)

Self.reader.writeSectorBlock(uid, 0, 2, b1, keyB = defaultKey)

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.KEYAB,

access_Inc = access.KEYAB, access_Dec = access.KEYAB)

block3 = access.fillBlock3(keyA = nextSectorKey, keyB = defaultKey)

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 = “”)

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

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.

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

Photo by the author.

In the photo above, we connected the breakout board to a Wemos D1 Mini ESP8266 board. There are five 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 hardware goes together quickly. The software took a lot longer to create. But all you need to do is copy and paste.

Class xmit:

I2C_ADDRESS = 0x10

DEVICEID = 0x00 # Never changes

CHIPID = 0x01 # Never changes

POWERCFG = 0x02 # Control registers start here

CHANNEL = 0x03

SYSCONFIG1 = 0x04

SYSCONFIG2 = 0x05

SYSCONFIG3 = 0x06

OSCILLATOR = 0x07

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
SMUTE           = (1 << 15)
DMUTE           = (1 << 14)
MONO            = (1 << 13)
RDSM            = (1 << 11)
SEEK_MODE       = (1 << 10)
SEEKUP          = (1 << 9)
SEEK            = (1 << 8)
DISABLE         = (1 << 6)
SEEKDN          = (1 << 1)
ENABLE          = (1 << 0)
TUNE            = (1 << 15)
RDSIEN          = (1 << 15)
STCIEN          = (1 << 14)
ENABLE_RDS      = (1 << 12)
DE              = (1 << 11)
AGCD            = (1 << 10)
BLNDADJ         = (1 << 6)
GPIO3           = (1 << 4)
GPIO2           = (1 << 2)
GPIO1           = (1 << 0)
SEEKTH          = 8           # The top 8 bits
BAND            = 6           # The next 2 bits
SPACE1          = 5
SPACE0          = 4
VOLUME_MASK     = 0x000F      # The bottom 4 bits
SMUTER          = 14          # 2 bits
SMUTEA          = 12          # 2 bits
RDSPRF          = 9           # RDS performance bit
VOLEXT          = 8           # Extended volume range
SEEK_SNR        = 4           # 4 bits wide
SEEK_CNT        = 0           # 4 bits wide
AHIZEN          = (1 << 14)
XOSCEN          = (1 << 15)
```

```
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
BLERB         = (1 << 14)     # 2 bits wide
BLERC         = (1 << 12)     # 2 bits wide
BLERD         = (1 << 10)     # 2 bits wide
READCHANBITS    = (1 << 9)      # Low 9 bits
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):
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] &= 0xFFF0          # Clear volume bits
Self.regs[self.SYSCONFIG2] |= 1               # Lowest volume
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):
```

Type_str = group_types[gt]

Return type_str

Def get_program_type(self, pt):

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

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] &= 0xFFF0

Self.regs[self.SYSCONFIG2] |= v & 0xF

Self.write_all_registers()

Sleep(0.2)

Def set_frequency(self, freq):

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):
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)
I = 0
While i < len(r):
Ind = int(((i/2) + 10) % 16)
Self.regs[ind] = (r[i] << 8) | r[i+1]
I += 2
Def dump_registers(self):
Print(“DVID CHIP PWRC CHAN SYS1 SYS2 SYS3 OSC      RSSI RCHN RDSA RDSB RDSC RDSD”)
For x in range(16):
Print(“{:04X}”.format(self.regs[x]), end=” “)
Print()
Def histogram(self):
Histogram(self)
D1 = Pin(5)
D2 = Pin(4)
D3 = Pin(0)
Def show_RDS(x, how_many):
x.RDS_performance(True)
old_s = “”
sleep(0.04)                    # Records come in 11.4 times per second
if x.read_RDS():
r = x.rssi()
s = x.program_service
s += “ {:3.1f} – {: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 + “]”
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 += ‘                    \r’  # The <CR> prevents scrolling. Lines write over old lines.
If s != old_s:
Print(s, end=””)
Old_s = s
x.RDS_performance(False)
tune_demo(x, times, slp)
show_data(x)
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)
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 __init__() method starts the receiver chip receiving using some default settings.

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.

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

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)
```

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

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.

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.

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.

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.

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.

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.

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

Apt install gpredict

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.

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, you 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.

■ We’re now on Bluesky!

Articles we think you’ll like:

What The Tech?! Space Shuttles

Shodan: A Map of the Internet

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

Python Radio: Simple Beginnings

The simplest digital radio mode

medium.com

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.

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.

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 to 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.

Python

Micropython

Radio

From Concept to Antenna

Armando Rodrigues

Core & Basics

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

Why antenna design matters

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.

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

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

Yagi-Uda Antenna

Core & Basics

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)

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.

Parabolic Dish Antenna

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

Horn Antenna

Loop Antenna

Understanding how the antenna works

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.

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.

Conclusion

Antenna

Rf

Engineering

Software Defined Radio & Radio Hacking: Part 2

Investigator515

Investigator515

Follow

8 min read

Sep 25, 2023

Listen

Core & Basics

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 lessor skill in regards to cyber security but it is an important skill with regards to isolating signal interference and under standing 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 it's 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:

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.

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

Triangulation:

Press enter or click to view image in full size

Time Delay On Arrival (TDOA)

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.

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

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.

Challenges and Limitations:

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 at attenuate the signal to continue to close in on the location.

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.

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.

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

Support me on Ko-fi: Investigator515

Follow us on Twitter

Articles we think you'll like:

Software Defined Radio & Radio Hacking

What the Tech?! Personal Computers

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.

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.

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

The code looks like this:

Def main():

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()
```

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).

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.

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

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:

```
Def main():

PIN_D4 = 2

Tone = 800

Morse = Morse(PIN_D4, tone)

Print("Morse code AM Beacon")

Morse.speed(20)

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:

```
Def main():

PIN_D4 = 2

Tone = 300

Morse = Morse(PIN_D4, tone)

Print("Morse code AM Beacon")

Morse.speed(20)

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.

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.

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

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

Core & Basics

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.

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.

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

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

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 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

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.

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 pi). 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.

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

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 * 1000000000000000

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”)
```

```
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

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 more.

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:

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")

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 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

Import pylab as pyl

Import numpy as np

Import sys

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:

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:

Class RP_CW:

Def __init__(self, carrier_pin, freq):

@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_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):

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

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

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.

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.

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:

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.

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

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 from obtaining what you need?

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

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

Press enter or click to view image in full size

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 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.

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.

Press enter or click to view image in full size

More Resources

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.

OTW should be a famiiar 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 .

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

■ Support me on Ko-fi:

■ Follow us on

■ Articles we think you'll like:

What the Tech?! GPS

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

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:

Pwr = ADC(0)

While True:

Avg = 0

For count in range(20):

Avg += pwr.read()

Avg /= 20

Print(avg)

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.

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:

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()

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

We convert the average reading (which would be between 0 and 1023) into a number between 0 and 15 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

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

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.

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

The Cavity Magnetron

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

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.

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

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

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

The Future

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.

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

Core & Basics

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 was Edwin Howard Armstrong.

Removing Static

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.

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 a 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.

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.

Class xmit:

I2C_ADDRESS = 0x63

REGISTERS_ADDRESSES = (1, 257, 259, 513, 514, 8448, 8449, 8450, 8451, 8452, 8453, 8454, 8455, 8704, 8705, 8706,

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

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)

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)

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)

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)
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()
```

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?

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.

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 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.

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

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:

Try:

Import usocket as socket

Except:

Import socket

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():
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():
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.

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.

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.

Name = "Iris"

Receiver = "Ryan"

Address = "10.10.10.10"

Name = "Ryan"

Receiver = "Iris"

Address = "10.10.10.10"

Radio Hackers

Python

Udp

Python Radio 32: Old-Time Radio

Build a Time Machine

Core & Basics

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.

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

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

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 aurhor

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

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.

```
Spi = SPI(2, baudrate=100000, polarity=0, phase=0, sck=18, mosi=12, miso=11)

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

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.

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

Core & Basics

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)

Power Values (dBm)

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

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 dBic)

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

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

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

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

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:

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)

Using S-Parameters to Check Antenna Matching

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

If |S11| 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 |S11| 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

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

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.

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.

Rf

Engineering

Electronics

RF Survival Guide — Part 3

PCB Transmission Lines

Follow

3 min read

Aug 1, 2025

Listen

Share

More

Coplanar Waveguides

Press enter or click to view image in full size

CPWG Example -> H: Substrate Thickness; ε■: 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!

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

Simples 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

Higher Topics

Python Radio 29: Far Away BBS

Using $1 LoRa Modules to store and forward messages over miles

Simon Quellen Field

Simon Quellen Field

Follow

19 min read

Oct 13, 2024

Listen

Share

More

Press enter or click to view image in full size

Long-distance communication using tiny radios.

MidJourney

For a dollar, you can get a little transceiver that puts out 63 milliwatts of power and can communicate over a distance of ten miles. That's already 158 miles per watt. With the Yagi-Uda antenna we built in Python Radio 28, we get an additional factor of 16 in distance. That's 160 miles, or 2,539 miles per watt.

But we have Python at our disposal. Why stop at just a chat mode? Why not set up a bulletin board system to store messages and allow a whole group to participate?

We will use either the Ra-01 module or the RF96 module, which both use the SX127x transceiver chip.

Press enter or click to view image in full size

The Ra-01 transceiver module.

The Ra-01 transceiver module (photo by author)

Press enter or click to view image in full size

The RF96 transceiver module.

The RF96 transceiver module (photo by author)

The RF96 is a tiny little board. Two of them are shown in my hand in the photo below:

Press enter or click to view image in full size

My hand holding two RF96 transceiver modules.

Photo by author

When the Ra-01 radio is connected to the RP2040 on the breadboard, it looks like this:

Press enter or click to view image in full size

Ra-01 connected to RP2040 on a breadboard.

Photo by author

The RF96 looks like this:

Press enter or click to view image in full size

RF96 and RP2040 on a breadboard.

Photo by author

On the RP2040, we use SPI zero, which means pin 16 connects to RF96 MISO, pin 19 is MOSI, and pin 18 is SCK.

There are several variants of the RP2040 since the design is open source. I have a bunch of the purple boards from AliExpress.com, whose pinouts are shown below:

Press enter or click to view image in full size

RP2040 Purple Board Pinout.

The file config_lora.py:

Import sys

Import os

Import time

IS_PC = False

IS_MICROPYTHON = (sys.implementation.name == 'micropython')

IS_ESP8266 = (os.uname().sysname == 'esp8266')

IS_ESP32 = (os.uname().sysname == 'esp32')

IS_RP2040 = (os.uname().sysname == 'rp2')

IS_TTGO_LORA_OLED = None

IS_RPi = not (IS_MICROPYTHON or IS_PC)

Def mac2eui(mac):

Mac = mac[0:6] + 'fffe' + mac[6:]

Return hex(int(mac[0:2], 16) ^ 2)[2:] + mac[2:]

If IS_MICROPYTHON:

Import machine

Import ubinascii

Uuid = ubinascii.hexlify(machine.unique_id()).decode()

If IS_RP2040:

NODE_NAME = 'RP2040'

If IS_ESP8266:

NODE_NAME = 'ESP8266_'

If IS_ESP32:

NODE_NAME = 'ESP32_'

Import esp

IS_TTGO_LORA_OLED = (esp.flash_size() > 5000000)

NODE_EUI = mac2eui(uuid)

NODE_NAME = NODE_NAME + uuid

Millisecond = time.ticks_ms

SOFT_SPI = None

If IS_TTGO_LORA_OLED:

SOFT_SPI = True

Elif IS_RP2040:

Else:

If IS_RPi:

Import socket

NODE_NAME = ‘RPi_’ + socket.gethostname()

Millisecond = lambda : time.time() * 1000

If IS_PC:

Import socket

NODE_NAME = ‘PC_’ + socket.gethostname()

Millisecond = lambda : time.time() * 1000

The controller_rp2040.py is a simple port, copying most of the file from the other controller_xxx.py files and changing a few lines to fit the RP2040:

Import config_lora

Import controller

Class Controller(controller.Controller):

PIN_ID_FOR_LORA_RESET = 14

PIN_ID_FOR_LORA_SS = 15

PIN_ID_SCK = 18

PIN_ID_MOSI = 19

PIN_ID_MISO = 16

PIN_ID_FOR_LORA_DIO0 = 20

PIN_ID_FOR_LORA_DIO1 = None

PIN_ID_FOR_LORA_DIO2 = None

PIN_ID_FOR_LORA_DIO3 = None

PIN_ID_FOR_LORA_DIO4 = None

PIN_ID_FOR_LORA_DIO5 = None

If config_lora.IS_RP2040:

ON_BOARD_LED_PIN_NO = 25

ON_BOARD_LED_HIGH_IS_ON = True

GPIO_PINS = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9,

Def __init__(self,

Pin_id_led = ON_BOARD_LED_PIN_NO,

On_board_led_high_is_on = ON_BOARD_LED_HIGH_IS_ON,

Pin_id_reset = PIN_ID_FOR_LORA_RESET,

Blink_on_start = (2, 0.5, 0.5)):

Super().__init__(pin_id_led,

On_board_led_high_is_on,

Pin_id_reset,

Blink_on_start)

Def prepare_pin(self, pin_id, in_out = Pin.OUT):

If pin_id is not None:

Pin = Pin(pin_id, in_out)

New_pin = Controller.Mock()

New_pin.pin_id = pin_id

New_pin.value = pin.value

If in_out == Pin.OUT:

New_pin.low = lambda : pin.value(0)

New_pin.high = lambda : pin.value(1)

Else:

New_pin.irq = pin.irq

Return new_pin

Def prepare_irq_pin(self, pin_id):

Pin = self.prepare_pin(pin_id, Pin.IN)

If pin:

Pin.set_handler_for_irq_on_rising_edge = lambda handler: pin.irq(handler = handler, trigger = Pin.IRQ_RISING)

Pin.detach_irq = lambda : pin.irq(handler = None, trigger = 0)

Return pin

Def get_spi(self):

Spi = None

Id = 0

If config_lora.IS_RP2040:

Try:

Spi = SPI(id, baudrate = 10000000, polarity = 0, phase = 0, bits = 8, firstbit = SPI.MSB,

Sck = Pin(self.PIN_ID_SCK, Pin.OUT, Pin.PULL_DOWN),

Mosi = Pin(self.PIN_ID_MOSI, Pin.OUT, Pin.PULL_UP),

Miso = Pin(self.PIN_ID_MISO, Pin.IN, Pin.PULL_UP))

Spi.init()

Except Exception as e:

Print(e)

If spi:

Spi.deinit()

Spi = None

Reset() # in case SPI is already in use, need to reset.

Return spi

Def prepare_spi(self, spi):

If spi:

New_spi = Controller.Mock()

Def transfer(pin_ss, address, value = 0x00):

Response = bytearray(1)

Pin_ss.low()

Spi.write(bytes([address]))

Spi.write_readinto(bytes([value]), response)

Pin_ss.high()

Return response

New_spi.transfer = transfer

New_spi.close = spi.deinit

Return new_spi

Def __exit__(self):

Self.spi.close()

The changes were mostly just associating the pin numbers with the Python variables.

The remaining files were untouched. I will show them here in case changes made to Wei Lin's driver break the code I present in this article.

The controller.py file:

Class Controller:

Class Mock:

Pass

ON_BOARD_LED_PIN_NO = None

ON_BOARD_LED_HIGH_IS_ON = True

GPIO_PINS = []

PIN_ID_FOR_LORA_RESET = None

PIN_ID_FOR_LORA_SS = None

PIN_ID_SCK = None

PIN_ID_MOSI = None

PIN_ID_MISO = None

PIN_ID_FOR_LORA_DIO0 = None

PIN_ID_FOR_LORA_DIO1 = None

PIN_ID_FOR_LORA_DIO2 = None

PIN_ID_FOR_LORA_DIO3 = None

PIN_ID_FOR_LORA_DIO4 = None

PIN_ID_FOR_LORA_DIO5 = None

Def __init__(self,

Pin_id_led = ON_BOARD_LED_PIN_NO,

On_board_led_high_is_on = ON_BOARD_LED_HIGH_IS_ON,

Pin_id_reset = PIN_ID_FOR_LORA_RESET,

Blink_on_start = (2, 0.5, 0.5)):

Self.pin_led = self.prepare_pin(pin_id_led)

Self.on_board_led_high_is_on = on_board_led_high_is_on

Self.pin_reset = self.prepare_pin(pin_id_reset)

Self.reset_pin(self.pin_reset)

Self.spi = self.prepare_spi(self.get_spi())

Self.blink_led(*blink_on_start)

Def add_transceiver(self,

Transceiver,

Pin_id_ss = PIN_ID_FOR_LORA_SS,

Pin_id_RxDone = PIN_ID_FOR_LORA_DIO0,

Pin_id_RxTimeout = PIN_ID_FOR_LORA_DIO1,

Pin_id_ValidHeader = PIN_ID_FOR_LORA_DIO2,

Pin_id_CadDone = PIN_ID_FOR_LORA_DIO3,

Pin_id_CadDetected = PIN_ID_FOR_LORA_DIO4,

Pin_id_PayloadCrcError = PIN_ID_FOR_LORA_DIO5):

Transceiver.transfer = self.spi.transfer

Transceiver.blink_led = self.blink_led

Transceiver.pin_ss = self.prepare_pin(pin_id_ss)

Transceiver.pin_RxDone = self.prepare_irq_pin(pin_id_RxDone)

Transceiver.pin_RxTimeout = self.prepare_irq_pin(pin_id_RxTimeout)

Transceiver.pin_ValidHeader = self.prepare_irq_pin(pin_id_ValidHeader)

Transceiver.pin_CadDone = self.prepare_irq_pin(pin_id_CadDone)

Transceiver.pin_CadDetected = self.prepare_irq_pin(pin_id_CadDetected)

Transceiver.pin_PayloadCrcError = self.prepare_irq_pin(pin_id_PayloadCrcError)

Transceiver.init()

Self.transceivers[transceiver.name] = transceiver

Return transceiver

Def prepare_pin(self, pin_id, in_out = None):

Reason = ‘’’

Raise NotImplementedError(reason)

```
Def prepare_irq_pin(self, pin_id):
Reason = ‘’
Raise NotImplementedError(reason)
Def get_spi(self):
Reason = ‘’
Raise NotImplementedError(reason)
Def prepare_spi(self, spi):
Reason = ‘’
Raise NotImplementedError(reason)
Def led_on(self, on = True):
Self.pin_led.high() if self.on_board_led_high_is_on == on else self.pin_led.low()
Def blink_led(self, times = 1, on_seconds = 0.1, off_seconds = 0.1):
For i in range(times):
Self.led_on(True)
Sleep(on_seconds)
Self.led_on(False)
Sleep(off_seconds)
Def reset_pin(self, pin, duration_low = 0.05, duration_high = 0.05):
Pin.low()
Sleep(duration_low)
Pin.high()
Sleep(duration_high)
Def __exit__(self):
Self.spi.close()
```

Finally, the sx127x.py file:

```
Import gc
Import config_lora
PA_OUTPUT_RFO_PIN = 0
PA_OUTPUT_PA_BOOST_PIN = 1
REG_FIFO = 0x00
REG_OP_MODE = 0x01
REG_FRF_MSB = 0x06
REG_FRF_MID = 0x07
REG_FRF_LSB = 0x08
REG_PA_CONFIG = 0x09
REG_LNA = 0x0c
REG_FIFO_ADDR_PTR = 0x0d
```

REG_FIFO_TX_BASE_ADDR = 0x0e

FifoTxBaseAddr = 0x00

REG_FIFO_RX_BASE_ADDR = 0x0f

FifoRxBaseAddr = 0x00

REG_FIFO_RX_CURRENT_ADDR = 0x10

REG_IRQ_FLAGS_MASK = 0x11

REG_IRQ_FLAGS = 0x12

REG_RX_NB_BYTES = 0x13

REG_PKT_RSSI_VALUE = 0x1a

REG_PKT_SNR_VALUE = 0x1b

REG_MODEM_CONFIG_1 = 0x1d

REG_MODEM_CONFIG_2 = 0x1e

REG_PREAMBLE_MSB = 0x20

REG_PREAMBLE_LSB = 0x21

REG_PAYLOAD_LENGTH = 0x22

REG_FIFO_RX_BYTE_ADDR = 0x25

REG_MODEM_CONFIG_3 = 0x26

REG_RSSI_WIDEBAND = 0x2c

REG_DETECTION_OPTIMIZE = 0x31

REG_DETECTION_THRESHOLD = 0x37

REG_SYNC_WORD = 0x39

REG_DIO_MAPPING_1 = 0x40

REG_VERSION = 0x42

MODE_LONG_RANGE_MODE = 0x80 # bit 7: 1 => LoRa mode

MODE_SLEEP = 0x00

MODE_STDBY = 0x01

MODE_TX = 0x03

MODE_RX_CONTINUOUS = 0x05

MODE_RX_SINGLE = 0x06

PA_BOOST = 0x80

IRQ_TX_DONE_MASK = 0x08

IRQ_PAYLOAD_CRC_ERROR_MASK = 0x20

IRQ_RX_DONE_MASK = 0x40

IRQ_RX_TIME_OUT_MASK = 0x80

MAX_PKT_LENGTH = 255

Class SX127x:

Def __init__(self,

Name = ‘SX127x’,

Parameters = {‘frequency’: 433E6, ‘tx_power_level’: 2, ‘signal_bandwidth’: 125E3,

‘spreading_factor’: 8, ‘coding_rate’: 5, ‘preamble_length’: 8,

‘implicitHeader’: False, ‘sync_word’: 0x12, ‘enable_CRC’: False},

onReceive = None):

self.name = name

self.parameters = parameters

self._onReceive = onReceive

self._lock = False

if parameters: self.parameters = parameters

Version = self.readRegister(REG_VERSION)

If version != 0x12:

Raise Exception(‘Invalid version.’)

Self.sleep()

Self.setFrequency(self.parameters[‘frequency’])

Self.setSignalBandwidth(self.parameters[‘signal_bandwidth’])

Self.writeRegister(REG_LNA, self.readRegister(REG_LNA) | 0x03)

Self.writeRegister(REG_MODEM_CONFIG_3, 0x04)

Self.setTxPower(self.parameters[‘tx_power_level’])

Self._implicitHeaderMode = None

Self.implicitHeaderMode(self.parameters[‘implicitHeader’])

Self.setSpreadingFactor(self.parameters[‘spreading_factor’])

Self.setCodingRate(self.parameters[‘coding_rate’])

Self.setPreambleLength(self.parameters[‘preamble_length’])

Self.setSyncWord(self.parameters[‘sync_word’])

Self.enableCRC(self.parameters[‘enable_CRC’])

If 1000 / (self.parameters[‘signal_bandwidth’] / 2**self.parameters[‘spreading_factor’]) > 16:

Self.writeRegister(REG_MODEM_CONFIG_3, self.readRegister(REG_MODEM_CONFIG_3) | 0x08)

Self.writeRegister(REG_FIFO_TX_BASE_ADDR, FifoTxBaseAddr)

Self.writeRegister(REG_FIFO_RX_BASE_ADDR, FifoRxBaseAddr)

Self.standby()

Def beginPacket(self, implicitHeaderMode = False):

Self.standby()

Self.implicitHeaderMode(implicitHeaderMode)

Self.writeRegister(REG_FIFO_ADDR_PTR, FifoTxBaseAddr)

Self.writeRegister(REG_PAYLOAD_LENGTH, 0)

Def endPacket(self):

```
Self.writeRegister(REG_OP_MODE, MODE_LONG_RANGE_MODE | MODE_TX)
While (self.readRegister(REG_IRQ_FLAGS) & IRQ_TX_DONE_MASK) == 0:
Pass
Self.writeRegister(REG_IRQ_FLAGS, IRQ_TX_DONE_MASK)
Self.collect_garbage()
Def write(self, buffer):
currentLength = self.readRegister(REG_PAYLOAD_LENGTH)
size = len(buffer)
Size = min(size, (MAX_PKT_LENGTH – FifoTxBaseAddr – currentLength))
For i in range(size):
Self.writeRegister(REG_FIFO, buffer[i])
Self.writeRegister(REG_PAYLOAD_LENGTH, currentLength + size)
Return size
Def aquire_lock(self, lock = False):
If not config_lora.IS_MICROPYTHON:  # MicroPython is single threaded, doesn’t need lock.
If lock:
While self._lock: pass
Self._lock = True
Else:
Self._lock = False
Def println(self, string, implicitHeader = False):
Self.aquire_lock(True)  # wait until RX_Done, lock and begin writing.
Self.beginPacket(implicitHeader)
Self.write(string.encode())
Self.endPacket()
Self.aquire_lock(False) # unlock when done writing
Def getIrqFlags(self):
irqFlags = self.readRegister(REG_IRQ_FLAGS)
self.writeRegister(REG_IRQ_FLAGS, irqFlags)
self.writeRegister(REG_OP_MODE, MODE_LONG_RANGE_MODE | MODE_STDBY)
self.writeRegister(REG_OP_MODE, MODE_LONG_RANGE_MODE | MODE_SLEEP)
if (outputPin == PA_OUTPUT_RFO_PIN):
Level = min(max(level, 0), 14)
Self.writeRegister(REG_PA_CONFIG, 0x70 | level)
Else:
Level = min(max(level, 2), 17)
Self.writeRegister(REG_PA_CONFIG, PA_BOOST | (level – 2))
```

Def setFrequency(self, frequency):

Self._frequency = frequency

Frfs = {169E6: (42, 64, 0),

433E6: (108, 64, 0),

434E6: (108, 128, 0),

866E6: (216, 128, 0),

868E6: (217, 0, 0),

915E6: (228, 192, 0)}

Self.writeRegister(REG_FRF_MSB, frfs[frequency][0])

Self.writeRegister(REG_FRF_MID, frfs[frequency][1])

Self.writeRegister(REG_FRF_LSB, frfs[frequency][2])

Def setSpreadingFactor(self, sf):

Sf = min(max(sf, 6), 12)

Self.writeRegister(REG_DETECTION_OPTIMIZE, 0xc5 if sf == 6 else 0xc3)

Self.writeRegister(REG_DETECTION_THRESHOLD, 0x0c if sf == 6 else 0x0a)

Self.writeRegister(REG_MODEM_CONFIG_2, (self.readRegister(REG_MODEM_CONFIG_2) & 0x0f) | ((sf << 4) & 0xf0))

Def setSignalBandwidth(self, sbw):

Bins = (7.8E3, 10.4E3, 15.6E3, 20.8E3, 31.25E3, 41.7E3, 62.5E3, 125E3, 250E3)

Bw = 9

For i in range(len(bins)):

If sbw

Open Source RF: Exploring The ISM Bands With RTL_433

Follow

5 min read

Jul 7, 2025

Listen

Share

More

The RTL-SDR dongle can be a cheap entry into the world of radio.

If you aren’t a medium member, you can read with no paywall via

We’ve spoken before in previous articles about the utility that you can get by acquiring one of the RTL-SDR dongles. These cheap USB devices are a great introduction to the world of radio and can be picked up extremely cheaply, sometimes running at less than $20 delivered to your door.

For the purposes of today's article, we’ll assume you already have an RTL-SDR as well as a computer with either Linux or Windows with WSL. Let’s go!

Press enter or click to view image in full size

The ISM Bands

Giving unlicensed users the ability to access the radio spectrum with no license required, the ISM bands are a small chunk of spectrum that (within limitations) is available to all users.

Press enter or click to view image in full size

RTL_433

The first thing to do is check out , as you'll get the latest info on the package as well as all the details you need to get it up and running. If you're on Linux or WSL, though, it's as easy as getting it installed. Simply follow the prompts after issuing the command

apt install rtl_433

rtl_433 -h

While you can run a broad array of flags with your initial command, to start detecting signals, you'll simply need to plug in your RTL device and run

rtl_433

Providing you've set your device up correctly, you should see the following messages in your terminal as the program starts to run.

Press enter or click to view image in full size

If you've got said messages, then that's it! You're ready to go, and any ISM signals within range should be displayed in your terminal thanks to your RTL-SDR.

Press enter or click to view image in full size

What Can I Find?

Well, as it happens, quite a lot actually. A quick look at the notes to find supported protocols shows more than a few interesting vendors there. Needless to say, weather stations, temperature sensors and rain gauges are just some of the devices that you'll find as supported protocols. Look a little harder, and you'll also find LORA, Home Automation devices and possibly even some remote controls of different types. There are, in fact, over 200 protocols listed as compatible in the RTL-433 documents, so there's plenty there to keep you amused.

It's worth mentioning, though, that despite the name, 400MHz isn't the only game in town supported by the RTL hardware. You'll also get coverage at both 300 and 900 MHz, as well, meaning that you shouldn't be surprised if you observe any systems operating within that frequency.

Your Gateway Drug

Before you jump too far into this, though, here's a word of warning.

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.

■ Enjoyed this article? Join the community! ■

■ Follow us on and

■ We're now on !

■ Articles we think you'll like:

What The Tech?!

Shodan:

✉■ Want more content like this?

Higher Topics

Rtlsdr

Radio

Radio Hackers

Python Radio 36: Mesh Networking

Follow

16 min read

Apr 15, 2025

Listen

Share

More

Get the word out to nodes you can't see.

Press enter or click to view image in full size

MidJourney

Here in California's San Francisco Bay Area, we have a very active group using the Meshtastic software to form a mesh network of LoRa nodes.

I can send messages to San Francisco 54 miles away using only a few milliwatts of power and a tiny solar-powered computer that cost me about $30 to put together.

By using the built-in WiFi instead of a separate LoRa radio chip, we save cost at the expense of long distance but gain a lot of extra bandwidth.

And the little thing is the size of my thumbnail:

Press enter or click to view image in full size

Photo by author

We use the tiny antenna to make testing our mesh network easier. We want to have some nodes out of reach of at least one node to make sure we are actually networking and not talking to them directly.

The Design

The goal of the project is to make it simple.

So we will not do any fancy routing. We will broadcast each message to all the neighbors we can reach. They will receive it and broadcast it to everyone they can reach.

Each packet has a time-to-live number that is decremented before retransmitting the packet. If the number gets to zero, the node drops the packet.

Each packet has a sequence number, and if a node sees a sequence number it has already processed, it drops the packet. We keep the last 20 sequence numbers in a circular list.

That's the whole design.

Much of the code is just a simple web server.

The Code

We have two support modules, connect.py and uping.py. We don't actually need uping.py, but it made debugging the code easier at one point. It 'ping's an IP address to verify the connection.

Here is connect.py, the module that set up the WiFi access point and connects to the other nodes' access points:

It sets up an access point, advertising itself as mesh_XXXXXX, where the Xs are the last 3 bytes of the MAC address, in hexadecimal.

The first node to power up will not find any. But the second one to power up within reach of the signal will find this node and connect.

The uping.py module implements the ping command:

Unlike the other modules, I didn’t write this one. But it came in quite handy, so I share it here.

The Main Program

As you might expect, most of the work is done in the main program. Some of it is just a simple web server, something we’ve covered earlier.

The unquote(), handle_query(), decode_path(), req_handler(), and client_handler() process packets that come in and parse the few HTTP protocols we accept. There is nothing “mesh networky” about them.

The dup_sequence() method handles the circular list of sequence numbers we’ve seen. When it finds a duplicate, it returns True to drop the packet.

When a node connects to our node, add_neighbor() puts it into a list of neighbors. It ignores neighbors we already have in the list. Importantly, it calls greet() to send a message back to the connecting node, telling it who we are, and collecting similar information in the reply.

Before I go any further, here is the code:

It also handles printing out messages, and the reply to the greet() packet. The latter calls who_name() to associate the name with the IP addresses we got from the connecting node.

The whole point of a mesh network is to communicate with nodes we can’t connect to directly. That’s where the direct() method comes in. It looks at all the IP addresses we know about and returns False if the address given is unreachable.

We finally get to direct_msg(), the heart of the code. This assigns sequence numbers and inserts them into the payload, and then tries 5 times to send the packet, stopping at the first success. If it gets a reply, it adds the node as a neighbor (it lets add_neighbor() handle duplicates).

The broadcast() method is simple. It goes through the neighbor list and sends the packet to each one.

If the command was “text”, then we got it from a web browser (more on that in a bit). We repackage the data (adding our IP addresses, time_to_live, etc.) and broadcast it.

The part of the HTTP packet ahead of the query is the path. This is normally the file the server will read and send to the browser, and indeed we do that if asked. But we also handle “fake” files /text and /none. The /text file just acknowledges that we got a text message, so the web browser that sent it knows we got it. The /none file answers the greet message by returning our name, ssid, and IP addresses.

The form looks like this:

I named my nodes Alice, Bob, and Carol (the next one will be Ted).

We can see that Carol has Alice as a neighbor, but Carol can’t see Bob. I sent a message to Bob, telling Bob my browser was Joe, and Carol sent it to Alice who then sent it to Bob. The meshing worked (finally, after three weeks of coding and debugging).

The something that it does is just to print out the neighbor list on the console (if there is one) and then broadcast a message to set everyone’s LED to a random color. Again, just to show the system is up and running.

When setting up a node out somewhere here on the farm, it is nice not to need a laptop to see what's going on. Watching the LED change color tells me everything is working.

It is now easy to add features. You can collect temperature, humidity, and other weather data and send it out to the mesh. You can turn lights on and off, or tell when the well pump is running (you all have those right? Or is it just me?)

Each node has 150 megabits of bandwidth to play with (15 megabytes per second). Our typical packet length is about 100 bytes, so we could theoretically send 150,000 packets per second. We send each of our packets 3 times per node on the assumption all nodes are within 3 hops of one another, but we could easily change this, or make it an option in the web form. But the network would have to get pretty big before we ran out of bandwidth, even though we waste it by using the simple design.

Programming

Education

Software Development

Python

Python Radio 39: Secret Unbreakable Code

Simon Quellen Field

Simon Quellen Field

Follow

10 min read

May 19, 2025

Listen

Share

More

A One-Time Pad is Cryptographically Secure.

Press enter or click to view image in full size

Secret agent reading coded message.

MidJourney

How sure are you that governments can't break the encryptions commonly in use? If you were a journalist in a repressive dictatorship, would you trust your freedom or your life to them?

There is a well-known method that is proven to be safe against cryptographic attacks. It is called a one-time pad.

The idea is simple. Each person has a pad of paper that has random letters on it. They code the message by adding the next random letter to each letter of the original message. Then they burn that page or the pad.

In the 1940s, the information theorist Claude Shannon proved this was unbreakable by cryptanalytic methods.

The last two things are the concern of the two people, and the code and hardware I present here don't address them.

Never reusing a page sounds simple. We just delete the page from memory. But is it truly deleted? Can the NSA or the FSB read the deleted text and then be able to decode old messages?

We handle that by writing random text over the old page instead of deleting it. The truly paranoid can do this in a loop, so that the page gets overwritten by new random data many times. My code does it just once.

Enter the Hardware

That leaves the first requirement: that we make the pad from truly random letters. The computer can't do that by itself. It needs hardware help.

Press enter or click to view image in full size

Schematic by the author

This design uses the unpredictable breakdown of a Zener diode that is due to the quantum mechanics of electron tunneling.

I ordered the parts and started writing the code.

But as I was writing, it occurred to me that I have another handy source of random entropy.

The Benefits of Radio Static

One of the problems we encountered when dealing with the data link using the 433 MHz transmitters and receivers was the problem of noise. When the signal was strong, the receiver performed well. The signal-to-noise ratio was excellent, even at half a kilometer.

But when we weren't transmitting, we got flooded with noise. The receiver would increase its sensitivity until it heard something, even when there was nothing to hear.

And I already had the receiver. All I needed to do was listen to the output when the transmitter was off. That radio noise has all the entropy we need. We don't even need an antenna.

I connected the power, ground, and output of the receiver to an ESP32-S3 and collected some data using the analog-to-digital port.

Some Software Assistance

Either hardware source has plenty of entropy (randomness). But we won't assume that there isn't some bias in there somewhere. Many of the samples will be zero, since I didn't bother to add a DC bias to bring negative samples into the range of the ADC. We can handle all of that in software if we collect more entropy than we need.

We are now ready to walk through the code.

Freq(240_000_000)

Green_led = Pin(34, Pin.OUT)

Hex_digit = {

“a”: 10,

“b”: 11,

“c”: 12,

“d”: 13,

“e”: 14,

“f”: 15,

Def df():

S = statvfs('//')   # Note: '//' represents the root filesystem

Block_size = s[0]

Free_blocks = s[3]

Print(f"Block size: {block_size}")

Print(f"Free blocks: {free_blocks}")

Return int(free_blocks * block_size)

Class OTP:

Def __init__(self):

Self.code = ""

Self.unused = 0

B = None

Try:

With open("one_time_pad.nxt", "rb") as f:

B = f.read()

Except OSError as e:

Pass

If b:

Self.unused = int.from_bytes(b, 'little')

Else:

Print(f"Next address is starting at {self.unused}")

Self.last_unused = self.unused

Def in_hex(self, b):

S = ""

For x in b:

S += f"{x:02x}"

Return s

Def update(self):

B = self.unused.to_bytes(4, 'little')

With open("one_time_pad.nxt", "wb") as f:

f.write(b)

if self.unused == 0:

size = self.unused – self.last_unused

random = (int.from_bytes(self.code[0:4], 'little') + ticks_cpu()) % 1_000_000

with open("one_time_pad.otp", "r+b") as f:

b = b''

f.seek(random, 0)

b = f.read(4)

```
random = (int.from_bytes(b, ‘little’) + ticks_cpu()) % 1_000_000
f.seek(random, 0)
b = f.read(size)
f.seek(self.last_unused, 0)
backward = bytes(reversed(b))
f.write(backward)                    # Clobber used-up pad bytes with random data
try:
remove( “one_time_pad.nxt”)
except OSError as e:
pass
try:
remove( “one_time_pad.otp”)
except OSError as e:
pass
Def create_one_time_pad(self):
Self.unused = 0
Leave_free = 512 * 1_024
Free = df() / 2                      # Don’t use more than half the remaining flash
Print(f”Filesystem free space: {free}”)
If free < leave_free:
Return
Adc = ADC(Pin(10))
Try:
Stat(“one_time_pad.otp”)
Print(“One time pad already exists”)
Return
Except OSError:
Pass
With open(“one_time_pad.otp”, “wb”) as f:
Start = ticks_ms()
Count = 0
While free > leave_free:
Noise_string = b’’
For characters in range(25):
Noise = 0
For foo in range(20):
Noise += int(adc.read())
```

Noise_string += noise.to_bytes(4, ‘little’)

Sha = sha256(noise_string).digest()

f.write(sha)

free -= len(sha)

left = free – leave_free

count += len(sha)

elapsed_seconds = ticks_diff(ticks_ms(), start) / 1_000

rate = count / elapsed_seconds

mins = int((left / rate) // 60)

hrs = int(mins // 60)

mins -= hrs * 60

Print(f"time left:{hrs:2d}:{mins:02d}", end="\r")

Self.unused = 0

Self.last_unused = 0

Self.update()

Def decrypt(self, pwd, hex_msg):

Pl = len(pwd)

Msg = hex_msg[0:8]

For x in range(8, len(hex_msg), 2):

Msg += chr((hex_digit[hex_msg[x]] << 4) | hex_digit[hex_msg[x+1]])

Unused = int(msg[0:8], 16)

L = len(msg) – 8

Out = “”

With open(“one_time_pad.otp”, “rb”) as f:

f.seek(unused, 0)

self.code = f.read(l)

p = ord(pwd[x % pl])

out += chr(p ^ self.code[x] ^ ord(msg[x+8]))

try:

msg_length = int(out[0:8], 16) + 8

except ValueError as e:

print(“Decode failed. Don’t decode on the same device you encode on.”)

if unused > self.unused:

self.unused = unused # Stay matched with the other device so we can reply

pl = len(pwd)

l = ((len(msg) // 32) + 1) * 32

```
out = bytearray(b'\x00' * (l+8))
out[:8] = bytearray(f"{self.unused:08.8x}", "utf-8")
with open("one_time_pad.otp", "rb") as f:
f.seek(self.unused, 0)
try:
self.code = f.read(l)
except Exception as e:
print_exception(e)
p = ord(pwd[x % pl])
out[x+8] = int(p ^ self.code[x] ^ ord(msg[x]))
start = len(msg) + 8
stop = l
print(cnt)
out[cnt] = int(self.code[cnt])
print("Test decode: ", end="")
test = bytearray(b'\x00' * len(msg))
p = ord(pwd[x % pl])
test[x] = int(p ^ self.code[x] ^ out[x+8])
if x >= 32 and x < 127:
print(chr(x), end="")
else:
print(".", end="")
print()
self.unused += l
self.update()
Green_led.on()
While True:
Test = input("Create, Remove, Encrypt, or Decrypt? ")
If len(test) == 0:
Print("Type c, r, e, or d")
Elif test[0].lower() == 'c':
Self.create_one_time_pad()
Elif test[0].lower() == 'r':
Confirm = input("Really delete the one time pad that takes a long time to build? ")
If confirm == "yes":
Self.delete_one_time_pad()
Elif test[0].lower() == 'e':
```

Pwd = input(“Enter password: “)

Msg = input(“Enter message to encrypt: “)

Coded = self.encrypt(pwd, f”{len(msg):08.8x}” + msg)

Elif test[0].lower() == ‘d’:

Pwd = input(“Enter password: “)

Msg = input(“Enter message to decrypt: “)

If msg != “”:

Decoded = self.decrypt(pwd, msg)

Print(f”Decoded: {decoded}”)

Else:

Print(“Type c, r, e, or d”)

Def main():

Otp = OTP()

Otp.ui()

Main()

We’ll start with the create_one_time_pad() method.

We want a large pad, but we don’t want to completely fill up our flash memory. We find out how much space we have, use only half, and leave at least half a megabyte free.

Now we collect some random bits. We add up 20 bytes from the ADC to get a 4-byte integer. We convert that into a 4-byte string and append it to our string of random characters. We do this 25 times to get 100 bytes of random noise.

We hand that string to the SHA256() method, and write the resulting 32 bytes to our one-time pad. We continue doing this until the pad is full.

This can take about 15 minutes, so we want to give the user some indication of progress. We calculate the rate at which the file grows and give an estimate of the completion time. On smaller devices with only 4 megabytes of flash, it may take five minutes or less.

Finally, we call the update() method.

Update() has two jobs. It keeps track of which part of the pad we have used in a file called one_time_pad.nxt. Its second job is to clobber the part we have used. We want to put random data there. But where can we find random data? Oh, yes! We have a whole file full of it!

We take 4 bytes of our last random patch and add the clock to it. Then we use that to seek into the file to get another one. We keep doing that until we get something that isn’t zero. I did this because it was convenient during debugging to kill the program before the whole 15 minutes were up, and the file was shorter than a megabyte. Randomly seeking would often lead off the end of the file, returning zeros.

Now we are ready to encrypt a message.

You have probably used two-factor authentication before as a security method. Three-factor authentication is a little better. The three parts are:

Something you know (such as a password).

Something you have (in our case, that is our one-time pad).

Something you are (this is where fingerprint readers or iris scans come in).

The password and the message to encode get passed to the encrypt() method.

Encrypt() adds the address into the pad as the first 8 bytes. This is not encrypted.

To keep an adversary from knowing how long our message is, we bump the length up to the next 32 bytes. Our encrypted coded message will always be a multiple of 32 bytes long.

The nice feature of the XOR function is that you can run it twice and get the original message back. That is how we decrypt. To make sure we did it right, we decrypt it right away and present it to the user. She already knows the message, so we haven't given anything away.

Lastly, we call update to save the address and clobber the code.

The only part left is the main() function, which is mostly self-explanatory.

It collects the password and message and adds the message length before calling encrypt(). At the other end, the message recipient will paste the message into decrypt() to read it.

The Results

Let's look at the code in action. Here is the encoding:

Create, Remove, Encrypt, or Decrypt? E

Enter password: this is a longish password with lots of entropy and yet still something I can remember.

Enter message to encrypt: This is a test of our one-time pad encryption mechanism.

Test decode: 00000038This is a test of our one-time pad encryption mechanism.

Coded:
b1b51b0081b606d8fe8ff404589622fdd606997e09829b88c019469e8c260942583acc8fc8a37aab28e3da5f829697d

Create, Remove, Encrypt, or Decrypt?

Next, we copy the coded message into the decoding machine:

Create, Remove, Encrypt, or Decrypt? D

Enter password: this is a longish password with lots of entropy and yet still something I can remember.

Enter message to decrypt:
b1b51b0081b606d8fe8ff404589622fdd606997e09829b88c019469e8c260942583acc8fc8a37aab28e3da5f829697d

Decoded: This is a test of our one-time pad encryption mechanism.

Create, Remove, Encrypt, or Decrypt?

Make sure you don't try to decrypt the message on the same machine you encrypted it on. That machine has already clobbered the one-time pad used to encrypt the message.

Some Words About Downloading

Our code requires the hashlib library. We install that using the mpremote program running on the host machine (a Windows machine in my case):

Mpremote connect com4 mip install hashlib

We build the one-time pad on one esp32, and then copy the two files to the other esp32:

Mpremote connect com4 fs cp :one_time_pad.nxt one_time_pad.nxt

Mpremote connect com4 fs cp :one_time_pad.otp one_time_pad.otp

Mpremote connect com8 fs cp one_time_pad.nxt :one_time_pad.nxt

Mpremote connect com8 fs cp one_time_pad.otp :one_time_pad.otp

The colon tells mpremote we are talking about a file on the remote machine. And (of course) you will change com4 and com8 to the ports you use on your own machine. On a Mac or Linux, they will be found in /dev.

Lastly, you must securely delete the one_time_pad.otp from your host machine. Write over it several times with random data.

Python Radio 41: Radar!

Follow

5 min read

May 30, 2025

Listen

Share

More

Exploring microwave signals above 3 GHz.

Press enter or click to view image in full size

All images by the author.

They don’t operate the same way that aircraft radars or weather radars do. Those send out pulses and time the reflections to get the range and detect wavelength changes to get the speed towards or away (using the Doppler effect).

Despite the advertising, these little boards detect the changes in the phase of the reflected waves by heterodyning them (“beating”) with the continuous-wave transmissions. The interference of the forward and reflected waves cause changes in the output amplitude that are in the audio range or lower.

Press enter or click to view image in full size

In the photo above, I have soldered a red LED between the output and ground, and powered the device from a 3.7-volt battery connected to the +3.3-volt input. The device can handle 4.2 volts on that input, but it also has a VIN input that connects to a voltage regulator, allowing as much as 28 volts to be used. It will run a long time on a 9-volt battery that way, as it consumes less than 3 milliamperes of current (more if the LED is lit).

But we aren’t going to use the board in its normal mode. We’re going to hack it to do much more.

Let’s look at a schematic of the board:

Press enter or click to view image in full size

The transmitter is the transistor at the right, forming an oscillator at about 3.175 GHz.

The antenna both sends out the signal and receives the echo. The echo signal is mixed in that same transistor, forming the sum and difference frequencies, just like a superheterodyne receiver. The sum frequency is filtered out on the way to pin 14 of the integrated circuit, leaving the low-frequency difference signal.

Below is the schematic of the integrated circuit:

Press enter or click to view image in full size

We are going to ignore most of it. What we want is the operational amplifier labeled OP1. The weak difference signal from the antenna and mixer comes in on pin 14. We want the amplified version of that on pin 16.

While it is possible to solder a thin wire to pin 16 on the IC, that's a little tricky because the pins are very small and close together.

But we're in luck.

Notice that one side of the resistor connects to pin 16.

Press enter or click to view image in full size

Here I have soldered a 30-gauge wire to the left solder point of where R3 would go.

Now we have the unprocessed signal from the receiver. We can feed it into an ESP32-C3 Super Mini's analog-to-digital converter and look at the signals.

Let's first look at it on the oscilloscope:

Press enter or click to view image in full size

Here I was moving closer to the board, causing a 3.329 Hz signal on pin 16.

The peak-to-peak voltage was 0.344 volts, well within the range of our ADC.

Now we can connect it to the Super Mini:

Press enter or click to view image in full size

We use pins 5, 6, and 7 on the Super Mini.

Pin 5 will be the ADC input.

Pin 6 we will set to HIGH to power the board.

We have turned the device into a 3.16 GHz Morse Code receiver.

Press enter or click to view image in full size

As you can see, I went all-out on the construction details. The code key acts as a switch, powering up the board from a 9-volt battery connected to VIN.

The red LED is the one I soldered to the board to use it as a motion detector. It still detects motion, but we ignore that.

Now, when I tap out Morse Code on the transmitter, I see the little blue LED on the Super Mini light up in sympathy.

Why did I use pins 6 and 7 to power the board instead of 3.3 volts and ground?

Because we can use the Super Mini to key the board. We can use the program we built in , and have it use Pin 6. We don't even need the extra transistor we used in that project.

Our 3.16 GHz transmitter has a wavelength of 94 millimeters. About the length of a cigarette. It travels through wood easily, but gets blocked by glass and metal.

Programming

Python

Radar

Radio Hackers: The SATNOGS Network(Pt 2)

Investigator515

Investigator515

Follow

7 min read

Mar 21, 2025

Listen

Share

More

Taking our ground station from cold and dark to a functional telemetry station.

In this previous article, we started looking at the Satnogs project. This project is an open-source, satellite ground station that collects data from in-orbit satellites. This data is then fed to the central Satnogs website where you'll be able to contribute your data right alongside the data of others. Overall, projects such as this aim to improve the efficiency of space-based assets by providing live, real-time data that can help monitor the health of spacecraft as they pass overhead.

Press enter or click to view image in full size

Satnogs aims to provide a live, real-time dashboard that helps monitor satellite health in real-time. Source: Satnogs.org

Let's Build

1x Raspberry Pi + Accessories

1x SDR Unit (HackRF, BladeRF, SDRPlay RTL-SDR etc)

1x. Antenna with Feedline.

You should also have an SD card that is preloaded with the Satnogs disk image. If you haven't organised that yet, you can find it here

Press enter or click to view image in full size

To do this, we'll simply visit the Libre Space community and register a new account as per the image above. From here, we can start to make changes to our station and apply the final configurations.

The Ground Station

Once we're into our account and the email has been verified we can move to the next step, which is adding our ground station. There are a few ways that you can look at doing this and the block diagram below gives a great breakdown of how this might work and what we'll need.

Press enter or click to view image in full size

If you aren't using ground-based preamplifiers or antenna rotators then your own block diagram will look a little different to this. Remember though, you won't need to have everything. While an omnidirectional antenna without a rotator has a distinct disadvantage in comparison to a tracked Yagi, you can still get started with something simple and carry out upgrades later on. Just remember that each time you add a change, update your details in the Satnogs portal so they remain current.

Let's visit the portal now and add our new ground station. Then, we can get the configuration sorted and start sending data to the community.

When we're in the portal, we'll want to select the Add Ground Station tab as seen in the image below.

Once we've done that step, you'll be taken to a fresh page where you'll put in the technical information regarding how your station runs. Here, we'll need to add info about our overall configuration that's in use, a description of any antennas we might be running as well as any other relevant information.

Press enter or click to view image in full size

Most of this is pretty simple stuff, but there are a few things to be aware of. Firstly, an accurate location is required to ensure that your station is correctly integrated into the network. You'll need to set the location in the "Advanced Edit" in the format of Lat/Long

Press enter or click to view image in full size

Once you've completed this step, the next is to add and configure an antenna. This is an important step, regardless of what type of antenna you might be using.

Press enter or click to view image in full size

Web Portal Setup

The next step is ensuring that we're able to feed data properly from our SDR unit into the Satnogs web portal. To do this, simply open your terminal and hit it with the following command

Sudo satnogs-setup

The Satnogs setup menu. Source: Wikipedia.com

If we're using Raspbian to configure a system, typically we will need to run the usual update & upgrade commands. In this instance though, the Satnogs software is pretty efficient. By running the setup command, we've also instructed the system to fetch all relevant updates as needed.

Most of this configuration will happen automatically with little input needed from your good selves. However, there are a few things we'll need to double-check before we put things online.

Don't Forget This!

Satnogs works as a global feeder so to ensure it doesn't go hungry, we'll need to keep it fed with plenty of data. To do this, we'll need an API key so that we can feed it data via the backend.

Press enter or click to view image in full size

To set this up correctly, we'll need to visit the portal again and then acquire our key. First, we'll visit the dashboard, then hit the tab that says API Key.

Press enter or click to view image in full size

Press enter or click to view image in full size

The Test Run

When setup is complete and you've got your ground station up and running, you'll need to check to ensure that it shows up as online.

Once that's done, really there is only one thing left to do and that's to start testing by capturing packets!

Press enter or click to view image in full size

When you have a satellite scheduled to arrive overhead, you should see your station going through the following motions as per the image above. If you're able to see this in your own terminal, then you can rest assured that everything is working as it should on your end as well.

Reminder: The best way to log in via SSH is by using a key instead of a password. If you must use a password though, ensure it is strong and unique.

The Backup

Because of this, it's worth grabbing a backup of the image on the SD card so that the current card can be flashed should circumstances require it. You can do this by doing a straight duplication, however it's also wise to copy your configuration file.

Did You Build?

You'll find some great resources in the Satnogs wiki that discusses these upgrades and these are particularly helpful in understanding exactly what kind of improvements each change will make.

Space communication can often be the pinnacle of weak signal work. Small transmitters, with small antennas in small satellites thousands of km away, mean that when you're stuck with omnidirectional antennas, the challenge is on.

You'd be surprised at the difference some small, well-thought-out changes can make to your received signal strength levels. Also, if you're building your own Satnogs ground station, don't forget to post in on socials with a tag to show off your build.

■ We're now on Bluesky!

Articles we think you'll like:

What The Tech?! Space Shuttles

Shodan: A Map of the Internet

The Satnogs Project: Selecting Your Antenna (Pt 3)

Investigator515

Investigator515

Follow

6 min read

Jul 22, 2025

Listen

Share

More

The right antenna will determine the success (or failure) of your satellite receiving station.

If you aren't a medium member, you can read with no paywall via substack

In this previous article, we took a look at some of the steps we'd need to cover to set up and configure a satellite receiving station that fed data to the Satnogs network as an independent node.

However, there are two distinct issues here that we'll need to cover. Firstly, signals from space and more particularly cubesat signals tend to be quite weak. This is partly due to the distance travelled as well as the fact that on a cubesat board, space is at a premium (pardon the pun). This means that the output power levels are typically quite low.

Press enter or click to view image in full size

Broadband or Fixed

Part of the reason the stock antenna is so terrible is that it's designed to be a broadband system. However, the Satnogs network will typically aim to collect data from satellites that have downlink frequencies within the 2 meter or 70 cm amateur radio bands.

Alternatively, if space is at a premium, we can even try our luck using a dual-band system, that means we can stick with a single antenna.

While the option is available to use tracked, directional antennas like Yagi antennas on a rotator, due to the complexity of such a system we'll be sticking with well-designed, yet basic omnidirectional systems.

Press enter or click to view image in full size

Antenna Options

Now that steered yagis are eliminated, we can get to looking at our available options.

Thanks to the SAtnogs community, we have 3 options to choose from, with each varying slightly in terms of overall build difficulty.

If you're after an easy option to get started, you might find the simple-to-make Turnstile antenna to be your best bet.

Press enter or click to view image in full size

If you're willing to spend some extra effort and don't mind working with coax, the Lindenblad might be a better option. You'll need a phasing harness to make it work correctly, but you can make these yourself if you're patient enough.

Press enter or click to view image in full size

The top shelf option, though, would be the Quadrafilar Helix Antenna or QFH. Providing great performance at weak signal levels, the QFH is a more advanced design that will put your homebrew skills to the test.

Making antennas can be a fun part of the radio journey. While antenna theory alone can and does fill entire books, you'll find enough info to get you started in the Satnogs Wiki.

If At First You Don't Succeed…Use Preamps

If you haven't played with radios before, you might be surprised at just how much of an improvement adding a preamp gives. While tracked, high-gain antennas come with their own performance benefits, even systems like that can benefit from adding a preamp to the end design.

Remember, though, there's no rule that says you need to build your entire station in one go. So, if you'd like, you can build yourself a basic system now and then add on extra components when you're ready to.

One Last Thing

Now we have downloaded our image, configured our station, given it a great set of ears and then put it through testing to ensure the station works properly. We're now well on the way to being able to put our station into service as a Satnogs feeder node.

There is one more thing you need to consider before we finish, though. In fact, some radio nerds might say it's the most important part of the system, and it's a topic we've covered in recent Radio Hackers articles.

Press enter or click to view image in full size

Did You Build?

Have you explored the Satnogs project or built your own receive station? Maybe you've found another interesting radio project to experiment with using the RTL-SDR or have been hacking on a hardware project of your own.

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.

Enjoyed this article? Join the community!

Higher Topics

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

Python Radio 22: Wi-Fi

Simon Quellen Field

Simon Quellen Field

Follow

8 min read

Sep 9, 2024

Listen

Share

More

Run a web server on a tiny computer.

Press enter or click to view image in full size

MidJourney

We can get to the REPL over Wifi, which can be quite convenient if the ESP32 is at the top of a pole controlling a weather station, or atop a tower holding radio antennas.

To do this is fairly simple. We put these two lines in boot.py:

Import webrepl

Webrepl.start()

And then create a file called webrepl_cfg.py that has a password in it:

PASS = ‘simon’

The ESP32 will then advertise itself so we can connect to it over Wifi, and enter in a web browser.

We can make it even easier by having the ESP32 connect to our local Wifi:

Wlan = WLAN(mode=WLAN.STA)

Nets = wlan.scan()

For net in nets:

If net.ssid == ‘mywifi’:

Print(’Network found!’)

Wlan.connect(net.ssid, auth=(net.sec, ‘mywifikey’), timeout=5000)

While not wlan.isconnected():

Print(’WLAN connection succeeded!’)

Break

Now we can simply enter its IP address on the local network into our browser, and save having to connect via Wifi to the ESP32 directly.

If you want the device to always have the same IP address (which makes it much easier to connect) you can put this code in boot.py:

```
Import machine

Wlan = WLAN() # get current object, without changing the mode

If machine.reset_cause() != machine.SOFT_RESET:

Wlan.init(WLAN.STA)

Wlan.ifconfig(config=('192.168.178.107', '255.255.255.0', '192.168.178.1', '8.8.8.8'))

If not wlan.isconnected():

Wlan.connect('mywifi', auth=(WLAN.WPA2, 'mywifikey'), timeout=5000)

While not wlan.isconnected():
```

Of course, the IP address can be read from a file on the ESP32, so that each device can share the same code, and only the configuration file has to be unique to each device.

Making the ESP32 present a web page with buttons that control things is a little more involved. But once we have a few helper classes on the machine, the actual part left to do becomes simple.

Here is the HTML of the web page we want to serve:

```
<html>

<head>

<title>

Simon's LED

</title>

<script>

Function on()

Let xhttp = new XMLHttpRequest();

Try

Xhttp.open( "GET", "/?on", true );

Xhttp.send();

Catch( err )

Console.log( "Caught error executing /?on" );

Function off()

Let xhttp = new XMLHttpRequest();

Try

Xhttp.open( "GET", "/?off", true );

Xhttp.send();

Catch( err )

Console.log( "Caught error executing /?off" );

</script>
```

```
</head>

<body>

<h1>

Simon’s LED

</h1>

<p/>

<br/><button type=”button” onclick=”on();”>On

<br/><button type=”button” onclick=”off();”>Off

</body>

</html>
```

All it is going to do is present two buttons that will turn on or off the backlight on the TTGO Display. Or any LED connected to pin 4 on any ESP32.

When we connect to the ESP32’s IP address in our browser, we get this page:

Press enter or click to view image in full size

Image by the author

Our main.py module controls getting connected to the Wifi and setting up the web server, but critically, it also handles what the buttons do:

```
Led = Pin(4, Pin.OUT)

Def callback(q):

Print(”Query is”, q)

If q == “on”:

Print(”Turning backlight On”)

Led(1)

Elif q == “off”:

Print(”Turning backlight Off”)

Led(0)

Return None

Def main():

Con = Connect(”MyLED”)

Con.verbose = True

Con.reconnect()

Con.verbose = True

Web = WebServer(callback, False)

Web.verbose = True

Def web_task():

While True:

Await web.serve()
```

```
Loop = get_event_loop()

Loop.create_task(web_task())

Loop.run_forever()

Loop.close()

Main()
```

Now we just need to look at the helper classes Connect and WebServer.

But, since we were kind enough to put the local network address in the name, we could instead avoid connecting to the ESP32’s Wifi, and instead just send our browser to the IP address in the name, using our normal local network. If your computer only has one Wifi port, this makes it much easier to go back and forth between the Internet and the ESP. You don’t have to keep switching Wifi servers.

The connect.py module looks like this:

```
Class Connect:

Def __init__( self, who ):

Self.ssid = ”

Self.who = who

Self.who_am_i = who

Self.verbose = True

Self.network_list = []

Self.sta = WLAN( STA_IF )

Self.sta.active( True )

Self.ap = WLAN( AP_IF )

Self.ap.active( True )

Self.ap.config( essid=self.who_am_i, authmode=AUTH_OPEN )

Def read_config_file( self ):

Self.known_networks = []

Try:

F = open( “network.cfg” )

If f:

Text = f.readline()

While text:

Self.known_networks.append( text.rstrip( “\r\n” ) )

Text = f.readline()

f.close()

except Exception as e:

print( “Read_config_file():”, e )

try:

self.network_list = self.sta.scan()
```

except Exception as e:

print(“read_config_file():”, e)

self.sta.disconnect()

self.sta.active(False)

self.sta = WLAN(STA_IF)

self.sta.active(True)

Def is_available(self, name_comma_password):

Name, pasw = name_comma_password.split(“,”)

For net in self.network_list:

Ssid = net[0]

Target = ssid.decode(“utf-8”)

If name == target:

Return True, name, pasw

Return False, ”, ”

Def do_connect(self):

If self.verbose:

Print(“Connect”)

Try:

Self.sta = WLAN(STA_IF)

Self.sta.active(True)

If not self.sta.isconnected():

Self.read_config_file()

For id in self.known_networks:

Known, self.ssid, paswd = self.is_available(id)

If known:

If self.verbose:

Print(“We know”, str(self.ssid))

Try:

Self.sta.connect(self.ssid, paswd)

Count = 0

While not self.sta.isconnected():

If count > 10:

Return False

Sleep(1)

Count += 1

Ip = self.sta.ifconfig()[0]

Self.who_am_i = self.who + “ “ + ip

Self.sta.config(dhcp_hostname=self.who_am_i)

Self.ap.config(essid=self.who_am_i, authmode=AUTH_OPEN)

Return True

Except Exception as e:

Print(“Error in do_connect():”, e)

Pass

Else:

If self.verbose:

Print(“Already connected”)

Return True

Except Exception as e:

Print(“do_connect():”, e)

Return False

Def keep_trying(self):

While not self.sta.isconnected():

Try:

Self.do_connect()

Except Exception as e:

Print(“Error in keep_trying():”, e)

Pass

If self.verbose:

Ip = self.sta.ifconfig()[0]

Print(”Connected to”, ip)

Def reconnect(self):

If self.verbose:

Print(“Reconnecting”)

Self.sta.disconnect()

Self.sta.active(False)

Self.keep_trying()

BirdfarmOffice2,12345678

BirdfarmBalcony,

Birdfarm2TVRoom,

BirdfarmGym,

BirdfarmBarn,

BirdfarmGuest,

DLINK,12345678

Simon’s pocket,

NETGEAR28,12345678

Netgear28,12345678

Netgear28,12345678

PubNetPatio,6503632620

PubNetBar,6503632620

Import uasyncio as asyncio

Head = "HTTP/1.1 200 OK

Content-Type: text/html

Connection: Closed

Class WebServer(object):

Def __init__(self, callback, captive_portal=False):

Import usocket as socket

Self.call = callback

Self.conn = None

Self.s = None

Self.addr = None

Self.path = "

Self.query = "

Self.request = "

Self.method = "

Self.verbose = False

Self.timeout = 3

Self.captive_portal = captive_portal

Try:

If self.captive_portal:

Self.cp = CaptivePortal(self.verbose)

Self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

Self.s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

Self.addr = socket.getaddrinfo('0.0.0.0', 80)[0][-1]

Self.s.bind(self.addr)

Self.s.listen(5)   # Allow 5 connections before refusing any new connections

Self.poller = poll()

Self.poller.register(self.s, POLLIN)

Except OSError as e:

Print("Can't create socket", e)

Sleep(20)

Reset()

```
Def decode_path(self, req):
If not req:
Return "/"
Cmd, headers = req.split("\r\n", 1)
Self.method, self.path, protocol = cmd.split(" ")
Self.query = ''
R = self.path.find('?')
If r > 0:
Self.query = self.path[r+1:]
Self.path = self.path[:r]
If self.path == '/':
Self.path = "index.html"
Else:
Self.path = self.path[1:]
Self.verbose and print("Query:", self.query, "Path:", self.path)
Return '/' + self.path
Self.verbose and print("Send file:", filename)
If filename == "connecttest.txt":
Return True
If filename == "wpad.dat":
Return True
Try:
F = open(filename, "rb")
Except OSError as e:
Print("Can't open", filename, e)
Return False
While True:
Try:
Block = f.read(100)
If len(block) > 0:
Await self.writer.awrite(block)
Else:
Break
Except OSError as e:
Print("Can't send", filename, e)
f.close()
f.close()
```

await self.writer.awrite(head)

await self.writer.awrite(html)

self.decode_path(self.request)

self.verbose and print(”serve”)

self.request = ”

html = None

if self.captive_portal:

self.cp.handle_dns()

self.conn = None

Got_something = self.poller.poll(1)   # 1 ms timeout

If got_something:

Try:

Self.conn, address = self.s.accept()

Self.conn.setblocking(False)

Self.call(”accepted”)

Break

Except OSError as e:

Self.conn.close()

If e.args[0] != ETIMEDOUT:

Print(”Exception in accept:”, e)

Sleep(10)

Reset()

Else:

Return False

Else:

Await asyncio.sleep(0.2)

If self.conn:

Self.verbose and print(’\nConnect from’, self.addr[0])

Self.reader = asyncio.StreamReader(self.conn)

Data = await self.reader.read(-1)

Self.request = data.decode(”utf-8”)

Data = None

Self.parse_request()

Self.request = ”

Answer = None

If self.query:

Answer = self.call(self.query)

If answer:

Try:

Await self.send_some_html(answer)

Except Exception as e:

Print(”Can’t send html”, e)

Else:

If not await self.send_file(self.path):

Self.send_file(”404.html”)

Await self.reader.aclose()

Await self.writer.aclose()

Self.path = ”

Self.query = ”

Self.method = ”

Self.call(”closing”)

Self.conn.close()

Self.call(”done”)

Return True

If there was no query, it reads a file from the flash directory and sends that. This is how index.html and favicon.ico are served.

That’s quite a bit of work just to turn on an LED, but to make the ESP32 do something else we only need to make a few changes to index.html and main.py.

Wifi

Python Programming

LoRaWAN — Everything You Need to Know About The Global IoT Standard

What you should know before starting any LoRaWAN project.

Armando Rodrigues

Armando Rodrigues

Follow

6 min read

Oct 31, 2024

Listen

Share

More

Press enter or click to view image in full size

Blue and black logo, spelling LoRaWAN.

Source.

Trying to start a DIY IoT project but don’t know where to start? Or maybe just starting to learn about IoT technologies but getting overwhelmed with all the different protocols and standards? Don't worry, I've got you covered! In this article I cover one of the most popular IoT standards among hobbyist and professionals: LoRaWAN, a standard built on top of LoRa. Here you'll find some of the most important concepts on this topic. I did my best to present them in an easy-to-digest manner.

LoRaWAN Standard

LoRaWAN is a LPWAN (Low-Power Wide Area Network) standard maintained by the LoRa Alliance, an open, non-profit association with the mission of promoting the standard, which the organization claims to be the leading IoT LPWAN specification.

Network Architecture

Press enter or click to view image in full size

Diagram displaying network servers in the center gateways on the left and application servers on the right.

LoRaWAN network in a star topology.

LoRaWAN networks use a star topology. These networks consist of end-nodes/end-devices, gateways, network servers and application servers.

The working of gateways is explained in The Things Network (TTN) website:

Each gateway is registered (using configuration settings) to a LoRaWAN network server. A gateway receives LoRa messages from end devices and simply forwards them to the LoRaWAN network server. Gateways are connected to the Network Server using a backhaul like Cellular (3G/4G/5G), WiFi, Ethernet, fiber-optic or 2.4 GHz radio links.

Network servers manage the whole network. They filter duplicate messages received from several gateways, route uplink application payloads to the correct application server, provide acknowledgments of messages, send Adaptive Data Rate (ADR) messages, handle join requests and execute many more security- and network-related functions.

Application servers process application-layer payloads and generate downlink application messages (downlink Medium Access Control (MAC) commands, which are network and end-node control messages are generated by network servers).

End-nodes are usually made up of microcontrollers connected to sensors, actuators, or both. These devices are usually battery-powered, have LoRa modulators and implement the LoRaWAN protocol stack in firmware. Every end-device must be registered with a network before sending and receiving messages. The process of joining a network is called "end-device activation". An end-node can be permanently tied to a pre-selected network using Activation By Personalization (ABP) or it can search and request to join a network using over-the-air activation (OTAA).

Devices that are part of a network with location-aware gateways can have GPS-free geolocation capabilities using a trilateration technique based on timestamps sent from the network.

LoRaWAN Network Protocol

Press enter or click to view image in full size

Diagram with a representation of LoRaPHY radio packets.

Representation of LoRaPHY radio packets, LoRaWAN link layer frames and network layer packets. The size of the physical radio packets is not specified because it is measured in symbols (not bytes) and it is highly variable as preamble and header length depends on the hardware (not on the LoRaWAN standard).

Device Classes

Press enter or click to view image in full size

Diagram with a representation of LoRaWAN classes’ transmission windows

LoRaWAN classes’ transmission windows.

All the diagrams in this article were created by the author.

IoT

Lorawan

Lora

Advanced Topics

Python Radio 18: DominoEX Mode

Simon Quellen Field

Simon Quellen Field

Follow

10 min read

Sep 5, 2024

Listen

Share

More

Press enter or click to view image in full size

Image by the author

DominoEX (often just called Domino, since the previous versions such as DominoF are no longer in use) has 6 baud rates. The lowest, DominoEX4, runs at 3.90625 baud, uses 173 hertz of bandwidth, and manages as much as 25 words per minute. The other rates are DominoEX5, 8, 11, and 22 (the actual baud rates are 5.3833, 7.8125, 10.766, 15.625, and 21.533). The words per minute are respectively 31, 50, 70, 100, and 140.

There is also a DominoEX Micro mode that uses 2 baud, DominoEX44, and DominEX88 that were not in the original specification.

Like FSQ, DominoEX uses Incremental Frequency Keying, where symbols are defined by the difference between tones, rather than the tones themselves. This makes tuning easy (you can be 200 hertz off frequency and still decode the message) and allows it to tolerate frequency drift.

There are 18 tones, and the varicode tables are grouped into sets of three tones. The dominoex_varicode.py module looks like this:

Dominoex_varicode = [

The dominoex_config.py module looks much like the ones from MFSK and FSQ:

Class DominoEXConfig:

Def __init__(self, baud, frq, call, location):

Self.dds = Radio()

Self.is_beacon = False

Self.beacon_interval = 30.0

Self.message = ‘’

Self.usb_offset = 1350.0

Self.num_tones = 18

Self.incremental_tone = 0.0

Self.all_done = False

Self.r = DOMINOEX(self.send_tone, self.report_all_done)

Self.set_baud(baud)

Self.frequency = frq

```
Self.call = call
Self.location = location
Self.r.set_frequency(frq)
Self.r.set_call(call)
Self.r.set_location(location)
Def get_radio(self):
Return self.dss
Def send_code(self):
Self.dds.send()
Def send_tone(self, tone):
Self.incremental_tone = (self.incremental_tone + float(tone) + 2) % self.num_tones
Self.f = int(int(self.frequency) + self.usb_offset + (self.incremental_tone + 0.5) *
self.tone_spacing – self.bandwidth / 2.0)
Self.dds.set_freq(0, self.f)
Self.dds.send()
Def report_all_done(self):
Print()
Print("All done!")
Self.all_done = True
If self.is_beacon:
Self.r.stop()           # stop sending bits
Sleep(float(self.beacon_interval))
Self.dds.on()
Else:
Self.r.stop()           # stop sending bits
Def set_baud(self, b):
Self.baud = b
If self.baud == 2:
Self.spaced = 1
Self.sample_rate = 8000.0
Self.symbol_length = 4000.0
Elif self.baud == 4:
Self.spaced = 2
Self.sample_rate = 8000.0
Self.symbol_length = 2048.0
Elif self.baud == 5:
Self.spaced = 2
```

Self.sample_rate = 11025.0

Self.symbol_length = 2048.0

Elif self.baud == 8:

Self.spaced = 2

Self.sample_rate = 8000.0

Self.symbol_length = 1024.0

Elif self.baud == 11:

Self.spaced = 1

Self.sample_rate = 11025.0

Self.symbol_length = 1024.0

Elif self.baud == 16:

Self.spaced = 1

Self.sample_rate = 8000.0

Self.symbol_length = 512.0

Elif self.baud == 22:

Self.spaced = 1

Self.sample_rate = 11025.0

Self.symbol_length = 512.0

Elif self.baud == 44:

Self.spaced = 2

Self.sample_rate = 11025.0

Self.symbol_length = 256.0

Elif self.baud == 88:

Self.spaced = 1

Self.sample_rate = 11025.0

Self.symbol_length = 128.0

Self.r.set_baud(self.sample_rate / self.symbol_length)

Self.r.set_bit_length(1000 / (self.sample_rate / self.symbol_length))

Self.tone_spacing = self.sample_rate * self.spaced / self.symbol_length

Self.bandwidth = self.num_tones * self.tone_spacing

Def set_message(self, msg):

Self.r.set_message(chr(0) + "\r" + msg + "\r")

Self.dds.on()

Self.r.send_code()

Self.all_done = False

Print("Frequency:", self.frequency)

Print("Baud:", self.baud)

Print(”Beacon?:”, self.is_beacon)

Print(”Message:”, self.r.message)

Print()

Print(”Bandwidth:”, self.bandwidth)

Print(”Tone spacing:”, self.tone_spacing)

Print(”Symbol length:”, self.symbol_length)

Print(”Bit length:”, 1000 / (self.sample_rate / self.symbol_length))

Print(”Baud:”, self.sample_rate / self.symbol_length)

Def set_beacon(self, onoff, interval):

Self.is_beacon = onoff

Self.beacon_interval = interval

Much of it is simply setting up the various baud rates.

The dominoex.py module handles the translation between letters and tones, using the by-now-familiar generator we still call “bit” although it is once again sending symbols:

Class DOMINOEX:

Def __init__(self, send_tone, report_message_end=None):

Self.send_tone = send_tone

Self.report_message_end = report_message_end

Self.set_baud(2)   # DOMINOEX MICRO

Self.frequency = “7104000”

Self.call = “N0CALL”

Self.location = “CM87xe”

Self.bit_length = int(1000 / float(self.baud))

Self.timer = Timer()

Def set_call(self, call):

Self.call = call

Def set_baud(self, baud):

Self.baud = float(baud)

Def set_bit_length(self, len):

Self.bit_length = int(len)

Def set_frequency(self, frequency):

Self.frequency = frequency

Def set_location(self, location):

Self.location = location

Def set_message(self, message):

Self.message = message.format(self.call, self.location)

Def bit(self):

For letter in self.message:

Code = dominoex_varicode[ord(letter)]

Count = 0

For tone in code:

If tone & 0x8 or count == 0:

Yield tone

Count += 1

Self.report_message_end()

Def stop(self):

Self.timer.deinit()

Def send_code(self):

Self.gen = self.bit()

Self.timer.init(period=self.bit_length, mode=Timer.PERIODIC, callback=self.bit_finished)

Def send_bit(self, unused):

Try:

Tone = next(self.gen)

Except StopIteration as tone:

Return self.report_message_end()

Self.send_tone(tone)

Def bit_finished(self, unused):

Self.send_bit(True)

Our main.py module looks like this:

Def main():

Dex = DominoEXConfig(4, 7040000, “AB6NY”, “CM87xe”)

While True:

Dex.send_code()

While dex.all_done == False:

Sleep(5)

Main()

We set up the baud rate (4 in this case), and the frequency, call, and location. Then we set up the message, and start sending.

The radio.py and SI5351.py modules are the same as before.

I did notice when using a baud rate of 11 (and even 8) that the first few characters in the message would often be garbled due to FLDIGI trying to be too smart about adjusting signal levels. Since the call sign was what was getting garbled (not a good thing) I added a few disposable characters in the beginning:

The result looked like this:

Press enter or click to view image in full size

Image by the author

Press enter or click to view image in full size

Image by the author

At 44 baud, things got progressively worse, but it was so fast that it was easy to get the whole message by reading several lines:

Press enter or click to view image in full size

Image by the author

To computer users used to a baud being a bit instead of a symbol, 44 baud sounds slow. However, since each symbol is a character in DominoEX, this is 44 characters per second or 440 bits per second. Still pathetic compared to Wi-Fi speeds, but this is five dollars' worth of hardware capable of bouncing over the horizon. And as a chat mode, 280 words per minute is faster than I can type anyway.

Dominoex

Amateur Radio

Radio Hackers: Electromagnetic Eavesdropping & Harmonics

Investigator515

Investigator515

Follow

6 min read

Jan 13, 2025

Listen

Share

More

EMR eavesdropping uses receiving techniques to exploit secure information.

If you aren't a medium member, you can read with no paywall via substack

In earlier articles, we covered the concept of the Van EckPhreak, which discussed how electromagnetic radiation (EMR) could be used to exploit private information. One point that we missed in that article though, was how an entire research field would come from this.

EMR & Cybersecurity

However, it's not just protocols like that that are open to research. For some researchers, detecting and analysing stray emissions can often be a way to recover data or uncover lesser-known exploits

To best understand this subject though, we'll have to discuss something we've looked at in the publication before. The concept of harmonic frequencies, as well as where / how we can find them.

WTF Is a Harmonic?!

It's worth mentioning that many electronic systems (radio transmitters in particular) apply strict filtering to harmonic frequencies. This is because badly filtered transmissions pose a very real interference threat to those operating on adjacent frequencies.

Press enter or click to view image in full size

If you'd like to read more about spurious emissions from badly filtered transmitters, this Hackaday article does a great job of breaking it down.

What Kind Of Exploits?

Much of this earlier research came about as a result of the TEMPEST project, something we also touched on in the earlier article. This program was initially the domain of government actors however as technology progressed and became more prolific, more and more of this research started to enter the public domain. By the time the 90s rolled around, civilian researchers were also contributing both data and exploits to attack and protect various electronic devices.

Despite this, we still see much in the way of research into these fields at events like Defcon, that are implemented to encourage free thinking and collaborative working. This alone is pretty interesting to follow.

How Can I Learn?

Press enter or click to view image in full size

Lastly, understanding how to capture, identify and decode different signal types is also helpful. This means you'll be better equipped to understand different transmissions that you might see on your SDR display as well as having a clear understanding of both harmonic and fundamental frequencies.

To operate within the radio spectrum ideally, we'll need to understand it properly first.

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.

Enjoyed this article? Join the community!

Follow us on Twitter and

■ We're now on Bluesky!

Articles we think you'll like:

What The Tech?! Rocket Engines

OSINT Investigators Guide to Self Care & Resilience

Surveillance

Radio

Information Security

Technology

Radio Hackers: Jamming & Spoofing Fundamentals

Follow

7 min read

Jul 9, 2024

Listen

Share

More

Understanding Spoofing and Jamming is an essential part of your offensive toolkit.

If you aren't a medium member, you can read with no paywall via substack

However, in Signals Intelligence, we're going to find ourselves dealing with large amounts of data, and in the world of intelligence, we don't automatically trust everything we discover without attempting to validate it first.

So, during the process of conducting Signals Intelligence (SIGINT), this means that determining the authenticity of a signal is an important part of the assessment process. Today, we'll be exploring some of the fundamentals behind signal spoofing and jamming and looking at why we need to know how to identify malicious signals. There's also a real-life example of a spoofed signal at work.

What Is Spoofing And Jamming?

It's also worth mentioning that these attacks can often be personalized or discriminate, where they are targeting a specific person or device of interest. We'll mostly see that at a nation-state level. Or, they could be indiscriminate, affecting everyone within the transmitting jammers range. The AN/ALQ-99 jamming pod on an electronic warfare aircraft is a good example of something that can work indiscriminately, providing general area effects.

Press enter or click to view image in full size

How Does It Work

It's pretty unlikely you'll have to deal with the consequences of an EA-18 Growler overhead anytime soon but it's entirely plausible that you'll see civilian examples of SIGINT and EW, particularly if you're working in sensitive or government-focused installations.

Press enter or click to view image in full size

Military Spec jammers are wide-band with high output power. In the civilian world, jammers are often single-use and lower-powered. Source: Wikipedia.

Does This Even Happen?

If you're not great with converting frequency to wavelengths, this is a weak signal right in the middle of the 1ghz band.

Press enter or click to view image in full size

GPS Signals are regularly jammed and spoofed in certain parts of the world. Source: Wikipedia.

Press enter or click to view image in full size

GPSJam lets us see jamming attacks in real time. Source: gpsjam.org

logs interference to the GPS system in a hot spot fashion, using colour variations to determine the active level of interference. Looking at the attached image we can see that nearly every spot that has an active conflict shows some form of interference with GPS. We can also see that there are high levels of interference around certain parts of the Russian Federation, namely Moscow, or more specifically the Kremlin.

We should also note that many countries that are reporting interference are not active participants in said conflict however the consequences of jamming the satellites still have an effect regardless of this.

Press enter or click to view image in full size

Finland has suffered the effects of EW attacks on GPS pretty consistently of late. Source: GPSjam.org

Press enter or click to view image in full size

In Closing.

If you'd like some more resources on the topic of spoofing and jamming attacks in the real world to help increase your knowledge level, and background then here are a few links to get you started.

While this is the first piece we've written that covers spoofing and jamming theory, we'll be exploring this a lot more in future Radio Hackers articles. We'll also be reviewing more simulated data in a controlled, lab environment using low-powered software-defined radio systems, to help further your understanding of how to detect and defend against these types of attacks.

If you haven't got a transmit-capable SDR to go with your Radio Hackers tutorials, then now might just be the time to splash out and buy one.

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.

■ Enjoyed this article? Join the community! ■

■ Follow us on

■ Articles we think you'll like:

Software Defined Radio &

OSINT Investigators Guide to

✉■ Want more content like this?

Hacking

Radio

Software Defined Radio

Learning

Python Radio 16: MFSK Mode

Follow

20 min read

Sep 3, 2024

Listen

Share

More

Bigger and faster…

Press enter or click to view image in full size

Photo by the author

In the last article, we hinted at the Si5351 module that will allow us to send on any frequency we like, with a resolution of less than one Hertz.

In this article, we get to use it.

The module uses the I2C interface, so all it needs are five-volt power, ground, a clock signal, and a data signal. The ESP32 and ESP8266 can also use this module, as they both support the I2C interface.

Like those microprocessors, the SI5351 also produces square waves by dividing a clock by integers, but it has much better resolution. Below 18 megahertz, it will never be more than half a hertz off of the target frequency. Even at 150 MHz the error is only 4 hertz, and 18% of the frequencies in that range don't miss at all. The module has a nominal range of 8 kilohertz to 160 megahertz, but it can actually reach the 1.25-meter band up around 220 megahertz.

With the SI5351, we can transmit in modes that send multiple tones only a few hertz apart, such as MFSK.

Our radio.py module handles the details of using the SI5351 to output square waves at the frequencies we choose. The SI5351 has two clocks, called A and B in the spec sheet, but we will refer to them as 0 and 1. For the most part, we will only deal with clock 0. Likewise, it has three outputs, but we will only use the first one, labeled CLK0.

Here is the code:

Most of the code is in the set_freq() method, and half of that is a comment.

The __init__() method handles setting up the I2C interface. We use the software version of I2C, because it is the same on all the microprocessors, so we don't have to special case any code (other than which pins to use).

The characters are encoded in what is called "varicode", where the more frequently used characters can be sent using fewer bits. Letters like "e" and "t" use only four bits. Characters such as "%" and "&" use ten bits.

Here is the whole table, in a module called mfsk_varicode.py:

The code looks like this:

Finally, we get to our tiny little main.py module:

It sets up pin 15 as the output, 7040000 as the frequency, and adds the call sign and Maidenhead Locator code to the message. Then it loops, sending the message over and over again, so that we can tune it in on the receiver.

The free software Fldigi program can receive and decode MFSK signals from your RTL-SDR. This makes testing much easier.

Mfsk

Python Programming

Python Radio 43: Super High Frequency Radar

Simon Quellen Field

Simon Quellen Field

Follow

5 min read

Jun 16, 2025

Listen

Share

More

24 Gigahertz radar in the 1.25 centimeter band

Press enter or click to view image in full size

24 GHz Radar

All photos by the author.

The simplest version, shown in the photo above, is very easy to use. It sends the range in centimeters via serial at 115200 baud.

```
Class Display:
Def __init__(self):
Self.i2c_display = I2C(0, sda=Pin(5), scl=Pin(6), freq=400_000)
Self.display = SH1106_I2C(128, 64, self.i2c_display, rotate=180)
Self.display.contrast(255)
BufferWidth, BufferHeight = 128, 64
ScreenWidth, ScreenHeight = 72, 40
Self.xOffset, self.yOffset = (BufferWidth – ScreenWidth) // 2, (BufferHeight – ScreenHeight) // 2
Def oled(self, s, line, column):
Self.display.fill_rect(self.xOffset, self.yOffset + 2 + line * 9, 128-self.xOffset, 9, 0)
Self.display.text(s, self.xOffset + 2 + column * 8, self.yOffset + 2 + line * 9, 1)
Self.display.show()
D = None
Try:
D = Display()
Except OSError as e:
If e == 19: # ENODEV
D = None
Uart = UART(1, baudrate=115200, tx=1, rx=2)
Def main():
If d:
d.oled(“Range:”, 0, 0)
d.oled(“cm”, 2, 0)
cm = “”
sleep_ms(1)
if uart.any():
line = uart.readline()
if line:
try:
cm = line.decode(“utf-8”)
except Exception as e:
print_exception(e)
print(“Line:”, line)
if “Range” in cm:
```

cm = cm[6:].strip()

print(cm)

if d:

d.oled(cm, 1, 0)

main()

The first half of the code handles the display, which we first introduced in Chapter 35 when we built the mailbox alarm.

When I took the picture, I was 9 centimeters from the device.

The simple device has one transmit antenna and one receive antenna. But the HLK-LD2450 device also comes in a package with two receiving antennas. That allows it to tell us the distance (the Y direction) and how far to the left or right the target is (the X direction).

Press enter or click to view image in full size

Radar with two receiving antennas.

In the photo, you can see a USB-to-Serial module attached to the HLK-LD2450. We are only using one serial line, as we won’t be sending commands to the radar, only receiving data. The TX pin connects to the RX pin on the serial adapter. 3.3-volt power and ground complete the connections.

This device has a more complex data stream than the first one. It sends the bytes raw (not ASCII), in frames that have a header and end codes, and require a little decoding.

Import matplotlib.pyplot as plt

Uart = Serial(“com7”, 256000)

Def convert_sign(x):

If x & 0x8000:

X = x – 0x8000

Else:

X = -x

Return x

Def main():

Fig = plt.figure(figsize=(7, 7))

Plt.ion()

Plt.show()

While True:

Sleep(0.001)

Line = uart.read(30)

If line and len(line) >= 29:

Header = int.from_bytes(line[:4], ‘little’)

A_x = convert_sign(int.from_bytes(line[4:6], ‘little’))

A_y = convert_sign(int.from_bytes(line[6:8], ‘little’))

A_speed = convert_sign(int.from_bytes(line[8:10], ‘little’))

```
A_resolution = int.from_bytes(line[10:12], ‘little’)
B_x = convert_sign(int.from_bytes(line[12:14], ‘little’))
B_y = convert_sign(int.from_bytes(line[14:16], ‘little’))
B_speed = convert_sign(int.from_bytes(line[16:18], ‘little’))
B_resolution = int.from_bytes(line[18:20], ‘little’)
C_x = convert_sign(int.from_bytes(line[20:22], ‘little’))
C_y = convert_sign(int.from_bytes(line[22:24], ‘little’))
C_speed = convert_sign(int.from_bytes(line[24:26], ‘little’))
C_resolution = int.from_bytes(line[26:28], ‘little’)
End_frame = int.from_bytes(line[28:], ‘little’)
Plt.clf()
Fig.text((500+a_x)/1000, a_y/5000, f”o”)
Fig.text(0, 0, f”{a_x} mm, {a_y} mm, {a_speed} cm/sec”)
Fig.canvas.draw()
Plt.pause(.02)
Plt.show()
Print(f”X: {a_x} mm”, end=” “)
Print(f”Y: {a_y} mm”, end=” “)
Print(f”Speed: {a_speed} cm/sec”, end=” “)
Print(“\r”, end=””)
Main()
```

The manual gives an example to make decoding easier. Here is a sample frame:

AA FF 03 00 0E 03 B1 86 10 00 40 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 CC

The first four bytes AA FF 03 00 are the header. Every frame starts with this.

The last two bytes 55 CC are the end-of-frame marker. Every frame ends with this.

The sign bit is backwards from what we see in normal two’s-complement arithmetic, with a 0 meaning negative and a 1 meaning positive. We handle this unusual format with a convert_sign() subroutine. The resolution (weirdly) does not use the sign bit convention.

Because we are running on a computer with a screen, we can plot the result using Matplotlib in Python. We set interactive mode (plt.ion()), adjust the coordinates to screen coordinates, and draw a little “o” to represent the person being tracked.

Here is what it looks like:

The 24 GHz Radar screen in action.

The distance resolution claims about one foot (32 centimeters), but the data from the device shows single-centimeter resolution, which I take with about 320 grains of salt. But averaging a few hundred readings might get us a pretty accurate distance, perhaps with centimeter resolution.

The range of the device is about 8 meters (about 25 feet). It is most sensitive to human bodies, perhaps by ignoring things that don’t move. It might even be possible to use it to detect a heartbeat with a little digital signal processing.

Python Radio 19: Thor Mode

Follow

7 min read

Sep 6, 2024

Listen

Share

More

The best of both worlds…

Press enter or click to view image in full size

Image by the author

What if we combined the forward error correction of MFSK with the robustness and ease of tuning of FSQ and DominoEX?

We would get Thor.

The screenshot above shows Thor11, the 11 baud mode that gave us trouble in DominoEX.

The random characters in between the lines of perfect copy are noise from the radio during the periods when we weren't sending.

Thor uses the same varicode as MFSK. It also has a shorter Thor varicode that is used to send callsigns when the system is idling. That looks like this:

The thor_config.py module looks familiar by now:

As with DominoEX, most of the code is just setting up the different baud rates.

The real work is done in the thor.py module:

A lot of the code is the NASA 7 forward error correction code borrowed from MFSK. Then we have our bit() generator and our send_code() method.

The send_code() method is a little bit larger than we are used to, since it has to worry about flushing the receiver's decoder before sending real bits, and then sending the idle preamble.

Amateur Radio

Python Programming

Python Radio 17: FSQ Mode

Simon Quellen Field

Simon Quellen Field

Follow

7 min read

Sep 4, 2024

Listen

Share

More

Fast Simple QSO

Press enter or click to view image in full size

Image by author

Instead of fixed tones, FSQ mode uses the difference from the last tone. This makes it much easier to tune and makes it virtually immune to frequency drift. It also makes it less affected by inter-symbol interference that can occur in Near Vertical Incidence Skywave (NVIS) communication, where people can converse with other nearby radio operators over the horizon by bouncing the signal straight up off of the ionosphere.

It can send 104 ASCII characters, and takes up less than 300 hertz of bandwidth, helping it to have a good signal-to-noise ratio.

People using FSQ mode hang out at 3,588,000 hertz in the 80-meter band, 7,044,000 in the 40-meter band, and 10,144,000 in the 30-meter band.

In FLDIGI, the reception looks like this:

Press enter or click to view image in full size

Image by author

You can see the 33 tones scattered across the waterfall display in the lower left and the decoded text above it.

Here is the fsq_varicode.py module:

```
Fsq_varicode = {

‘a’:  ( 1,  0),

‘b’:  ( 2,  0),

‘c’:  ( 3,  0),

‘d’:  ( 4,  0),

‘e’:  ( 5,  0),

‘f’:  ( 6,  0),

‘g’:  ( 7,  0),

‘h’:  ( 8,  0),

‘i’:  ( 9,  0),

‘j’:  (10,  0),

‘k’:  (11,  0),

‘l’:  (12,  0),

‘m’:  (13,  0),

‘n’:  (14,  0),

‘o’:  (15,  0),

‘p’:  (16,  0),

‘q’:  (17,  0),

‘r’:  (18,  0),

‘s’:  (19,  0),

‘t’:  (20,  0),

‘u’:  (21,  0),
```

‘v’: (22, 0),

‘w’: (23, 0),

‘x’: (24, 0),

‘y’: (25, 0),

‘z’: (26, 0),

‘A’: (1, 29),

‘B’: (2, 29),

‘C’: (3, 29),

‘D’: (4, 29),

‘E’: (5, 29),

‘F’: (6, 29),

‘G’: (7, 29),

‘H’: (8, 29),

‘I’: (9, 29),

‘J’: (10, 29),

‘K’: (11, 29),

‘L’: (12, 29),

‘M’: (13, 29),

‘N’: (14, 29),

‘O’: (15, 29),

‘P’: (16, 29),

‘Q’: (17, 29),

‘R’: (18, 29),

‘S’: (19, 29),

‘T’: (20, 29),

‘U’: (21, 29),

‘V’: (22, 29),

‘W’: (23, 29),

‘X’: (24, 29),

‘Y’: (25, 29),

‘Z’: (26, 29),

0: (28, 30), # IDLE

‘\u00B1’: (10, 31), # plus/minus

‘\u00F7’: (11, 31), # division sign

‘\u00B0’: (12, 31), # degrees sign

‘\u00D7’: (13, 31), # multiply sign

‘\u00A3’: (14, 31), # pound sterling sign

```
'\b':     (27, 31), # BS

'\u007F': (28, 31), # DEL
```

As with our other modes, we have an fsq_config.py module to isolate details of the implementation from other parts of the program:

```
Class FSQConfig:

Def __init__(self, frq, baud, call, location):

Self.frequency = frq

Self.baud = baud

Self.mycall = call

Self.location = location

Self.r = FSQ(self.send_tone, self.baud, self.all_done)

Self.dds = Radio()

Self.dds.send()

Self.is_beacon = False

Self.message = ''

Self.spacing = 8.7890625                # 8.7890625 Hz

Self.usb_offset = 1350.0

Self.is_directed = False

Self.tocall = "N0CALL"

Self.beacon_interval = 60.0

Self.incremental_tone = 0.0

Self.r.set_frequency(self.frequency)

Self.r.set_call(self.mycall)

Self.r.set_location(self.location)

Self.r.set_call(self.mycall)

Self.r.set_location(self.location)

Print("Frequency:", self.frequency)

Print("Baud:", self.baud)

Print("Beacon?:", self.is_beacon)

Print("Directed?:", self.is_directed)

Print("To callsign:", self.tocall)

Def get_radio():

Return dss

Def set_message(self, msg):

Self.message = msg.format(self.mycall, self.location)

Self.all_done = False

If self.is_beacon:
```

```
Else:
Def send_code(self):
Self.dds.on()
Sleep(.1)
Self.r.send_code()
Def send_tone(self, tone):
Self.incremental_tone = (self.incremental_tone + float(tone) + 1.0) % 33
Self.f = int(int(self.frequency) + self.usb_offset + self.incremental_tone * self.spacing)
Self.dds.set_freq(0, self.f)
Self.dds.send()
Def all_done(self):
If self.is_beacon:
Self.r.stop()           # stop sending bits
Self.dds.off()
Sleep(float(self.beacon_interval))
Self.dds.on()
Else:
Self.r.stop()           # stop sending bits
Self.dds.off()
Self.all_done = True
Def crc(self, text):
Self.table = []
For x in range(256):
Byte_val = x
For y in range(8):
If byte_val & 0x80:
Byte_val = (byte_val * 2) ^ 7
Else:
Byte_val = (byte_val * 2) ^ 0
Self.table.append(byte_val & 0xFF)
Val = 0
For ch in text:
Val = self.table[val ^ ord(ch)] & 0xFF
Return "%0.2X" % (val & 0xFF)
```

The code basically handles setting up the frequency, baud rate, call, and location, and handles formatting the message. The first part of any FSQ transmission is the call sign and a 2-character Cyclic Redundancy Check to ensure that the call sign was properly received.

The fsq.py module is much simpler than the MFSK module was, since it has much less to do:

```
Class FSQ:

Def __init__(self, baud, send_tone, report_message_end=None):

Self.send_tone = send_tone

Self.report_message_end = report_message_end

Self.set_baud(baud)

Self.frequency = “7104000”

Self.call = “N0CALL”

Self.location = “CM87xe”

Self.baud = baud

Self.bit_length = int(1000 / float(baud))

Self.timer = Timer()

Self.all_done = False

Def set_call(self, call):

Self.call = call

Def set_baud(self, baud):

Self.baud = baud

Self.bit_length = int(1000 / float(self.baud))

Def set_frequency(self, frequency):

Self.frequency = frequency

Def set_location(self, location):

Self.location = location

Def set_message(self, message):

Self.message = message.format(self.call, self.location)

Def bit(self):

For letter in self.message:

Code = fsq_varicode.get(letter)

If not code:

Code = fsq_varicode.get(” “)                # Make illegal characters send as spaces

Count = 0

For tone in code:

If tone > 0 or count == 0:

Yield tone

Count += 1

Self.all_done = True

Def stop(self):

Self.timer.deinit()
```

Self.all_done = True

Def send_code(self):

Self.all_done = False

Self.gen = self.bit()

Self.timer.init(period=self.bit_length, mode=Timer.PERIODIC, callback=self.bit_finished)

Def send_bit(self, unused):

Try:

Tone = next(self.gen)

Except StopIteration as tone:

Self.all_done = True

Self.stop()

Self.report_message_end()

Return

Self.send_tone(tone)

Def bit_finished(self, unused):

Self.send_bit(True)

We have a generator to give us each symbol (we call it a ‘bit’ as we did in the RTTY and MFSK modules, but it is actually a symbol since it is a tone that maps onto one of 32 characters).

We set up the timer to send the bits at the right rate, and the timer callback calls bit_finished() which simply calls send_bit().

The main.py module is even simpler:

Def main():

Fsq = FSQConfig(7040000, 12, “AB6NY”, “CM87xe”)

While True:

Fsq.send_code()

While fsq.all_done == False:

Sleep(5)

Main()

Fsq Mode

Python Programming

Further Stuff

Python Radio 7: The ESP32

Simon Quellen Field

Simon Quellen Field

Follow

3 min read

Aug 24, 2024

Listen

Share

More

More power!

Press enter or click to view image in full size

Image: Lilygo

The ESP32 is the big brother to the ESP8266.

Because it has several accessible hardware timers and UARTs (serial ports), it makes several radio projects possible that would be difficult to do with the smaller chip.

ASCII text by radio

We will use two ESP32 boards. The board we will call Ichabod will transmit messages to the board we will call Rumpelstiltskin.

TX = 21

RX = 37

Def main():

Uart = UART(1, baudrate=2400, tx=TX, rx=RX)

If name == “Ichabod”:

Count = 0

While True:

Msg = “UUU “ + str(count) + “: Hello, world!\r\n”

Print(”Sending”, msg, end=”)

Uart.write(msg)

Count += 1

While True:

Try:

M = uart.read()

If m:

Print(m.decode(’utf-8’), end=”)

Except UnicodeError:

Pass

Main()

Name = “Rumpelstiltskin”

Before loading it onto one of the boards. Then I changed the line to

Name = “Ichabod”

And loaded it onto the other board. Now, as I make changes to main.py (such as changing the baud rate), I can just load the same file onto both boards.

The circuit is simple:

Press enter or click to view image in full size

Image by author

The receiver circuit looks like this:

Press enter or click to view image in full size

Image by author

In later projects, we will be using the power of the ESP32 to do a lot more work. Some radio modes require a good deal of computing power.

It is also nice to have a display to print things to.

Esp32

Micropython

Python Programming

Radio Communications

Texting

Python Radio 46: Satellites and Atomic Time

Simon Quellen Field

Simon Quellen Field

Follow

5 min read

Jun 30, 2025

Listen

Share

More

A GPS Disciplined NTP Server in Micropython

Press enter or click to view image in full size

A GPS satellite over North America.

MidJourney

For some weak signal protocols, such as WSPR, we need a clock that keeps good time. Anyone wishing to receive our signal knows to start receiving at an exact time, and if our clock doesn’t match theirs, we can’t communicate.

When we have an Internet connection, this is easy. We just use an NTP server to get accurate time from the web, and set our computer’s clock by that.

But if we are camping out in the wilderness, we will need something else.

The T-Beam has a GPS receiver built in. It also has Wi-Fi. And becoming an NTP server is surprisingly easy.

Here is the code:

```
Freq(240_000_000)

Last_rtc_sync_time = 0

RTC_SYNC_INTERVAL_SECONDS = 6 * 3600

NTP_PORT = 123

NTP_DELTA = 3155673600

Def setup_wifi_ap():

WIFI_SSID, WIFI_PASS = “T-Beam Time Server”, “gps-time-rocks”

Ap = WLAN(AP_IF)

Ap.active(True)

Ap.config(essid=WIFI_SSID, password=WIFI_PASS)

While not ap.active():

Sleep(1)

Print(f”--- Wi-Fi Access Point Started: SSID={WIFI_SSID}, IP={ap.ifconfig()[0]} ---“)

Def calculate_weekday(y, m, d):

If m < 3:

M += 12

Y -= 1

T = [0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4]

Day = (y + y // 4 – y // 100 + y // 400 + t[m – 1] + d) % 7

Return day if day != 0 else 7

Def set_rtc_from_gps(gps_object):

Global last_rtc_sync_time

Is_initial_sync = (last_rtc_sync_time == 0)

Time_since_last_sync = time() – last_rtc_sync_time

Should_sync = is_initial_sync or (time_since_last_sync > RTC_SYNC_INTERVAL_SECONDS)

If should_sync and gps_object.date[0] != 0:

Y, m, d = gps_object.date[2] + 2000, gps_object.date[1], gps_object.date[0]

H, mn, s = gps_object.timestamp[0], gps_object.timestamp[1], int(gps_object.timestamp[2])

Dt = (y, m, d, calculate_weekday(y, m, d), h, mn, s, int((gps_object.timestamp[2] – s) * 1e6))

RTC().datetime(dt)

Last_rtc_sync_time = time()

Msg = “Initial RTC sync” if is_initial_sync else “RTC re-synchronized”

Print(f”\nINFO: {msg}. Current Time: {y}-{m:02d}-{d:02d} {h:02d}:{mn:02d}:{s:02d} UTC\n”)

Def main():
```

Further Stuff

GPS_RX=34

GPS_TX=12

Setup_wifi_ap()

Uart = UART(2, baudrate=9600, tx=GPS_TX, rx=GPS_RX, timeout=10)

My_gps = MicropyGPS(location_formatting=’dd’)

Ntp_socket = socket(AF_INET, SOCK_DGRAM)

Ntp_socket.bind((‘’, NTP_PORT))

Ntp_socket.setblocking(False)

Cnt = 0

Lat_avg = 0

Lon_avg = 0

Alt_avg = 0

While True:

If uart.any():

If uart_data := uart.read():

For char_byte in uart_data:

My_gps.update(chr(char_byte))

If my_gps.satellites_in_use > 0 and my_gps.latitude[0] != 0.0:

Set_rtc_from_gps(my_gps)

Lat_data, lon_data = my_gps.latitude, my_gps.longitude

Lat_val, lon_val = lat_data[0], lon_data[0]

If lat_data[1] == ‘S’:

Lat_val = -lat_val

If lon_data[1] == ‘W’:

Lon_val = -lon_val

Alt_val, sats = my_gps.altitude, my_gps.satellites_in_use

If lat_avg == 0: lat_avg = lat_val

If lon_avg == 0: lon_avg = lon_val

If alt_avg == 0: alt_avg = alt_val

If cnt < 20:

Cnt += 1

Lat_avg = (lat_avg * (cnt-1) + lat_val) / cnt

Lon_avg = (lon_avg * (cnt-1) + lon_val) / cnt

Alt_avg = (alt_avg * (cnt-1) + alt_val) / cnt

Feet = alt_avg * 3.280839895

Timestamp = f”{my_gps.timestamp[0]:02d}:{my_gps.timestamp[1]:02d}:{my_gps.timestamp[2]:02.0f}”

Payload_str = f”{lat_avg:.9f}, {lon_avg:.9f}, {alt_avg:.5f} meters ({feet:.5f} feet), UTC:

{timestamp}, {sats:3d} satellites”

Print(f”{payload_str}”)

Sleep_ms(300)

Else:

Sleep(3)

If last_rtc_sync_time != 0:

Try:

Data, addr = ntp_socket.recvfrom(48)

If data:

Print(f”NTP Request from {addr[0]}”)

Recv_timestamp = time() + NTP_DELTA

Ntp_response = bytearray(48)

Ntp_response[0] = 0x24

Ntp_response[24:32] = data[40:48]

Secs, frac = int(recv_timestamp), int((recv_timestamp % 1) * (2**32))

Pack_into(‘!II’, ntp_response, 32, secs, frac)

Pack_into(‘!II’, ntp_response, 40, secs, frac)

Ntp_socket.sendto(ntp_response, addr)

Except OSError as e:

If e.args[0] != 11:

Print(f”NTP Socket Error: {e}”)

Main()

Starting in main(), we tell the program where to find the GPS receive and transmit pins. The GPS chip on the T-Beam connects to the UART at 9600 baud.

Then we bind a socket to the NTP port 123 and set it to non-blocking. This is how we will talk to NTP clients.

GPS latitude, longitude, and especially altitude are not perfectly accurate. We average the last 20 readings to get more precision.

Then we print out the data.

If we have synced our clock with the GPS satellite’s atomic clock, we are ready to serve NTP clients.

We get the request (if there is one) from the socket, and prepare our reply.

The setup_wifi_ap() routine is very simple, mostly boilerplate. You can set the SSID and password to suit your taste.

The micropyGPS.py module comes from GitHub: . Download it and copy it to the T-Beam along with main.py.

When you boot the T-Beam, it starts printing to the Putty window (the terminal emulator I use on Windows):

37.203254700, -122.008956909, 487.72998 meters (1600.16394 feet), UTC: 22:54:26, 10 satellites

37.203254700, -122.008956909, 487.67343 meters (1599.97852 feet), UTC: 22:54:27, 10 satellites

37.203254700, -122.008956909, 487.61972 meters (1599.80225 feet), UTC: 22:54:27, 10 satellites

37.203254700, -122.008956909, 487.56870 meters (1599.63489 feet), UTC: 22:54:27, 10 satellites

37.203254700, -122.008956909, 487.51031 meters (1599.44336 feet), UTC: 22:54:28, 10 satellites

37.203254700, -122.008956909, 487.45483 meters (1599.26123 feet), UTC: 22:54:28, 10 satellites

NTP Request from 192.168.4.2

37.203254700, -122.008956909, 487.40210 meters (1599.08826 feet), UTC: 22:54:28, 10 satellites

37.203254700, -122.008956909, 487.34198 meters (1598.89099 feet), UTC: 22:54:29, 10 satellites

37.203254700, -122.008956909, 487.28491 meters (1598.70374 feet), UTC: 22:54:29, 10 satellites

37.203254700, -122.008956909, 487.23065 meters (1598.52576 feet), UTC: 22:54:29, 10 satellites

37.203254700, -122.008956909, 487.17914 meters (1598.35681 feet), UTC: 22:54:30, 10 satellites

NTP Request from 192.168.4.2

37.203254700, -122.008956909, 487.13019 meters (1598.19617 feet), UTC: 22:54:30, 10 satellites

37.203254700, -122.008956909, 487.08368 meters (1598.04358 feet), UTC: 22:54:30, 10 satellites

37.203254700, -122.008956909, 487.03949 meters (1597.89856 feet), UTC: 22:54:31, 10 satellites

37.203254700, -122.008956909, 486.99750 meters (1597.76086 feet), UTC: 22:54:31, 10 satellites

37.203254700, -122.008956909, 486.95761 meters (1597.63000 feet), UTC: 22:54:31, 10 satellites

37.203254700, -122.008956909, 486.90973 meters (1597.47290 feet), UTC: 22:54:32, 10 satellites

UTC does not have daylight saving time, and reflects the time at the zero degree meridian, as it replaces the older Greenwich Mean Time (GMT).

In Windows, you use the command net start w32time to start the NTP client on your machine. Then you type:

W32tm /config /manualpeerlist:"192.168.4.1" /syncfromflags:manual /reliable:yes /update

The T-Beam sees the request and sends the reply, and the NTP client on the laptop updates its clock.

Now my laptop computer can be out in the woods and still get the correct time when the UTC system decides we need another leap second.

Space Weather: Imagery via Elektro-L3

Investigator515

Investigator515

Follow

6 min read

Aug 25, 2025

Listen

Share

More

The Geostationary Elektro satellites offer a far greater challenge than the POES satellites.

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

In today's article, we'll be taking a look at the Russian-designed Elektro satellite program and seeing what, if anything, it has to offer amateur satellite enthusiasts. Let's go take a look!

Press enter or click to view image in full size

The Elektro series would be Russia's first foray into deep-space Geostationary satellites after the fall of the Soviet Union. Here's an image of Elektro-L in the clean room before launch. Source: Wikipedia.

Background

With Sputnik, the first of many Soviet satellites to enter orbit during the height of the space race, you didn't have to be a genius to realise that this would change the dynamic forever.

As such, the USSR would attempt to capitalise on this immediately, and due to this, all manner of satellites would be launched. Of these, many would be considered to be dual use, in the sense that they would carry both military and civilian payloads.

Press enter or click to view image in full size

The Elektro's L-band data link provides a far greater challenge than the VHF downlink of the earlier Meteor-M series. Source: Wikipedia.

The Elektro Series

As technology progressed, we'd see many of these deep-space assets become more complex, but it would be the late 90s and early 2000s shrinking of electronic components that would offer true revolution in modern spacecraft operations and the Elektro series would be one of the first to leverage many of these new technologies.

So, with this said, the Elektro series would be born, and as such, the first Elektro L-1 satellite would fly in late 1994 as the first Russian-designed geostationary spacecraft since the collapse of the Soviet Union just a few years prior. As you'd imagine, this would be hugely noteworthy considering the political climate at the time.

While the program would start with just one satellite (Elektro L-1), the program would eventually evolve to include multiple satellites that would be positioned to give near-global, real-time coverage, a huge boost in service in comparison to assets that were available earlier.

Press enter or click to view image in full size

Amateur Image Captures

While the amateur radio community would make plenty of noise about placing a Geostationary transponder system in orbit, thanks to satellites like the now-defunct AO-40, the reality was that placing niche, small-use systems into orbit was becoming harder as educational launch opportunities would begin to dry up. While several projects would eventually fly on both Russian and European systems, the writing would be on the wall regarding modern iterations of these types of systems, meaning that geostationary hams had little to attract their attention.

Coming Soon

With little more than some good software, a capable antenna and in some instances a preamp, most users are away, capturing high-quality images of Earth from space.

It's fair to say that geo-stationary satellites have little in common with these simpler VHF systems. In fact, even experienced amateurs might find the Elektro satellites to be more of a challenge than expected. The sometimes fickle nature of propagation in the 1600MHz band means that, sometimes, establishing a system that works reliably can be an exercise in frustration.

Further Stuff

If you’ve ever had an interest in amateur meteorology or satellite communications, then stick around. Now is your time to shine.

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.

Enjoyed this article? Join the community!

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

Space

Radio

Weather

Software Defined Radio

Technology

Software Defined Radio & Radio Hacking: Space Communications (Part 3)

Investigator515

Investigator515

Follow

9 min read

Feb 6, 2024

Listen

Share

More

It’s time to spy on things that orbit the earth

This is multi part series. To catch up, read Part 1 and Part 2. Or find everything in our publication Radio Hackers.

As we’ve explored the world of Software Defined Radio, we started to explain some fundamentals around transmissions, antennas and receiving systems. It’s time to start putting these new pieces of knowledge to the test by looking at practical ways we can further develop these skills. In earlier articles, we promised a primer on space communications. Today, we deliver! But first, a quick update.

This series was far more popular than anticipated. Which we love, because it means there are plenty of inquisitive minds out there. Because of this, we’ve extended the series so we can focus on more in-depth tutorials around SDR over a longer time frame.

We’ve also launched a new publication where all these articles are posted, so you can streamline updates. Most of all, we’d love to hear about your journey. So if you’re using SDR we’d encourage you to share your journey by doing a write up and adding it to our publication. Contact us direct or drop a comment to arrange this.

The Status:

While exploring space-based assets can be pretty intriguing, it's fair to say that if you're a beginner to this world, the array of information to take in can be a little overwhelming. There's frequency management, research of what to look at and where, as well as many other factors to be successful in your quest.

Today, we'll look at intercepting communications from the International Space Station. To do this, we'll have to look at ways to obtain orbital information, find out what frequencies and mode our target is operating with and put all that together to hopefully intercept either digital or voice communications from our space-based platform. Time to plan!

Planning the Event:

To successfully achieve our goal today there are a few critical pieces of information we'll need. We need to know where our target is (location), when it will be there (timing), how we are to listen in on it (transmission mode) and where we are to listen to it (transmission frequency).

Orbital Information:

If we wish to do this on a mobile device, our recommendation it to use a mobile app like ISS detector. It's available on both Apple and Android devices and will give you real-time orbital information as well as advanced future orbits a few days ahead of time. Find ISSDetector on the Play Store via this link.

If you're a Linux user, you can install it with the APT package manager with this command

Apt install gpredict

Press enter or click to view image in full size

Press enter or click to view image in full size

There's a vast communications suite onboard but in the interests of streamlining things, we'll focus on transmissions from the onboard Automatic Packet Reporting System (APRS) as we can then decode them later on using a plugin.

Transmission Types & Frequencies:

To achieve our last two requirements we need to know frequency information and type and we can obtain this from Gpredict as well.

Press enter or click to view image in full size

Frequency information is easy to find in your tracker of choice. Source: Author

Press enter or click to view image in full size

We can see in our attached image that the APRS transmitter has a downlink of 145.825mhz, the middle of the VHF amateur band and uses FM modulation. We also note that there's a baud rate there, as it's a digital transmission. We'll focus on decoding that later on, but this data gives us our transmission frequency and type.

One Last Caveat:

One last item to check before you attempt your pass is the current status of the transmitter. Often, the system will be turned off during docking operations or maintenance periods, meaning that despite being overhead no transmissions will be heard. Mitigate this by checking the status of the station via this website. You can also find information via various social media accounts.

Press enter or click to view image in full size

Further Stuff

During the Pass:

It’s good practice to have your station set up a few minutes ahead of the pass to ensure any problems are uncovered prior to the pass commencing.

Short, sharp and effective. The transmission is strong and clear in the waterfall display. Source: Author

Doppler: One Quick Word

Press enter or click to view image in full size

Doppler shift varies according to frequency, with higher shifts noticeable as frequency increases. Source: Wikipedia

In Closing:

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.

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

Support me on Ko-fi: Investigator515

Follow us on Twitter

Articles we think you’ll like:

What the Tech?! GPS Technology

Space

Osint

Hacking

Information Security

Radio

Python Radio 20: The CC1101 Module

Follow

12 min read

Sep 7, 2024

Listen

Share

More

Half a megabit per second over a kilometer.

Press enter or click to view image in full size

Photo by the author

The CC1101 is a very flexible sub-gigahertz transceiver. It can transmit and receive in three wide frequency ranges: 300 to 348 MHz, 387 to 464 MHz, and 779 to 928 Mhz. That middle range includes the European license-free ISM band (433.05 MHz to 434.79 MHz), as well as the U.S. Amateur Radio 70 cm band (420 to 450 MHz). That means that with an Amateur Radio license, you can amplify the CC1101’s 10-milliwatt output to as much as 50 watts (but as most communication in this band is line-of-sight, 5 watts is usually more than enough).

The last band includes the European 868 MHz license-free ISM band (863 MHz to 870 MHz) and U.S.

915 Mhz license-free ISM band (902 MHz to 928 MHz).

10-milliwatts can reach a kilometer between two CC1101's in the open with good antennas placed high above the ground.

Modules containing the chip are usually limited to one of the three ranges. In this section, we will use the 433 MHz version that can reach the U.S. Amateur Radio frequencies.

The module is programmed using the SPI (Serial Peripheral Interface), which needs 5 pins (power, ground, clock, input, and output) as well as a chip select pin, and two general purpose pins called GDO0 and GDO2.

With 8 pins to worry about, this is already one of our most complicated modules. But it doesn't stop there. There are 47 configuration registers, 13 status registers, and many modes and functions.

The chip can support synchronous and asynchronous serial modes up to half a megabit, and packetized modes with cyclic redundancy checks, preambles, sync words, forward error correction, interleaving, and more.

Press enter or click to view image in full size

Image by the author

Because of this complexity, even something as simple as our Morse code transmitter and receiver takes quite a bit of configuring.

Alice is the transmitter. All of the pins are outputs.

Bob is the receiver. The GDO2 pin is an input and will go high when the CC1101 detects a carrier from Alice. When it does, Bob will send a square wave to the speaker attached to pin 13, and the user will hear an 800-hertz tone.

The morse.py module is only slightly changed. It simply calls the on() and off() methods of the radio module.

Our the_code.py module has not changed.

As you might expect, most of the complexity resides in the cc1101.py module:

It then sets the frequency and calibrates the oscillator. We don't use the catch0 and catch2 arguments when sending and receiving CW.

Finally, the on() and off() methods control whether the transmitter is transmitting or not by sending a signal on the GDO0 pin.

We have added an argument to the CC1101 driver: it now needs to know the baud rate. The transmitter (Alice) sets the UART tx pin to the same pin as GDO0. Alice does not care about the UART receive pin, but sets it to 9 anyway.

The cc1101.py module's only changes are to the PATABLE and the __init__() method:

patable_power_433 = [0x00,0x12,0x0E,0x34,0x60,0xC5,0xC1,0xC0]

Many of the configuration registers changed, and each baud rate causes even more to change. But beyond that, everything else is the same.

The radio is now happily sending and receiving bytes at 250,000 bits per second (25,000 bytes per second).

The whoami.py module:

{"name":"Alice","ssid":"BirdfarmOffice2"}

{"name":"Bob","ssid":"BirdfarmOffice2"}

Further Stuff

As usual, change BirdfarmOffice2 to your own SSID. In this project, we aren’t using Wi-Fi, so it doesn’t really matter.

Python Radio 40: Unleashing the CC1101

Advanced Packet Radio

Follow

20 min read

May 28, 2025

Listen

Share

More

Press enter or click to view image in full size

All photos by the Author

We used the CC1101 transceiver earlier as a CW transmitter (Morse Code).

But the radio can do so much more.

It is a full transceiver:

It can transmit and receive at data rates from 600 baud up to 600,000 baud.

It has programmable filters to match the signal-to-noise ratio to the baud rate.

It has automated packet handling, with preambles, sync words, checksums and forward error correction, modulation and demodulation, bit synchronization, byte synchronization, whitening and de-whitening, interleaving, and decoding. All without burdening the computer.

It supports amplitude modulation, frequency modulation, and phase modulation, and on-off keying. It can also handle Manchester coding and decoding.

In frequency shift keying, it can support 2-FSK, 4-FSK, and it has a Gaussian filter (GFSK) to allow higher data rates in narrower bandwidths.

It can perform six different tests to determine link quality.

So let’s explore some of the more powerful aspects of the radio.

Connecting to the ESP32

Since we want both a transmitter and a receiver, let’s use two different ESP32 boards — the ESP32-S3 and the ESP32-C3 Super Mini.

The radio has 8 pins. We will use seven of them.

Pin Connections

Our ESP32-C3 looks like this on the breadboard:

Press enter or click to view image in full size

The ESP32-S3 is a larger board, and needed two breadboards:

Press enter or click to view image in full size

Frequencies

The radio can transmit and receive on several license-free bands:

300 to 348 Megahertz

387 to 464 Megahertz

779 to 928 Megahertz

I get about a kilometer with the tiny antennas that come with the radio, even though the radio only emits 12 dBm (15.85 milliwatts).

The Software

To show the features and power of the radio, we will send and receive packets with preamble words, sync words, forward error correction, and checksums, all handled in the radio without having to code them up in the ESP32.

Here is the code:

Configuring the Radio

The __init__() method sets up the pins and tests the connection to the radio, reporting any problems.

The SNOP test is an optional debug aid to test the radio. The dump_regs() method is another debug method.

Next come the routines that read and write the registers in the radio to control its operation.

When changing the baud rate, several registers have to cooperate. The bandwidth filters have to match the data rate, and the deviation has to change (the frequency difference between a one and a zero).

The set_frequency_mhz() method calculates the values to put into the three registers that govern the frequency. We need 24 bits of data to cover the huge range of frequencies this radio can cover.

The next three methods set the address (we aren't using address filtering), channel (we only use channel zero), and sync words. Channel spacing is every 100 kilohertz, but you can also just set the frequency to anything you like in the three bands.

Now we come to the packet handling code.

Transmitting a Packet

The send_packet() method first ensures that the packets are all the same length, using the pad() method. The radio can do fixed-length packets, variable-length packets, or infinite-length packets (where the host processor handles the length). But we have chosen to use FEC, and that is only supported in fixed-length packets.

The radio has to be in IDLE state before changing into TX or RX states. We put it in IDLE, then flush any data in the transmit FIFO buffer. Then we fill the FIFO buffer.

We loop waiting to see that signal, and to see it drop back to low, indicating that the full packet has been sent.

We change back to IDLE state, and flush the transmit FIFO buffer if there were any errors.

Receiving a Packet

When GDO0 drops to low, the end of a packet has been detected, the FEC codes have corrected any correctable errors, and the checksum indicates that there are no errors detected.

Next, the bytes are read from the receive FIFO buffer, and the RSSI (Receive Signal Strength Indicator) and LQI (Link Quality Indicator) status bytes are read.

The Main Routine

Bob receives, calling receive_packet() and printing out the message.

The Output

Here is what the output looks like:

RX #11736 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109177 This additional part of th'
RX #11737 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages properly'
RX #11738 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the r'
RX #11739 -12.5 dBm: Message (60 bytes): 'eceiver.'
RX #11740 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109178'
RX #11741 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109179 This additional part of th'
RX #11742 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages properly'
RX #11743 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the r'
RX #11744 -13.0 dBm: Message (60 bytes): 'eceiver.'
RX #11745 -13.0 dBm: Message (60 bytes): 'Hello from ESP32! Message #109180'
RX #11746 -13.0 dBm: Message (60 bytes): 'Hello from ESP32! Message #109181 This additional part of th'
RX #11747 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages properly'
RX #11748 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the r'
RX #11749 -12.5 dBm: Message (60 bytes): 'eceiver.'
RX #11750 -13.0 dBm: Message (60 bytes): 'Hello from ESP32! Message #109182'
RX #11751 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109183 This additional part of th'
RX #11752 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages properly'
RX #11753 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the r'
RX #11754 -12.5 dBm: Message (60 bytes): 'eceiver.'
RX #11755 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109184'
RX #11756 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109185 This additional part of th'
RX #11757 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages properly'
RX #11758 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the r'
RX #11759 -12.5 dBm: Message (60 bytes): 'eceiver.'
RX #11760 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109186'
RX #11761 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109187 This additional part of th'
RX #11762 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages properly'
RX #11763 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the r'
RX #11764 -12.5 dBm: Message (60 bytes): 'eceiver.'
RX #11765 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109188'
RX #11766 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109189 This additional

part of th'
RX #11767 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages properly'
RX #11768 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the r'
RX #11769 -12.5 dBm: Message (60 bytes): 'eceiver.'
RX #11770 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109190'
RX #11771 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109191 This additional part of th'
RX #11772 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages properly'
RX #11773 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the r'
RX #11774 -12.5 dBm: Message (60 bytes): 'eceiver.'

Here is what the 38,400 baud signal looks like in our SDR:

Press enter or click to view image in full size

Here is what 250,000 baud looks like:

Press enter or click to view image in full size

We get much better range at the lower baud rate, due to the tighter filtering possible, which increases the signal-to-noise ratio a lot.

Gaussian Frequency Shift Keying uses two frequencies and softens the transition between them using a Gaussian filter. This uses less bandwidth than without the filtering and allows higher data rates.

The radio can also handle 4-FSK, where four tones are used, doubling the data rate in the same bandwidth. Here is what that looks like:

Here is the whoami.py module:

Here is alice.cfg (copy it to whoami.cfg on the transmitter):

{"name":"Alice","neo_pin":21,"neo_how_many":1, "gdo0":10, "gdo2":9, "csn":15, "sck":12, "mosi":11, "miso":13, "baud":38.4, "fec":1}

Here is bob.cfg (copied to whoami.cfg on the receiver):

{"name":"Bob","neo_pin":21,"neo_how_many":1, "gdo0":4, "gdo2":1, "csn":3, "sck":8, "mosi":10, "miso":9, "baud":38.4, "fec":1}

On Windows, I use this script to copy the files:

```
set comport=%1
if "%comport%" == "com4" copy alice.cfg whoami.cfg
if "%comport%" == "com5" copy bob.cfg whoami.cfg
mpremote connect %comport% rm main.py
mpremote connect %comport% rm whoamy.py
mpremote connect %comport% rm whoamy.cfg
mpremote connect %comport% cp whoami.py :whoami.py
mpremote connect %comport% cp whoami.cfg :whoami.cfg
mpremote connect %comport% cp main.py :main.py
```

You don’t actually need the lines that remove the old files. They are there because, during debugging, sometimes the processor would start up before the next file could be sent, and removing main.py stopped that.

Projects

Python Radio 9: A 40-meter CW transmitter

Simon Quellen Field

Simon Quellen Field

Aug 26, 2024

Using just the ESP32.

Press enter or click to view image in full size

Photo by author

Our little ESP32 can produce square waves at frequencies as high as 40 megahertz. But that comes with some limitations. Not all frequencies from zero to 40 MHz are actually possible.

Microprocessors use timer peripherals to produce the Pulse Width Modulation (PWM). These timers have a high-speed clock (in the case of our ESP32 that clock runs at 80 MHz) that they then "divide down". Each time the clock ticks, a counter is decremented, and when it gets to zero, a pin changes state.

Since the clock can only be "divided" by integers, only certain frequencies are actually available to us.

In the amateur radio bands available to those with a license, those frequencies are:

160-meter band: 1802816, 1807909, 1813031, 1818181, 1823361, 1828571, 1833810, 1839080, 1844380, 1849710, 1855072, 1860465, 1865889, 1871345, 1876832, 1882352, 1887905, 1893491, 1899109, 1904761, 1910447, 1916167, 1921921, 1927710, 1933534, 1939393, 1945288, 1951219, 1957186, 1963190, 1969230, 1975308, 1981424, 1987577, 1993769, 2000000

80-meter band: 3506849, 3516483, 3526170, 3535911, 3545706, 3555555, 3565459, 3575418, 3585434, 3595505, 3605633, 3615819, 3626062, 3636363, 3646723, 3657142, 3667621, 3678160, 3688760, 3699421, 3710144, 3720930, 3731778, 3742690, 3753665, 3764705, 3775811, 3786982, 3798219, 3809523, 3820895, 3832335, 3843843, 3855421, 3867069, 3878787, 3890577, 3902439, 3914373, 3926380, 3938461, 3950617, 3962848, 3975155, 3987538, 4000000

40-meter band: 7013698, 7032967, 7052341, 7071823, 7091412, 7111111, 7130919, 7150837, 7170868, 7191011, 7211267, 7231638, 7252124, 7272727, 7293447, 7314285

30-meter band: 10118577, 10138613, 10158730

20-meter band: 14027397, 14065934, 14104683, 14143646, 14182825, 14222222, 14261838, 14301675, 14341736, 14382022

17-meter band: 18091872, 18156028, 18220640

15-meter band: 21026694, 21069958, 21113402, 21157024, 21200828, 21244813, 21288981, 21333333, 21377870, 21422594, 21467505

12-meter band: 24914841, 24975609, 25036674

10-meter band: 28054794, 28131868, 28209366, 28287292, 28365650, 28444444, 28523676, 28603351, 28683473, 28764044, 28845070, 28926553, 29008498, 29090909, 29173789, 29257142, 29340974, 29425287, 29510086, 29595375, 29681159, 29767441

I have omitted the 60-meter band since it is channelized and does not allow arbitrary frequencies.

```
Def main():
Pin = Pin(18, Pin.OUT)
Pwm = PWM(pin, freq=10, duty=512)
Bands = (
( 1800000,  2000000), # 160 meters
( 3500000,  4000000), # 80 meters
( 7000000,  7300000), # 40 meters
(10100000, 10150000), # 30 meters
(14000000, 14350000), # 20 meters
(18068888, 18168000), # 17 meters
(21000000, 21450000), # 15 meters
(24890000, 24990000), # 12 meters
(28000000, 29700000)  # 10 meters
Guess = 0
For x in bands:
F_lo, f_hi = x
For f in range(f_lo, f_hi):
Pwm.freq(f)
Actual = pwm.freq()
If actual != guess:
Print(str(actual) + ", ", end=")
Guess = actual
Print()
Main()
```

That said, it is unlikely that you will get into trouble building our first transmitter, as it can only reach a few feet. If you have an amateur radio listener in the apartment next door, you might bother them, so we will place what we call a “dummy load” on our transmitter to ensure that it is not an “intentional transmitter” and will thus fall into the FCC’s Part 15 rules, and be allowed. A dummy load is just a resistor between the antenna pin and the ground pin. A nearby shortwave radio will allow you to hear the Morse code it sends, but it won’t leave the room.

Since we will be sending Morse code, we can re-use our Morse code program. But we will make a small change. Our original program modulated the transmitter with an audio signal so we could hear it on a speaker. But since we will be receiving our signal using a shortwave radio, we will instead be sending an unmodulated carrier wave. This mode is known as “continuous wave” or CW.

Our modified module looks like this:

```
Class CWMorse:
```

Character_speed = 18

Def __init__(self, pin, freq):

Self.key = PWM(Pin(pin, Pin.OUT))

Self.key.freq(freq)

Def speed(self, overall_speed):

If overall_speed >= 18:

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 >= 18:

Self.LETTER_SPACE = int(3 * self.DOT) – self.CYPHER_SPACE

Self.WORD_SPACE = int(7 * self.DOT) – self.CYPHER_SPACE

Else:

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):

For c in str:

If c == ‘ ‘:

Self.key.duty(0)

Sleep_ms(self.WORD_SPACE)

Else:

Cyphers = code[c.upper()]

For x in cyphers:

If x == ‘.’:

Self.key.duty(512)

Sleep_ms(self.DOT)

Else:

Self.key.duty(512)

Sleep_ms(self.DASH)

Self.key.duty(0)

Sleep_ms(self.CYPHER_SPACE)

Self.key.duty(0)

Sleep_ms(self.LETTER_SPACE)

The only change is in the __init__() method. Instead of using a fixed frequency of 300 Hertz, we pass in the frequency as a parameter to the method. This is because instead of modulating the carrier from an external transmitter, the ESP32 itself will be the actual transmitter.

Our main program looks like this:

Def main():

OUT_PIN = 18

F = 7032966

Pin = Pin(18, Pin.OUT)

Pwm = PWM(pin, freq=f, duty=512)   # So that we can read the actual frequency

Actual = pwm.freq()

Pwm.deinit()

Pwm = None

Cw = CWMorse(OUT_PIN, f)

Cw.speed(20)

Print(”CW transmitter”)

Msg = “AB6NY testing ESP32 as a 40 meter transmitter sending on “ + str(actual) + “ Hertz.”

While True:

Print(msg)

Cw.send(msg)

Sleep(5)

Main()

The next little bit of code is only there because we want to print out the actual frequency. We set up the PWM, get the actual frequency, and then tear it back down so that the pin is available to our CWMorse module.

Make sure you change the call sign from mine (AB6NY) to your own before you transmit.

Our simple circuit looks like this:

Press enter or click to view image in full size

Image by author

The resistor value is not critical. I used 220 ohms. A 50-ohm dummy load is more common, as most transmitters are designed to drive 50-ohm loads. Our little computer was not designed to be a transmitter, so any value that will absorb the energy will do.

We have a problem before we can get on the air, however. Our transmitter is putting out square waves.

Press enter or click to view image in full size

Photo by author

Square waves are made up of a sine wave (called the fundamental) and every odd harmonic of the fundamental. So our 7,032,967 Hertz transmitter is also transmitting on 21,098,901 Hertz (at one-third the power), 35,164,838 Hertz (at one-fifth the power), 49,230,769 Hertz (at one-seventh the power), and so on.

This is not good. We don’t want to transmit on any frequency but one.

We can clean up our signal using a low-pass filter. It will only pass the low frequency, and block (or seriously reduce) all of the others.

A simple low-pass filter is just a coil and a capacitor. The coil is in series, and the capacitor is in parallel.

Press enter or click to view image in full size

Image by author

This is called an “L” network since the inductor and capacitor form an “L” shape.

Filters not only convert square waves to sine waves by removing harmonic frequencies, they can also be used to match the output impedance of a transmitter to the input impedance of an antenna. However, the simple “L” network can only match one impedance to another with one combination of inductance and capacitance. By adding a second capacitor, we can get more freedom in selecting parts values, and we get more control over how steeply the high frequencies are attenuated.

Press enter or click to view image in full size

Image by author

This configuration is known as a “Pi” filter since the inductor and two capacitors look like the Greek letter pi.

Finally, we don’t want any DC voltage getting to our antenna when the microprocessor is starting up. So we put a capacitor between the transmitter and the filter. This will look like there is no connection when the signal is not changing.

Press enter or click to view image in full size

The resulting oscilloscope trace of our waveform now looks much closer to a sine wave:

Press enter or click to view image in full size

Photo by author

The actual filter looks like this:

Press enter or click to view image in full size

Photo by author

The input and ground connections are at the bottom right. The antenna is the black insulated wire on the left. If you are using a coax cable to an external antenna, the shield would be soldered to the ground wire at the bottom left, and the center conductor would attach where the black insulated wire is.

To aid in calculating the inductance and capacitance, we can run the following program on the desktop computer:

Import math

Def humanify(value, label):

Index = 0

While value < 0.005:

```
Index += 1

Value *= 1000000

Labels = [” “, “ u”, “ p”]

Value = round(value, 4)

Return str(value) + labels[index] + label

Def common_capacitors(value):

Vals = [10, 22, 33, 47, 68, 100, 150, 220, 330, 470, 560,

Old_diff = 1000000000000

For x in vals:

Y = x / 1000000000000

Diff = abs(value – y)

If diff < old_diff:

Return_val = y

Old_diff = diff

Return return_val

Def common_inductors(value):

Vals = [.1, .15, .47, .68, 1, 1.5, 2.2, 3.3, 4.7, 6.8, 8.2, 10, 15, 22, 33,

Old_diff = 1000000000000

For x in vals:

Y = x / 1000000

Diff = abs(value – y)

If diff < old_diff:

Return_val = y

Old_diff = diff

Return return_val

Def filter(input_impedance, output_impedance, Q, F):

If input_impedance < output_impedance:

Lo = input_impedance

Hi = output_impedance

Else:

Lo = output_impedance

Hi = input_impedance

newQ = Q

newQ += 0.1

squared = (hi / lo) / (newQ * newQ + 1 – (hi / lo))

newQ += 0.1

squared = (hi / lo) / (newQ * newQ + 1 – (hi / lo))
```

```
print(”Squared:”, squared)

if newQ > Q:

Q = newQ

Print(”Boosted Q to”, Q, “to get the impedance to match”)

C2_reactance = lo * math.sqrt((hi / lo) / (Q * Q + 1 – (hi / lo)))

C2 = 1 / (2 * math.pi * F * C2_reactance)

C1_reactance = hi / Q

C1 = 1 / (2 * math.pi * F * C1_reactance)

L1_reactance = (Q * hi + (hi * lo / C2_reactance)) / (Q * Q + 1)

L1 = L1_reactance / (2 * math.pi * F)

Print()

Print(”C1:”, humanify(C1, “F”))

Print(”L1:”, humanify(L1, “H”))

Print(”C2:”, humanify(C2, “F”))

If abs(C1 – C2) < 0.0000001:

Print(”Output impedance:”, round(math.sqrt(L1 / C2)))

If C1 > C2:

Center_frequency = 1 / (2 * math.pi * math.sqrt(L1 * C1))

Cutoff_frequency = 1 / (math.pi * math.sqrt(L1 * C1))

Else:

Center_frequency = 1 / (2 * math.pi * math.sqrt(L1 * C2))

Cutoff_frequency = 1 / (math.pi * math.sqrt(L1 * C2))

Print(”Center frequency:”, round(center_frequency))

Print(”3dB cutoff frequency:”, round(cutoff_frequency))

C1 = common_capacitors(C1)

C2 = common_capacitors(C2)

L1 = common_inductors(L1)

Print()

Print(”In closest common values:”)

Print(”C1:”, humanify(C1, “F”))

Print(”L1:”, humanify(L1, “H”))

Print(”C2:”, humanify(C2, “F”))

If abs(C1 – C2) < 0.0000001:

Print(”Output impedance:”, round(math.sqrt(L1 / C2)))

If C1 > C2:

Center_frequency = 1 / (2 * math.pi * math.sqrt(L1 * C1))

Cutoff_frequency = 1 / (math.pi * math.sqrt(L1 * C1))
```

Else:

Center_frequency = 1 / (2 * math.pi * math.sqrt(L1 * C2))

Cutoff_frequency = 1 / (math.pi * math.sqrt(L1 * C2))

Print(”Center frequency:”, round(center_frequency))

Print(”3dB cutoff frequency:”, round(cutoff_frequency))

Def main():

Print()

Print(”For example, try 50, 50, 1, 7000000”)

Print()

Input_impedance = float(input(”Input impedance? “))

Output_impedance = float(input(”Output impedance? “))

Q = float(input(”Q? “))

F = float(input(”Center frequency? “))

Filter(input_impedance, output_impedance, Q, F)

Main()

For example, try 50, 50, 1, 7000000

Input impedance? 50

Output impedance? 50

Q? 1

Center frequency? 7032967

C1: 452.5969 pF

L1: 1.1315 uH

C2: 452.5969 pF

Output impedance: 50

Center frequency: 7032967

3dB cutoff frequency: 14065934

In closest common values:

C1: 470.0 pF

L1: 1.0 uH

C2: 470.0 pF

Output impedance: 46

Center frequency: 7341270

3dB cutoff frequency: 14682540

A low-pass filter takes advantage of the properties of coils and capacitors. If we put direct current across a capacitor, it will charge up to its capacitance, but then it looks like an open switch, and no energy passes through. But at high frequencies, the capacitor charges and discharges rapidly, and the alternating current passes through as if the switch were closed.

A coil has the opposite characteristic. It lets direct current through once it has created the magnetic field around it. But that magnetic field collapses when the current is turned off or reversed, and as it collapses, it generates current in the coil, keeping the current flowing in the same direction it had been. So a coil resists changes in the current.

In our filter, low frequencies are barely impeded by our tiny coil. It does not build up much of a magnetic field because it is just a few turns of wire. Since the transmitter and the antenna are directly connected to the coil, low frequencies pass right on through.

Python Radio 10: Another 40-meter transmitter

Follow

5 min read

Aug 27, 2024

Listen

Share

More

This time using the Raspberry Pi Pico 2040

Press enter or click to view image in full size

Image by Raspberry Pi

When we built the 40-meter CW transmitter using the ESP32 we noted that the PWM feature could not reach all frequencies in the amateur radio bands because it had to divide its 80 MHz timer clock by integers.

The $4 Raspberry Pi Pico 2040 has a PWM timer that runs at the system clock frequency (nominally 133 MHz but you can change that). This gives us these frequencies in the amateur radio bands:

160-meter band: 1811594, 1824818, 1838235, 1851852, 1865672, 1879699, 1893939, 1908397, 1923077, 1937984, 1953125, 1968504, 1984127, 2000000,
80-meter band: 3521127, 3571429, 3623188, 3676471, 3731343, 3787879, 3846154, 3906250, 3968254,
40-meter band: 7142857, 7352941,
20-meter band: 14705882,
10-meter band: 27777778

Things look a little sparse. Only one frequency in the areas where technicians can operate.

But, maybe all is not lost.

It might take all night to finish running the program. Or it might still be running next week.

Maybe there’s a better way.

The following program will send beeps to a radio in CW mode on the frequency 7030000 Hertz:

The program uses a feature of the RP2040 called the PIO. That is a set of programmable state machines (rudimentary computers) that have their own instruction set and can be controlled by either of the two main processors in the chip.

the rp2.StateMachine line says to run the square program on state machine 0, at a rate of 2 * self.f (since there are two instructions in the program, and we want a square wave at the frequency self.f on the pin self.pin.

Once we have set up our state machine, we can activate it and deactivate it to control whether or not we are transmitting.

With this little bit of code, we now have access to every amateur radio frequency in the HF bands, with single Hertz resolution.

Our CW transmitter program now looks like this:

While the RP2040's PIO processor allows a much better range of frequencies than its or the ESP32's PWM feature, it is still based on a clock that is divided by integers. We can't get single-hertz resolution with it.

Raspberry Pi

Raspberry Pi Pico

Electronics

Radio

Python

Python Radio 27: Work the World

Using a Raspberry Pi Pico to control a 1,500-watt transmitter

Simon Quellen Field

Simon Quellen Field

Follow

6 min read

Sep 21, 2024

Listen

Share

More

Press enter or click to view image in full size

Propagation map of the world

Screenshot by the author

Throughout this series, I have emphasized frequencies and power levels that are legal in the U.S. to use without a license. Those frequencies are so high (their wavelengths are so short) that they travel right through the ionosphere and out into space.

These frequency bands have names like VHF and UHF (very high frequency and ultra-high frequency).

Right below them is HF. High Frequency. These bands can refract and reflect off of the ionosphere and the earth to bounce around the globe.

Not surprisingly, when your signal can cross national boundaries you need a license recognized by international treaties. Such a license is very easy to get.

Having said all that, this project is about how to "key" a radio transmitter to send Morse code. The ideas here can be used to turn on and off many non-radio devices, up to 40 volts and half an ampere.

Our little Raspberry Pi Pico can only handle 3 volts, and only a few milliwatts. To switch on bigger things, like a 1,500-watt transmitter, we will use a transistor: the 2N4401 (although any NPN transistor will work).

2N4401 pinout

Screenshot by the author

Press enter or click to view image in full size

RP2040 with 2N4401 and mono plug

Image by author

The image above shows our entire hardware setup. The base of the transistor connects to pin 13 of the RP2040. The emitter connects to ground. The collector does not touch the computer. The transistor acts like a switch, connecting the collector to ground, and this completes the circuit to anything the mono plug is plugged into.

Press enter or click to view image in full size

A close view

A closer view (author's image)

```
Class CWMorse:
Character_speed = 18
Def __init__(self, pin):
Self.key = Pin(pin, Pin.OUT)
Def speed(self, overall_speed):
If overall_speed >= 18:
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 >= 18:
Self.LETTER_SPACE = int(3 * self.DOT) – self.CYPHER_SPACE
Self.WORD_SPACE = int(7 * self.DOT) – self.CYPHER_SPACE
Else:
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):
For c in str:
If c == ' ':
Self.key.off()
Sleep_ms(self.WORD_SPACE)
```

Else:

Cyphers = code[c.upper()]

For x in cyphers:

If x == ‘.’:

Self.key.on()

Sleep_ms(self.DOT)

Else:

Self.key.on()

Sleep_ms(self.DASH)

Self.key.off()

Sleep_ms(self.CYPHER_SPACE)

Self.key.off()

Sleep_ms(self.LETTER_SPACE)

The lines self.key.on() and self.key.off() turn pin 13 high (3 volts) and low (0 volts) respectively.

Since pin 13 is connected to the base of the transistor, 3 volts turn the transistor fully on, as if the collector was connected directly to ground. Turning off pin 13 disconnects the collector, turning off whatever the mono plug controls.

Press enter or click to view image in full size

The RP2040 connected to a Pixie 2 transceiver.

Photo by author

Our main.py module is simple:

Def main():

Cw = CWMorse(13)

Cw.speed(10)

Print(“CW keyer”)

Msg = “AB6NY testing RP2040 as a CW keyer.”

While True:

Print(msg)

Cw.send(msg)

Sleep(5)

Main()

The module the_code.py looks like this:

Code = {

‘A’: ‘.-‘,

‘B’: ‘-...’,

‘C’: ‘-.-.’,

‘D’: ‘-..’,

‘E’: ‘.’,

‘F’: ‘..-.’,

‘G’: ‘--.’,

‘H’: ‘....’,

‘I’: ‘..’,

‘J’: ‘.---‘,

‘K’: ‘-.-‘,

‘L’: ‘.-..’,

‘M’: ‘—‘,

‘N’: ‘-.’,

‘O’: ‘---‘,

‘P’: ‘.--.’,

‘Q’: ‘--.-‘,

‘R’: ‘.-.’,

‘S’: ‘...’,

‘T’: ‘-‘,

‘U’: ‘..-‘,

‘V’: ‘...-‘,

‘W’: ‘.—‘,

‘X’: ‘-..-‘,

‘Y’: ‘-.—‘,

‘Z’: ‘--..’,

An excellent walk-through of the Pixie 2 transceiver is here.

Press enter or click to view image in full size

The NS-40+ QRP transmitter.

Image by author

Connect to the same EFHW antenna and as much as 12 volts (shown above using a 9-volt battery, which works fine and gets over 3 watts out). That’s enough power to get anywhere in the world with good antennas.

You can buy a 1,500-watt amplifier to boost either of these transmitters to the full legal limit. But why, when you can already work the whole world?

Radio

Decoding POCSAG using Gqrx & RTL-SDR

Noë Flatreaud

Noë Flatreaud

Follow

Projects

4 min read

Oct 6, 2024

Listen

Share

More

Recent events, involving Hezbollah, Mossad and the explosion of thousands of trapped pagers, sparked my interest in learning more about the POCSAG protocol and its decoding process.

In this article we'll find out POCSAG pager protocol specifications, behaviour and how to sniff it.

POC-what ?

POCSAG (Post Office Code Standardisation Advisory Group) also known as Radio Paging Code ■1 or RPC1 is a one-way 2FSK paging protocol that supports 512, 1200, and 2400 bps speed. Transmissions can include tone, numeric, and alphanumeric data. The protocol uses FSK modulation with a ±4.5 kHz shift on the center carrier, where a +4.5 kHz shift represents a 0 and a -4.5 kHz shift represents a 1. You can find POCSAG signals on the VHF or UHF band and 12.5 or 25 kHz channel spacing.

Typical 2 & 4 FSK Power Spectrum you'd find

Press enter or click to view image in full size

Here is an example POCSAG signal waterfall (right) and it's decoded messages (left)

Frequencies

POCSAG pagers operate on various frequencies depending on the region. Here are some common frequency ranges:

HF-High/VHF-Low Band: 25 MHz — 54 MHz

VHF Mid Band: 66 MHz — 88 MHz

VHF High Band: 138 MHz — 175 MHz

UHF: 406 MHz — 422 MHz

UHF High: 435 MHz — 512 MHz

'900' Band: 929 MHz — 932 MHz

You can find each specific frequency by region and service on

Required Hardware and Software

RTL-SDR Dongle: A USB dongle capable of receiving frequencies from 500 kHz up to 1.75 GHz.

Gqrx: An open-source software-defined radio receiver.

Sox: A command-line audio processing tool.

For this tutorial, I am using Dragon OS, an all-in-one GNU/Linux distribution dedicated to radio hacking and wireless activities, but you can install it standalone.

Setting Up Gqrx

Launch Gqrx: Open the Gqrx application.

Configure the RTL-SDR Dongle: Select your RTL-SDR device from the input controls.

Press enter or click to view image in full size

Make sure that your Audio output is sampled at 48 kHz.

Enable UDP Server: Go to the “Input Controls” tab and enable the UDP server. Set the port to 7355.

The remote host and port number are configurable.

Once configured, you can start streaming signals.

Press enter or click to view image in full size

You can verify the data is coming through at the opposite end using netcat:

$ nc -l -u 7355

You should see a lots of symbols scroll through the terminal that you can pipe to the next tool.

Capturing and Decoding the Signal

You task now is to capture the signal received from the RTL-SDR in Gqrx piped through the UDP Socket on port 7355.

Use the following command to capture the signal from Gqrx and decode it using Multimon-ng:

$ nc -l -u localhost 7355 | sox -t raw -esigned-integer -b16 -r 48000 - -t raw -esigned-integer -b16 -r 22050 - | multimon-ng -t raw -a POCSAG512 -a POCSAG1200 -a POCSAG2400 -f alpha -e --timestamp –

You’d normally be able to receive plaintext messages from nearby emergencies & firefighters

Press enter or click to view image in full size

Hooray, you just sniffed and decoded paging activity

Python Radio 26: Double Your Power

…and also build a full duplex repeater

Simon Quellen Field

Simon Quellen Field

Follow

8 min read

Sep 19, 2024

Listen

Share

More

Press enter or click to view image in full size

Two powerful robot twins

MidJourney

With this project, we will double the range of our communications and double the processing power of our computer at the same time.

Our HC-12 radio was half duplex. It could either transmit or receive, but not both at the same time. This project will fix that (by using two HC-12 modules. Hey, they’re cheap.)

The radios already have a range of over a kilometer. But we always want more. A repeater is a radio that can listen to our HC-12 on one channel, and repeat that data on another channel. We can place it between two locations that want to communicate but can’t, either because they are too far away, or because there is an obstruction in the way, such as a building, or in my case, a mountain.

To enable us to listen to one radio and transmit on the other at the same time, we will be using both processors on either the ESP32 or the RP2040. They both have two main processors, in addition to some lesser processors that we won’t be using here.

We do this by using a lock. MicroPython makes using both processors easy, and it makes locking memory very easy as well, as we will see.

```
LED = Pin(2, Pin.OUT)

Count = 0

Def blink(cps, lock):

Global count

Print(“Blink is running”)

While True:

LED.value(LED.value() ^ 1)

Sleep(1/cps)

With lock:

Count += 1

Lock = allocate_lock()

Def main():

Global count, lock

While True:

With lock:

C = count

Print(“Main says blink count is”, c)

Sleep(3)

Start_new_thread(blink, (5,lock))

Main()
```

The blink() routine will run on the second processor. It announces itself, and then loops, flashing the LED at the rate cps (cycles per second), and keeping a count of how many times it blinked.

We allocate a lock, and pass it as an argument to blink() when we start the thread on the second processor using the start_new_thread() method. The arguments to blink() are sent in a tuple which is the second argument to start_new_thread().

Then we call main(), which loops, getting the lock before reading the blink count and printing it.

Notice how easy it is to synchronize the two processors using the with lock: statement.

We are now ready to build our repeater.

Press enter or click to view image in full size

433 MHz repeater using the RP2040

Photo by the author

The photo shows the repeater using the RP2040. The radio in front is using UART 1 on pins 4 and 5. The radio in the back is using UART 0 on pins 16 and 17. I put two electrolytic capacitors from power to ground to reduce noise, but they may not be necessary (the user guide suggests using one).

Class HC12:

Def __init__(self, unum, tx, rx, setpin, baud):

Self.unum = unum

Self.tx = tx

Self.rx = rx

Self.setpin = setpin

Self.baud = baud

Self.response = ‘’

Self.set = Pin(self.setpin, Pin.OUT)

Self.uart = UART(self.unum, baudrate=9600, tx=Pin(self.tx), rx=Pin(self.rx), timeout=100, timeout_char=20)

Self.long_distance()

Def cleanup(self, data):

If data:

Return data

Def command(self, cmd):

Self.set(0)

Sleep_ms(40)

If cmd == ‘’:

Self.uart.write(“AT”)

Else:

Self.uart.write(“AT+” + cmd)

Sleep_ms(40)

Self.set(1)

Sleep_ms(1000)   # Give the module time to process the command and reply

Self.response = self.read()

Self.response = self.cleanup(self.response)

Try:

If self.response:

Try:

Decoded = self.response.decode(‘utf-8’)

Self.response = decoded

Except Exception as e:

Print(“Can’t decode “, self.response, “into utf-8”, e)

Except Exception as e:

Print(“Exception in HC12.command:”, e)

Print(“Response was:”, self.response)

Def long_distance(self):

Self.command(“FU4”)

Sleep_ms(100)

Self.uart.deinit()

Self.uart = UART(self.unum, baudrate=1200, tx=Pin(self.tx), rx=Pin(self.rx), timeout=100, timeout_char=20)

Sleep_ms(1000)

Def write(self, msg):

Cnt = self.uart.write(msg)

Return cnt

Def read(self):

Return self.uart.read()

Def read_str(self):

R = self.cleanup(self.uart.read())

If r:

Return r.decode(‘utf-8’)

Return “”

Def any(self):

Return self.uart.any()

Def status(self):

Self.command(“RX”)

Print(self.response)

The rest of the code follows the previous HC-12 project and should be familiar or self-explanatory.

Our main.py module looks like this:

LED = Pin(25, Pin.OUT)

W = WhoAmI()

Send = None

Run_thread = True

Def transmit(lock):

Global send, run_thread

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

Radio.long_distance()

```
Radio.command(w.tx_channel())
Radio.status()
Print("Transmit is running")
While run_thread:
With lock:
If send:
Print("Transmit sending: " + send)
LED(0)
Radio.write(send)
LED(1)
Send = None
Class Ping:
Def __init__(self):
Self.radio = HC12(1, 4, 5, 3, 1200)
Self.radio.long_distance()
Self.radio.command(w.tx_channel())
Self.radio.status()
Self.count = 0
Def timeout(self, timer):
Self.radio.write(w.name() + " sending " + str(self.count))
Self.count += 1
Def main():
Global send, run_thread, lock
Print("I am:", w.name())
Lock = allocate_lock()
Radio = HC12(0, 16, 17, 15, 1200)
Radio.long_distance()
Radio.command(w.rx_channel())      # Not actually needed: channel 1 is the default
Radio.status()
If w.name() == "repeater":
Start_new_thread(transmit, (lock,))
Tim = None
If w.name() == "rover1":
Ping = Ping()
Tim = Timer(period=5000, mode=Timer.PERIODIC, callback=ping.timeout)
Try:
While True:
```

If radio.any():

Input_str = radio.read_str()

Print(“Main received: “ + input_str)

With lock:

Send = input_str

Except KeyboardInterrupt as e:

Print(“KeyboardInterrupt in main:”, e)

If tim:

Print(“Shutting down timer”)

Tim.deinit()

Run_thread = False

Else:

Print(“No timer to shut down”)

Sleep(5)

Main()

This code will be run on two identical boards. One board will be the repeater. The other board will be called “rover1” and it will act as the client radio sending and receiving from the repeater. The two roles are established by the whoami.py and whoami.cfg files which will be shown shortly.

Starting with main(), we see that it prints out its name, allocates a lock, and sets UART 0 to be on channel 1.

If its name is “repeater”, it starts the transmit() thread on the second processor.

If it is “rover1” it starts a timer to call a timeout every five seconds. This will send a message to the repeater, acting as the client radio.

The main() routine then loops, reading from UART 0, and putting anything it reads into the shared variable send so that the transmit() thread can rebroadcast it.

There are a couple of housekeeping details we need to worry about. If we want to shut things down by hitting control-C, we want to turn off the timer and stop the thread running on the other processor.

The transmit() thread sets up UART 1 on the transmit channel (which we set in the whoami.cfg file). I use channel 4 to transmit and channel 1 to receive. The first four channels are legal to use without a license. The higher 96 channels require an amateur radio license but have the advantage that you can use an amplifier to get as much as 50 watts of output power.

You can power the repeater with a solar panel and a battery and place it in a waterproof box on a roof or the top of a mountain.

Any number of rovers can use the repeater. It is up to them to play nice and take turns so the text is not intermingled. The standard way to do this is to send the word “over” when you are finished with a thought.

{“name”:”rover1”,”tx_channel”:”C001”,”rx_channel”:”C004”}

{“name”:”repeater”,”tx_channel”:”C004”,”rx_channel”:”C001”}

Note that the receive channel and the transmit channel are swapped between the two.

The whoami.py file looks like this:

```
Class WhoAmI:
Def __init__(self):
Try:
With open(“whoami.cfg”,”rb”) as f:
Line = f.read(1024)
Self.me = loads(line)
Except OSError as e:
Print(“Error reading whoami.cfg:”, e )
Def name(self):
If “name” in self.me:
Return self.me[“name”]
Return “Unknown”
Def tx_channel(self):
If “tx_channel” in self.me:
Return self.me[“tx_channel”]
Return None
Def rx_channel(self):
If “rx_channel” in self.me:
Return self.me[“rx_channel”]
Return None
```

Radio Hackers

Python

Radio

Python Radio 51: A Peek Under The Covers

Inside MicroPython to Build an SDR Radio

Simon Quellen Field

Simon Quellen Field

Follow

20 min read

Aug 26, 2025

Listen

Share

More

Press enter or click to view image in full size

Peeking under the covers

MidJourney

In Python Radio 50, we built an entire AM radio in software. In Python, no less.

This $5 processor will replace both the laptop computer and the RTL-SDR USB dongle we have used in the past.

I knew going into this project that it was ambitious. What I did not know was that it would take me six weeks of effort. For you, it will only take a few minutes, as I will provide not only the working code but the script to build it.

The RP2350 has a 12-bit analog-to-digital converter (ADC) that we will connect to an antenna. It also features a pulse-width modulator (PWM) that we will connect to a speaker. We add a battery and the software, and we're done.

Right away, we run into problems.

MicroPython on the RP2350 only supports reading one sample at a time from the ADC. This would be OK if we could call it millions of times in a second, but MicroPython is not up to that.

We have the same problem with the PWM. One sample at a time.

To solve both problems, we need to add high-speed DMA access to both peripherals to the MicroPython firmware. That means we need to write a Python module in C and link it in.

So we do it the hard way (something that in itself cost me three weeks of work). But I made a script that does all the work. You just fire and forget.

Wsl -d Ubuntu

Now you are running a Linux shell. It's that easy.

In my Linux home directory, I made two subdirectories, sdr_radio, and AM_sdr_radio_final. Then I executed the following shell script to build MicroPython with my sdr_radio.c module (placed in the sdr_radio directory):

```
Set -e # Exit immediately if any command fails

MPY_VERSION=”v1.25.0”

BOARD=”RPI_PICO2”

PROJECT_ROOT=~/AM_sdr_pico2_final

USER_SOURCE_DIR=~/sdr_radio

If [ ! -d “${USER_SOURCE_DIR}” ]; then

Echo “Error: User source directory not found at ${USER_SOURCE_DIR}”

Exit 1

Fi

Echo “--- STARTING THE DEFINITIVE BUILD (MANUAL VENDOR + CORRECT PATHS) ---”

Echo “--- [1/5] Setting up project structure...”

Rm -rf ${PROJECT_ROOT}

Mkdir -p ${PROJECT_ROOT}

Echo “--- [2/5] Cloning MicroPython and its core submodules...”

Git clone –depth 1 -b ${MPY_VERSION}  ${PROJECT_ROOT}/micropython

cd ${PROJECT_ROOT}/micropython

git submodule update --init --recursive

Git submodule add  lib/pico-extras
```

git submodule update --init lib/pico-extras

Echo “Manually downloading and vendoring CMSIS-DSP library...”

Mkdir -p ./lib/vendor/CMSIS_5

Wget -O cmsis.zip

Unzip -q cmsis.zip -d ./lib/vendor/

Mv ./lib/vendor/CMSIS_5-5.9.0/* ./lib/vendor/CMSIS_5/

Rm cmsis.zip

Rm -rf ./lib/vendor/CMSIS_5-5.9.0/

ARM_MATH_PATH=”./lib/vendor/CMSIS_5/CMSIS/DSP/Include/arm_math.h”

Echo “Verifying that arm_math.h exists at ${ARM_MATH_PATH}...”

If [-f “$ARM_MATH_PATH”]; then

Echo “SUCCESS: arm_math.h found in vendored directory.”

Else

Echo “FATAL ERROR: arm_math.h was NOT found after manual download.”

Exit 1

Fi

Echo “--- [3/5] Creating sdr_radio module and copying all required sources... ---”

MODULE_PATH=./extmod/sdr_radio

Mkdir -p ${MODULE_PATH}

Echo “Copying your staged module files from ${USER_SOURCE_DIR}...”

Cp ${USER_SOURCE_DIR}/* ${MODULE_PATH}/

Echo “--- [4/5] Configuring the MicroPython build... ---”

Cd ./ports/rp2

Cp mpconfigport.h.orig mpconfigport.h 2>/dev/null || cp mpconfigport.h mpconfigport.h.orig

Echo “” >> mpconfigport.h

Echo “// Enable the custom sdr_radio module” >> mpconfigport.h

Echo “#define MICROPY_PY_SDR_RADIO (1)” >> mpconfigport.h

Cp CMakeLists.txt.orig CMakeLists.txt 2>/dev/null || cp CMakeLists.txt CMakeLists.txt.orig

TARGET_LINE_SOURCES=”set(PICO_SDK_COMPONENTS”

CUSTOM_BLOCK_SOURCES=”\

\n\

Include_directories(\n\

\${MICROPY_DIR}/extmod/sdr_radio \n\

\${MICROPY_DIR}/py \n\

\${MICROPY_DIR}/ports/rp2 \n\

\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Include \n\

\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/PrivateInclude \n\

```
\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/Core/Include \n\
\${MICROPY_DIR}/lib/pico-extras/src/rp2_common/pico_audio_i2s/include \n\
\${MICROPY_DIR}/lib/pico-extras/src/common/pico_audio/include \n\
\${MICROPY_DIR}/lib/pico-extras/src/common/pico_util_buffer/include \n\
)\n\
\n\
Set(CMSIS_DSP_SOURCES\n\
${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/BasicMathFunctions/BasicMathFunctions.c\"\n\
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/CommonTables/CommonTables.c\"\n\
CROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/ComplexMathFunctions/ComplexMathFunctions.c\"\n\
MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/ControllerFunctions/ControllerFunctions.c\"\n\
"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/FastMathFunctions/FastMathFunctions.c\"\n\
${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/FilteringFunctions/FilteringFunctions.c\"\n\
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/MatrixFunctions/MatrixFunctions.c\"\n\
MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/StatisticsFunctions/StatisticsFunctions.c\"\n\
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/SupportFunctions/SupportFunctions.c\"\n\
${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/TransformFunctions/TransformFunctions.c\"\n\
DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/QuaternionMathFunctions/QuaternionMathFunctions.c\"\n\
)\n\
\n\
List(APPEND MICROPY_SOURCE_PORT \n\
\${MICROPY_DIR}/extmod/sdr_radio/sdr_radio.c\n\
\${CMSIS_DSP_SOURCES}\n\
)\n\
List(APPEND MICROPY_SOURCE_QSTR \${MICROPY_DIR}/extmod/sdr_radio/sdr_radio.c)\n\
\n\
Echo "--- [5/5] Starting the final MicroPython build ---"
Make -j4 BOARD=${BOARD}
Echo ""
Echo "--- BUILD SUCCESSFUL! ---"
Echo "Firmware is at: ${PROJECT_ROOT}/micropython/ports/rp2/build-${BOARD}/firmware.uf2"
Ls -l build-${BOARD}/firmware.uf2
Cp build-${BOARD}/firmware.uf2 /mnt/c/simon/sdr_radio_pico
Echo "--- VERIFYING MODULE PRESENCE IN SYMBOL TABLE ---"
If arm-none-eabi-nm "build-${BOARD}/firmware.elf" | grep -q "sdr_radio_user_cmodule"; then
Echo "SUCCESS: sdr_radio module symbol found in the firmware."
Else
```

Echo "ERROR: sdr_radio module symbol was NOT found in the firmware."

Exit 1

Fi

Echo ""

Echo "--- ALL STEPS COMPLETE. The module will now be visible in the REPL. ---"

Whew!

Now we have a file called firmware.uf2. We hold down the little button on the RP2350, cycle power, and it's in boot mode, and shows up in Windows as a new disk drive. We copy firmware.uf2 into that new directory, and the microcomputer boots the new firmware.

Of course, before building it, we need our new module:

Typedef struct _sdr_radio_obj_t {

Mp_obj_base_t base;

Uint32_t tune_freq_hz;

Q31_t nco_phase;   // Current phase accumulator

Q31_t nco_phase_increment;   // Phase step per sample

Q31_t nco_i;   // Current I value (cos) of the NCO, Q31 format

Q31_t nco_q;   // Current Q value (sin) of the NCO, Q31 format

Q31_t nco_cos_inc;   // Pre-calculated cos(phase_increment)

Q31_t nco_sin_inc;   // Pre-calculated sin(phase_increment)

Q31_t dc_block_i_x1;

Q31_t dc_block_i_y1;

Q31_t dc_block_q_x1;

Q31_t dc_block_q_y1;

Q31_t ema_i_s1, ema_i_s2, ema_i_s3;

Q31_t demod_mag_x1;

Q31_t audio_hpf_x1;

Q31_t audio_hpf_y1;

Q31_t agc_smoothed_peak;

Q31_t audio_ema_lpf;

Bool is_am_mode;

Q31_t bfo_phase;

Q31_t bfo_phase_increment;

////////////// Transmitter Section ///////////////////////////

Uint32_t tx_carrier_freq_hz;

Q31_t tx_nco_phase;

Q31_t tx_nco_phase_increment;

Float32_t tx_modulation_index;

Uint32_t capture_sample_rate;

Uint32_t capture_num_samples;

Uint32_t adc_clkdiv;

} sdr_radio_obj_t;

// The internal C buffers that the DMA will write to.

// The size MUST match the buffer size used in the Python script.

Static int adc_dma_chan_A = -1;

Static int adc_dma_chan_B = -1;

Static uint32_t capture_buf_A[MAX_CAPTURE_BUFFER_SIZE];

Static uint32_t capture_buf_B[MAX_CAPTURE_BUFFER_SIZE];

// Helper function to guarantee a clean state

Static void reset_sdr_state(sdr_radio_obj_t *self) {

Self->nco_phase = 0;

Self->dc_block_i_x1 = 0;

Self->dc_block_i_y1 = 0;

Self->dc_block_q_x1 = 0;

Self->dc_block_q_y1 = 0;

Self->ema_i_s1=0; self->ema_i_s2=0; self->ema_i_s3=0;

Self->ema_q_s1=0; self->ema_q_s2=0; self->ema_q_s3=0;

Self->agc_smoothed_peak = 1000;

// Initialize the Audio HPF state

Self->demod_mag_x1 = 0;

Self->audio_hpf_y1 = 0;

Self->bfo_phase = 0;

Self->audio_ema_lpf = 0;

// Exposed to Python to make tests deterministic

Static mp_obj_t sdr_radio_reset_state(mp_obj_t self_in) {

Sdr_radio_obj_t *self = MP_OBJ_TO_PTR(self_in);

Reset_sdr_state(self);

Return mp_const_none;

Static MP_DEFINE_CONST_FUN_OBJ_1(sdr_radio_reset_state_obj, sdr_radio_reset_state);

Static mp_obj_t sdr_radio_set_mode(mp_obj_t self_in, mp_obj_t is_am_obj) {

Sdr_radio_obj_t *self = MP_OBJ_TO_PTR(self_in);

Self->is_am_mode = mp_obj_is_true(is_am_obj);

Return mp_const_none;

Static MP_DEFINE_CONST_FUN_OBJ_2(sdr_radio_set_mode_obj, sdr_radio_set_mode);

Static mp_obj_t sdr_radio_make_new(const mp_obj_type_t *type, size_t n_args, size_t n_kw, const

```
mp_obj_t *args) {

Sdr_radio_obj_t *self = mp_obj_malloc(sdr_radio_obj_t, type);

Reset_sdr_state(self);

Self->bfo_phase = 0;

Self->nco_phase_increment = (uint32_t)( ( (uint64_t)self->tune_freq_hz << 32 ) /
ADC_SAMPLE_RATE );

Self->capture_sample_rate = 0;

Self->capture_num_samples = 0;

Return MP_OBJ_FROM_PTR(self);

Static mp_obj_t sdr_radio_tune(mp_obj_t self_in, mp_obj_t freq_obj) {

Sdr_radio_obj_t *self = MP_OBJ_TO_PTR(self_in);

// 1. Get the desired station frequency (e.g., 810000) from Python.

Uint32_t station_freq_hz = mp_obj_get_int(freq_obj);

// --- Alias Calculation ---

// This logic calculates the NCO frequency needed to tune to a station

// by using undersampling (aliasing) to bring it into the first Nyquist zone.

// Find the remainder when the station frequency is divided by the sample rate.

Uint32_t remainder = station_freq_hz % ADC_SAMPLE_RATE;

Uint32_t nco_tune_freq_hz;

// Check which half of the Nyquist zone the remainder falls into.

If (remainder < (ADC_SAMPLE_RATE / 2)) {

// If it’s in the lower half, the alias appears directly.

Nco_tune_freq_hz = remainder;

} else {

// If it’s in the upper half, the alias is mirrored from the top.

Nco_tune_freq_hz = ADC_SAMPLE_RATE – remainder;

// Store the calculated NCO frequency in our object.

Self->tune_freq_hz = nco_tune_freq_hz;

// Recalculate the NCO phase increment with the new frequency.

Self->nco_phase_increment = (q31_t)(((uint64_t)self->tune_freq_hz << 31) / ADC_SAMPLE_RATE);

Return mp_const_none;

Static MP_DEFINE_CONST_FUN_OBJ_2(sdr_radio_tune_obj, sdr_radio_tune);

Static mp_obj_t fast_sdr_pipeline(mp_obj_t self_in, mp_obj_t args_in) {

Sdr_radio_obj_t *self = MP_OBJ_TO_PTR(self_in);

Size_t n_args;

Mp_obj_t *args;

Mp_obj_get_array(args_in, &n_args, &args);
```

If (n_args < 3) {

Mp_raise_TypeError(MP_ERROR_TEXT("Requires at least adc, out, and scratch buffers"));

Mp_buffer_info_t adc_info;   mp_get_buffer_raise(args[0], &adc_info,   MP_BUFFER_READ);

Mp_buffer_info_t out_info;   mp_get_buffer_raise(args[1], &out_info,   MP_BUFFER_WRITE);

Mp_buffer_info_t scratch_info; mp_get_buffer_raise(args[2], &scratch_info, MP_BUFFER_WRITE);

// --- Buffer Pointers and Sizes ---

Uint16_t *adc_in_ptr = (uint16_t *)adc_info.buf;

Uint32_t *pwm_out_ptr = (uint32_t *)out_info.buf;

Const int num_adc_samples = adc_info.len / sizeof(uint16_t);

Const int num_audio_samples = out_info.len / sizeof(uint32_t);

// --- DSP Constants ---

Const q31_t DC_BLOCK_R = 0x7F800000;

Const q31_t RF_LPF_ALPHA = 0x20000000; // Alpha=0.25, wide ~20kHz RF LPF

Const q31_t RF_LPF_ONE_MINUS_ALPHA = 0x7FFFFFFF – RF_LPF_ALPHA;

Const int DECIMATION_FACTOR = ADC_SAMPLE_RATE / 22050;

Const q31_t AUDIO_HPF_R = 0x7E000000; // ~112 Hz HPF cutoff

Q31_t *temp_audio_buf = (q31_t*)scratch_info.buf;

Int audio_idx = 0;

Int decimation_counter = 0;

Q31_t i_filtered = 0;

Q31_t q_filtered = 0;

If (self->is_am_mode) {

For (int i = 0; i < num_adc_samples; i++) {

Q31_t sample = ((q31_t)adc_in_ptr[i] – 2048) << 19;

Q31_t nco_s = arm_sin_q31(self->nco_phase);

Q31_t nco_c = arm_cos_q31(self->nco_phase);

Self->nco_phase += self->nco_phase_increment;

Q31_t i_raw = mult_q31(sample, nco_c);

// 3-Stage Cascaded EMA Low-Pass Filter

Q31_t i_s1_out = mult_q31(self->ema_i_s1, RF_LPF_ONE_MINUS_ALPHA) + mult_q31(i_raw, RF_LPF_ALPHA);

Self->ema_i_s1 = i_s1_out;

Q31_t i_s2_out = mult_q31(self->ema_i_s2, RF_LPF_ONE_MINUS_ALPHA) + mult_q31(i_s1_out, RF_LPF_ALPHA);

Self->ema_i_s2 = i_s2_out;

// q31_t i_filtered = mult_q31(self->ema_i_s3, RF_LPF_ONE_MINUS_ALPHA) + mult_q31(i_s2_out, RF_LPF_ALPHA);

```
I_filtered = mult_q31(self->ema_i_s3, RF_LPF_ONE_MINUS_ALPHA) + mult_q31(i_s2_out, RF_LPF_ALPHA);

Self->ema_i_s3 = i_filtered;

Q31_t q_s1_out = mult_q31(self->ema_q_s1, RF_LPF_ONE_MINUS_ALPHA) + mult_q31(q_raw, RF_LPF_ALPHA);

Self->ema_q_s1 = q_s1_out;

Q31_t q_s2_out = mult_q31(self->ema_q_s2, RF_LPF_ONE_MINUS_ALPHA) + mult_q31(q_s1_out, RF_LPF_ALPHA);

Self->ema_q_s2 = q_s2_out;

// q31_t q_filtered = mult_q31(self->ema_q_s3, RF_LPF_ONE_MINUS_ALPHA) + mult_q31(q_s2_out, RF_LPF_ALPHA);

Q_filtered = mult_q31(self->ema_q_s3, RF_LPF_ONE_MINUS_ALPHA) + mult_q31(q_s2_out, RF_LPF_ALPHA);

Self->ema_q_s3 = q_filtered;

// Decimation and Audio Path

If (++decimation_counter >= DECIMATION_FACTOR) {

Decimation_counter = 0;

If (audio_idx < num_audio_samples) {

// --- AM Demodulation (Fast Approximation) ---

Q31_t abs_i = (i_filtered > 0) ? i_filtered : -i_filtered;

Q31_t abs_q = (q_filtered > 0) ? q_filtered : -q_filtered;

Q31_t max_val, min_val;

If (abs_i > abs_q) {

Max_val = abs_i;

Min_val = abs_q;

} else {

Max_val = abs_q;

Min_val = abs_i;

// Magnitude ≈ max + 0.25*min

Q31_t magnitude = __QADD(max_val, min_val >> 2);

Q31_t demodulated_signal = magnitude;

// Audio HPF

Q31_t diff = __QSUB(demodulated_signal, self->audio_hpf_x1);

Q31_t sum = __QADD(self->audio_hpf_y1, diff);

Q31_t audio_sample = mult_q31(AUDIO_HPF_R, sum);

Self->audio_hpf_x1 = magnitude;

Self->audio_hpf_y1 = audio_sample;

Temp_audio_buf[audio_idx++] = audio_sample;
```

} else {

For (int i = 0; i < num_adc_samples; i++) {

// Step 1: ADC Scaling

Q31_t sample = ((q31_t)adc_in_ptr[i] – 2048) << 19;

// Step 2: NCO & Mixer

Q31_t nco_s = arm_sin_q31(self->nco_phase);

Q31_t nco_c = arm_cos_q31(self->nco_phase);

Self->nco_phase += self->nco_phase_increment;

Q3

Python Radio 52: Undercover Adventures

An AM Transmitter All in Software

Follow

7 min read

2 days ago

Listen

Share

More

Press enter or click to view image in full size

MidJourney

In , we built fast analog-to-digital and digital-to-analog routines that used DMA, freeing up the CPU to do digital signal processing.

Now we will use those routines to transmit AM radio signals, using nothing more than the $5 Raspberry Pi Pico 2 (the RP2350 and two resistors).

Press enter or click to view image in full size

The Completed AM Radio Transmitter (all photos by the author)

Our Python code uses lessons we learned when building the receiver:

The ADC ping-pong happens in the C code. We manage the PWM ping-pong buffers in Python.

The RP2350 can easily run at 250 MHz without needing any extra cooling or heat syncs, so we set that up right away. For our transmitting frequency, we chose 600 kHz somewhat arbitrarily (there were no local stations on that frequency in my area).

The real work is done in am_transmit_pipeline().

It takes as input the ADC buffer we collect from the analog-to-digital converter. In this case, I am using input from the sound card on my computer, which ranges from -1 volt to 1 volt.

Because the ADC can't see voltages less than zero, we use two 10kΩ resistors to make a voltage divider. We connect them in series from the 3.3-volt positive power supply to ground.

The ADC now gives us numbers in the range of zero to 4095. The middle is 2048, so by subtracting that amount and then dividing by 2048, we get numbers between -1 and 1. We have just undone the work of the two resistors.

The main loop in am_transmit_pipeline() is a linear interpolator, which smoothes out the jumps between the samples, making it appear that we were sampling at 5 MHz instead of 22050 Hz.

We use the phase increment we established when tuning to walk through the loop, creating the carrier sine wave. We multiply that by the interpolated modulation signal from the ADC buffer, giving us an AM-modulated carrier wave.

Our final output looks like this:

Press enter or click to view image in full size

In this run, we didn't max out the volume from the sound card, so the modulation depth does not go from zero to 4095. All that solid blue in the top graph is actually a sine wave, as you can see when we zoom in:

Press enter or click to view image in full size

That low modulation depth shows up in the small amount of energy in the sidebands, as shown in the frequency spectrum. AM modulation has a peak in the center, at the carrier frequency, and two (in this case, tiny) sidebands that carry the information content.

I sent the transmitter a 1600 Hz sine wave as my input. That's why the two sidebands are 1600 Hz from the carrier. Voice or music would show much broader sidebands.

So there you have it — an AM transmitter done all in software. And you get to see how MicroPython works under the covers using fast routines in C to deal with hardware and to speed up digital signal processing (DSP).

Programming

Software Development

Python

Radio

Digital Signal Processing

Python Radio 12: An AM Transmitter

Follow

4 min read

Aug 30, 2024

Listen

Share

More

Using the RP2040 microcontroller

Press enter or click to view image in full size

MidJourney

We can easily turn the RP2040 into an AM radio transmitter. It is capable of transmitting AM signals anywhere in the entire AM radio band, and also (with a license) in any of the HF amateur radio bands.

The first step is to generate a square wave on the frequency to which we will tune the radio. In the example here, we will use 540 kilohertz. We will output that on pin 15 of the RP2040.

If we connect pin 15 to either a long wire antenna or (more easily) to a good ground, such as the metal case of some equipment that is plugged into the wall, or the screw that holds the outlet cover or switch cover to the wall, then the signal will be heard clearly throughout the building without any further amplification.

With just the carrier, all we will hear on the AM radio is that the static suddenly gets quiet when we start transmitting.

To AM modulate the signal, we can simply add a PWM tone on another pin (such as pin 14) and connect that to pin 15. Now we hear that tone on the radio.

Of course, we can then send Morse code using that tone. Here is the code to do that, starting with the main routine:

The cwmorse.py module looks familiar, with just a few changes:

We have added a tone pin and changed the on() and off() methods to turn on and off the PWM on that pin. The rest of the module is unchanged.

Using the household wiring ground allows us to hear the signal throughout the building, without much of the signal escaping to bother anyone next door. The low power levels and lack of an antenna make the device legal to operate.

If you like, you can modify the code to use the ringtones from our previous project to send annoying music over the radio.

We can, however, arrange to send less annoying signals. I have an antique AM radio, and I like to have it play old-time radio shows instead of modern AM radio broadcasts.

On the Internet, you can find many MP3 recordings of old-time radio shows. Your computer has an audio jack where speakers or earphones can be connected. We connect the shield of the phone plug to the ground of the RP2040, and the tip of the phone plug to pin 15. We also connect the anode of a 1N4148 diode to pin 15. The cathode (the side of the diode that has the black band) connects to a good ground.

The diode acts as a modulator, allowing the sound from the computer to vary the amplitude of the signal from the RP2040.

My various modern AM radios can hear the signal clearly anywhere in the house. My antique radio picks up too much noise if it is farther away than about four feet from the transmitter. This may be because its cord does not have a ground wire. Still, the transmitter can hide in a drawer, and the radio can sit on the cabinet above the drawer and play old radio shows. The Lone Ranger, Fibber McGee and Molly, the CBS Mystery Show, Jack Benny, and even radio news recordings of the Hindenburg disaster or the Pearl Harbor attack.

The schematic looks like this:

Press enter or click to view image in full size

Image by author

If the MP3 player was emitting a 16 kilohertz square wave, and the RP2040 was putting out a 1600 kilohertz carrier, the modulated waveform would look something like this:

Press enter or click to view image in full size

Image by author

Python Radio 51: A Peek Under The Covers

Inside MicroPython to Build an SDR Radio

Follow

20 min read

Aug 26, 2025

Projects

Listen

Share

More

Press enter or click to view image in full size

MidJourney

In Python Radio 50, we built an entire AM radio in software. In Python, no less.

This $5 processor will replace both the laptop computer and the RTL-SDR USB dongle we have used in the past.

I knew going into this project that it was ambitious. What I did not know was that it would take me six weeks of effort. For you, it will only take a few minutes, as I will provide not only the working code but the script to build it.

The RP2350 has a 12-bit analog-to-digital converter (ADC) that we will connect to an antenna. It also features a pulse-width modulator (PWM) that we will connect to a speaker. We add a battery and the software, and we're done.

Right away, we run into problems.

MicroPython on the RP2350 only supports reading one sample at a time from the ADC. This would be OK if we could call it millions of times in a second, but MicroPython is not up to that.

We have the same problem with the PWM. One sample at a time.

To solve both problems, we need to add high-speed DMA access to both peripherals to the MicroPython firmware. That means we need to write a Python module in C and link it in.

So we do it the hard way (something that in itself cost me three weeks of work). But I made a script that does all the work. You just fire and forget.

wsl -d Ubuntu

Now you are running a Linux shell. It's that easy.

In my Linux home directory, I made two subdirectories, sdr_radio, and AM_sdr_radio_final. Then I executed the following shell script to build MicroPython with my sdr_radio.c module (placed in the sdr_radio directory):

Whew!

Now we have a file called firmware.uf2. We hold down the little button on the RP2350, cycle power, and it's in boot mode, and shows up in Windows as a new disk drive. We copy firmware.uf2 into that new directory, and the microcomputer boots the new firmware.

Of course, before building it, we need our new module:

That's a lot of code to go over. But much of it is boilerplate used to connect C to MicroPython.

Unlike the RTL-SDR, which has megahertz of bandwidth, our ADC can only run as fast as 500,000 samples per second.

But AM radio starts above that, at 530 kHz, and goes up from there to 1,700 kHz. How can we sample way up there?

We embrace aliasing. We sub-sample.

There's a strong AM radio station in my area at 810 kHz. Sampling at 500 kHz gives me an alias frequency using this formula:

alias = 500 - (810 % 500)

The 500 is our sampling rate, and the 810 is the frequency we want to listen to. The result is 190 kHz. We need to tune to 190 kHz.

We create a sine wave (and a cosine wave) at 190 kHz and multiply our incoming ADC buffer by that. This “mixes” our target signal down to zero (DC). Since an AM signal carries the sound in two side-bands, we have just mixed the carrier down to DC and one sideband off the chart altogether, leaving the high sideband down in the audio range.

We get rid of the energy in the carrier (which is now DC) using a DC blocker (a high-pass filter). Then we get rid of the high-frequency aliases (that are above our hearing anyway) using a low-pass filter.

To demodulate the AM signal, we would like to find the square root of the sum of the in-phase signal and the quadrature signal. These are the signals we created by mixing (respectively) the sine and the cosine waves.

But the square root subroutine takes too long. To our rescue comes the arm_math.h library, which is a set of highly optimised math routines that use special Arm instructions. The __QADD() routine does an approximation of the math we need.

Lastly, we filter once again to remove any thumping artifacts our math has created below the audio range. This is another high-pass filter.

So, what does Python see?

After setting up the sizes and sample rates, it calls audio_configure() to set up the PWM DMA.

Then it gets an instance of the SDR_Radio object. This has the configure_and_init_capture() method to set up the ADC DMA.

We “prime the pump” by capturing an ADC buffer, running it through our SDR pipeline to filter, mix, demodulate, filter, and AGC, and then finally play the audio using audio_play_chunk().

While the buffer is playing, the ADC is collecting the next buffer of samples. The decimation ensures that the time it takes to play the audio is exactly the same time needed to fetch the next buffer of ADC samples.

While both of these are happening in the background, we have 16 milliseconds to process the buffer in the pipeline. Thanks to the fancy Arm math library, this only takes about 5 milliseconds, so we never stall the ADC or the PWM.

Our little RP2350, with only the help of a long wire and a speaker, is now a full-blown AM/CW/SSB radio.

And it only took six weeks of scull-sweat and about 5,000 re-compiles of the MicroPython code.

Python Radio 30: Catch the Fox

A Game Like Cat and Mouse

Simon Quellen Field

Simon Quellen Field

Follow

9 min read

Dec 1, 2024

Listen

Share

Projects

More

Press enter or click to view image in full size

A fox with a transmitter.

MidJourney

Radio fox hunting. It's a fun game that came about through necessity.

It is unfortunately often that we have to find a source of radio interference, rescue a boat at sea, or find an enemy transmitter on a battlefield. Radio amateurs practice doing this with an entertaining game called a fox hunt.

Someone hides a small battery-powered transmitter, and others use techniques and tools to find it. It's a bit like geocaching. At Easter time, it's fun to hide a transmitter inside a plastic Easter egg and play the game with kids.

We will build a few transmitters and receivers, as well as show how to use amateur radio transceivers and directional antennas in the hunt.

In keeping with our unspoken theme in this series, we will add the constraint of being cheap, as well as small, battery-powered, and programmable in Micropython.

Our computer

Press enter or click to view image in full size

A handful of ESP8266–01S's

A handful of ESP8266–01S's

They are 32-bit computers, running at 160 megahertz, with a megabit of flash memory (128k bytes), 32k bytes of instruction RAM, 80k bytes of data RAM, 802.11 b/g/n Wi-Fi, I2C, SPI, UART, and a 10-bit ADC.

I can't help but compare them to the original IBM PC, which was much slower, only 8-bit, had no flash memory, had less memory, no UART, no Wi-Fi, networking, I2C, ADC, or SPI. And the IBM PC was 1,500 times more expensive.

The original Apple Macintosh came out a few years later with a 16-bit computer, but it also lacked the specs of the ESP8266–01S and was much slower and had no networking.

Since we want long battery life, we chose the ESP8266–01S over the Wemos D1 Mini we have used before. The D1 has a USB adapter on the board, which uses power all the time. It is also larger than the 01S, and we want to hide our fox in little pill bottles.

Programming

Press enter or click to view image in full size

ESP8266–01S programmer

ESP8266–01S programmer

The 01S plugs into it like this:

Press enter or click to view image in full size

ESP8266–01S programmer with computer

ESP8266–01S programmer with computer

Be sure to get one with the programming switch on it, or you will have to solder one on yourself. Some have a slide switch like the one above, others have a pushbutton.

To program the 01S, set the switch to PROG (or hold down the push button) and plug it into a

USB slot. Once it is plugged in, it is programming mode and you can release the button or set the slide to UART.

I use the following setup.cmd program on Windows to program the 01S:

Set comport=%1

Esptool –port %comport% erase_flash

Pause

Esptool –port %comport% --baud 460800 write_flash –flash_size=detect 0 esp8266_mpy_1M.bin –verify

Micropython

Download that last one, and rename it esp8266_mpy_1M.bin, and then run “setup com3”. Your com port will probably be some other number.

The Python code

Def beep(f):

Print(“F is”, f)

Speaker = Pin(0, Pin.OUT)

PWM(speaker, freq=f, duty=512)

Sleep_ms(100)

PWM(speaker, freq=f, duty=0)

Def main():

Sta = WLAN(STA_IF)

Sta.active(True)

Ap = WLAN(AP_IF)

Ap.active(True)

Mac = ap.config(‘mac’)

Mac_str = “”

For b in mac:

Ch = hex(b)

Mac_str = mac_str + ch[2:] + ‘:’

Mac_str = mac_str[:-1]

Who_am_i = “Fox “ + mac_str

Print(“I am”, who_am_i)

Ap.config(essid=who_am_i, authmode=AUTH_OPEN)

Nets = sta.scan()

For net in nets:

Print(“net:”, net)

Ssid = net[0].decode(“utf-8”)

If ssid.startswith(“Fox “):

Print("Found a fox!", ssid)

Sta.connect(ssid)

While not sta.isconnected():

Print("Connected to", ssid)

Break

While True:

Rssi = sta.status('rssi')

Print("RSSI:", rssi)

If rssi < 0:

Beep((90 - -rssi) * 10)

Sleep_ms(300)

Main()

To load main.py onto the 01S, we use ampy:

Ampy -p com3 put main.py

The beep() function uses pulse-width-modulation to send a tone to pin 0. We will connect a speaker to that pin.

The main() function sets up two Wi-Fi connections. One is an access point. We copy its MAC address into a string and add it to "Fox " to create its SSID.

The hound looks at the RSSI (receive signal strength indicator) and calls the beep() function to emit a tone that rises in pitch as we get closer to the transmitter.

The receiver

Press enter or click to view image in full size

Wiring harness

Wiring harness

When everything is connected, it looks like this:

Press enter or click to view image in full size

Wired up

Wired up

I prefer rechargeable batteries to AAs, but two or three AA cells in series will work just fine instead of the lithium polymer battery I used. AA cells (and my LiPo battery) produce 1800 milliampere-hours of power. This will power the radios all day.

The positive wire from the battery connects to the 3V3 pin on the computer. The negative wire connects to ground. The last wire to connect is the one between the speaker and the IO0 pin. The other side of the speaker is connected to ground.

The IO0 pin cannot be connected before the computer is powered up, or the computer will not boot.

The Transmitter

Press enter or click to view image in full size

The transmitter

The transmitter

We just connect it to the battery.

The Game

I like to hide the transmitter in a little pill bottle. They are waterproof, so I can toss it into wet weeds or a freshly watered lawn. Plastic Easter eggs are also fun.

A poor man's dish

Blocking the signal with your body is a time-tested radio direction finding trick. But we can do better.

You are probably familiar with parabolic reflectors. They are used in telescopes, in directional microphones, and solar cookers. But making a dish parabolic is a chore.

Instead, we will build a spherical dish. Our radio has an antenna that is larger than the focus of a parabolic dish anyway, and a spherical dish will still focus the energy on our (relatively) large antenna (the gold printed squiggle on the printed circuit board).

I inflated a latex balloon and crumpled some aluminum foil over it. It was spherical enough. At 2.4 gigahertz, the wavelength is about 12 centimeters. This means the crumpled peaks and valleys of the foil are still well under a tenth of a wave, so the radio waves will focus nicely.

On top of the foil I used school glue and paper towels to make a quick and dirty paper mache back to keep the foil in shape. A pencil stuck through the center of the dish holds the radio.

Press enter or click to view image in full size

The front of the dish

The front of the dish

The battery and speaker are hot glued to the back.

Press enter or click to view image in full size

The back of the dish

The back of the dish

The pitch of the receiver changes quite a bit as we aim the dish around — much more than it did when we just blocked the signal with our body. The dish focuses the signal when aimed at it, and blocks the signal when aimed away. It is still light enough to easily hold in one hand, and the paper mache is stiff enough to handle normal use.

Going Further

It uses the FS1000A transmitter, and optionally a ground plane antenna. We change the main.py code just a little:

Ap_if = WLAN(AP_IF)

St_if = WLAN(STA_IF)

Sleep_type(SLEEP_MODEM) # Adding this made no difference since we turned the WIFI off

Def main():

If ap_if.active():

Ap_if.active(False) # Disable access point

St_if.active(False) # Disable station interface

```
Deepsleep(1, 4)     # The radio only turns off when we go into deepsleep
GPIO_0 = 0
GPIO_2 = 2
Led = Pin(GPIO_2)
Led.off()
Morse = Morse(GPIO_0)
Wpm = 5
Morse.speed(int(wpm))
While(True):
Morse.send("catch me if you can!")
Main()
```

Here again we use the 01S to save battery power. To save more power, we turn off the Wi-Fi radio. A peculiarity of this computer is that we can only turn off Wi-Fi by going into deep sleep mode. When we wake up (by grounding the reset pin or toggling power) Wi-Fi remains off (unless we turn it back on in Micropython or reflash the device).

At 19 milliamps of power draw (21 when transmitting) the 1800 mAh battery (LiPo or AA) will last about a hundred hours.

We can use the little Morse receiver from the earlier project as the receiver, but it has a serious drawback: the signal does not change as you get closer. If it hears the Morse code at all, it beeps the speaker. Neither the volume nor the pitch change.

We can play with the receiver's antenna, blocking it or shortening it until the signal goes away. But this is a challenge.

We can use our Yagi-Uda antenna, or we can use a more robust commercial antenna with its much lower wind load than a big sheet of foam core board.

Press enter or click to view image in full size

A Baofeng radio with a Yagi-Uda directional antenna

A Baofeng radio with a Yagi-Uda directional antenna

An antenna like that can be used to talk to amateur radio satellites. A computer program that tracks the satellites will tell you where to aim.

Radio Hackers

Python

Radio

Python Radio 42: Buttons!

Simon Quellen Field

Simon Quellen Field

Follow

17 min read

Jun 8, 2025

Listen

Projects

Share

More

Reverse Engineering a Remote Control

Press enter or click to view image in full size

Remote control and its receiver.

All photos by the author.

It is remarkably robust as a system, delivering long-range without false positives.

In the photo above, I connected four green LEDs to the outputs of the receiver. Pushing a button toggles the respective LED (D, C, B, or A). I pushed the B button to take the picture.

But what if we want to have a computer running Python send codes to the receiver? Or receive codes from the transmitter?

To do that, we would need to know the protocol the devices use to talk to one another.

Luckily, we have an RTL-SDR and some Python code to operate it.

Let’s look at the code I came up with:

```
Def normalize(data, limit):

L = len(data)

Result = [0] * l

For i in range(l):

If abs(data[i]) >= limit:

Result[i] = 1

Return result

Def smooth(data):

Result = []

For i in range(len(data) – 11):

Result.append(sum(data[i:i+10])/10)

Return result

Def main():

Import numpy as np

Import matplotlib.pyplot as plt

Sdr = RtlSdr()

Sdr.sample_rate = 2048000 # Hz

Sdr.center_freq = 433.92e6   # Hz

Sdr.freq_correction = 60  # PPM

Sdr.gain = 49.6

Sdr.read_samples(4096)          # Throw away the first few samples

Print(“Reading samples”)

X = sdr.read_samples(2048000 * 5)
```

```
Sdr.close()
Print("Done reading samples")
Reals = x.real
Burst = []
For i in range(len(reals)):
For q in range(8_000):
If abs(reals[i+q]) > .5:
I += q
Break
Else:
I += 100
While abs(reals[i]) < .5:
I += 1
Start = i – 10
Burst = reals[start:start+90_000]
Plt.figure(figsize=(25, 2), dpi=100)
Plt.xlabel("Milliseconds")
Plt.minorticks_on()
Plt.plot(burst)
Plt.savefig("rtlsdr.svg", dpi=300)
Plt.tight_layout()
Plt.show()
Burst = normalize(burst, .6)
Burst = smooth(burst)
Burst = normalize(burst, .4)
Plt.figure(figsize=(25, 2), dpi=100)
Plt.xlabel("Milliseconds")
Plt.minorticks_on()
Plt.plot(burst)
Plt.savefig("rtlsdr_canonical.svg", dpi=300)
Longs = []
Shorts = []
I = 0
Stop = len(burst) – 5
While i < stop:
If burst[i]:
Count = 0
```

```
While i < stop and burst[i]:
Count += 1
I += 1
Print(f"{i}: {count} samples high {round(count / 2.048)} microseconds")
If count > 100:
If count < 1000:
Shorts.append(count)
Else:
Longs.append(count)
Else:
Count = 0
While i < stop and burst[i] == 0:
Count += 1
I += 1
If count > 100:
If count < 3000:
If count < 1000:
Shorts.append(count)
Else:
Longs.append(count)
Print(f"{i}: {count} samples low {round(count / 2.048)} microseconds")
If len(shorts) > 0 and len(longs) > 0:
Avg_short = (sum(shorts) / len(shorts)) / 2.048
Avg_long = (sum(longs) / len(longs)) / 2.048
Print(f"Average long pulse length: {avg_long} microseconds")
Print(f"Average short pulse length: {avg_short} microseconds")
Plt.tight_layout()
Plt.show()
Exit()
Print("No burst found")
Plt.plot(reals)
Plt.legend(["Signal", "Samples"])
Plt.savefig("rtlsdr.svg", dpi=300)
Plt.show()
Main()
```

At first, all I did was tell the RTL-SDR where to look (around 433.92 MHz) and gather some samples, showing them in a graph.

The Data

I got something that looked like this when I pushed the A button:

Press enter or click to view image in full size

Raw button data.

Some noise, then a short pulse, then some noise, then a long pulse, and so on. There were 25 pulses in all. The spacing between the pulses varied.

That's when I wrote the normalise() and smooth() functions to clean up the data. Now it looked like this:

Press enter or click to view image in full size

Processed button data.

It looks like the short pulse is followed by a silence that is three times longer than the short pulse.

The long pulse is three times longer than the short pulse, and it is followed by a silence that is as long as the short pulse.

I decided to call short-long a zero, and long-short a one.

The four buttons produce these codes:

REMOTE_BUTTON_CODES = {

'A': "0110000011000011111110000",

'B': "0110000011000011111101000",

'C': "0110000011000011111100100",

'D': "0110000011000011111100010",

The last five bits are what distinguish the buttons. More likely, four bits and a stop bit. The first 20 bits are thus the address, or a sync code. This prevents noise from triggering the receiver.

The Timings

The RTL-SDR code prints out this information:

Found Rafael Micro R820T/2 tuner

Reading samples

Done reading samples

8: 8 samples low 4 microseconds

469: 461 samples high 225 microseconds

2529: 2060 samples low 1006 microseconds

4509: 1980 samples high 967 microseconds

5262: 753 samples low 368 microseconds

7277: 2015 samples high 984 microseconds

8039: 762 samples low 372 microseconds

8606: 567 samples high 277 microseconds

10694: 2088 samples low 1020 microseconds

11299: 605 samples high 295 microseconds

13392: 2093 samples low 1022 microseconds

13970: 578 samples high 282 microseconds

16051: 2081 samples low 1016 microseconds

16616: 565 samples high 276 microseconds

18722: 2106 samples low 1028 microseconds

19332: 610 samples high 298 microseconds

21403: 2071 samples low 1011 microseconds

23361: 1958 samples high 956 microseconds

23365: 4 samples low 2 microseconds

23381: 16 samples high 8 microseconds

23386: 5 samples low 2 microseconds

23395: 9 samples high 4 microseconds

24150: 755 samples low 369 microseconds

26147: 1997 samples high 975 microseconds

26924: 777 samples low 379 microseconds

27479: 555 samples high 271 microseconds

29577: 2098 samples low 1024 microseconds

30181: 604 samples high 295 microseconds

30183: 2 samples low 1 microseconds

30202: 19 samples high 9 microseconds

32288: 2086 samples low 1019 microseconds

32855: 567 samples high 277 microseconds

34939: 2084 samples low 1018 microseconds

35561: 622 samples high 304 microseconds

37656: 2095 samples low 1023 microseconds

39650: 1994 samples high 974 microseconds

39654: 4 samples low 2 microseconds

39682: 28 samples high 14 microseconds

40453: 771 samples low 376 microseconds

42460: 2007 samples high 980 microseconds

43230: 770 samples low 376 microseconds

45201: 1971 samples high 962 microseconds

45206: 5 samples low 2 microseconds

45223: 17 samples high 8 microseconds

46004: 781 samples low 381 microseconds

47994: 1990 samples high 972 microseconds

47995: 1 samples low 0 microseconds

48024: 29 samples high 14 microseconds

48790: 766 samples low 374 microseconds

50812: 2022 samples high 987 microseconds

51581: 769 samples low 375 microseconds

53579: 1998 samples high 976 microseconds

54361: 782 samples low 382 microseconds

56335: 1974 samples high 964 microseconds

56340: 5 samples low 2 microseconds

56355: 15 samples high 7 microseconds

57158: 803 samples low 392 microseconds

57687: 529 samples high 258 microseconds

57690: 3 samples low 1 microseconds

57706: 16 samples high 8 microseconds

59852: 2146 samples low 1048 microseconds

60463: 611 samples high 298 microseconds

62578: 2115 samples low 1033 microseconds

63142: 564 samples high 275 microseconds

63144: 2 samples low 1 microseconds

63161: 17 samples high 8 microseconds

65269: 2108 samples low 1029 microseconds

65801: 532 samples high 260 microseconds

65802: 1 samples low 0 microseconds

65820: 18 samples high 9 microseconds

75760: 9940 samples low 4854 microseconds

76223: 463 samples high 226 microseconds

78292: 2069 samples low 1010 microseconds

80798: 2506 samples high 1224 microseconds

81661: 863 samples low 421 microseconds

83706: 2045 samples high 999 microseconds

84451: 745 samples low 364 microseconds

84994: 543 samples high 265 microseconds

84997: 3 samples low 1 microseconds

85013: 16 samples high 8 microseconds

87106: 2093 samples low 1022 microseconds

87695: 589 samples high 288 microseconds

87700: 5 samples low 2 microseconds

87716: 16 samples high 8 microseconds

89791: 2075 samples low 1013 microseconds

89984: 193 samples high 94 microseconds

Average long pulse length: 1008.9742726293102 microseconds

Average short pulse length: 312.73626512096774 microseconds

This is the information we need to reverse engineer the device, using our CC1101 transceiver from the earlier projects (this one and this one).

The Micropython Code

Import gc

REMOTE_BUTTON_CODES = {

‘A’: “0110000011000011111110000”,

‘B’: “0110000011000011111101000”,

‘C’: “0110000011000011111100100”,

‘D’: “0110000011000011111100010”,

IOCFG2 = 0x00; IOCFG1 = 0x01; IOCFG0 = 0x02; FIFOTHR = 0x03

SYNC1 = 0x04; SYNC0 = 0x05; PKTLEN = 0x06; PKTCTRL1 = 0x07

PKTCTRL0 = 0x08; ADDR = 0x09; CHANNR = 0x0A; FSCTRL1 = 0x0B

FSCTRL0 = 0x0C; FREQ2 = 0x0D; FREQ1 = 0x0E; FREQ0 = 0x0F

MDMCFG4 = 0x10; MDMCFG3 = 0x11; MDMCFG2 = 0x12; MDMCFG1 = 0x13

MDMCFG0 = 0x14; DEVIATN = 0x15; MCSM2 = 0x16; MCSM1 = 0x17

MCSM0 = 0x18; FOCCFG = 0x19; BSCFG = 0x1A; AGCCTRL2 = 0x1B

AGCCTRL1 = 0x1C; AGCCTRL0 = 0x1D; WOREVT1 = 0x1E; WOREVT0 = 0x1F

WORCTRL = 0x20; FREND1 = 0x21; FREND0 = 0x22; FSCAL3 = 0x23

FSCAL2 = 0x24; FSCAL1 = 0x25; FSCAL0 = 0x26; RCCTRL1 = 0x27

RCCTRL0 = 0x28; FSTEST = 0x29; PTEST = 0x2A; AGCTEST = 0x2B

TEST2 = 0x2C; TEST1 = 0x2D; TEST0 = 0x2E

PARTNUM = 0xF0; VERSION = 0xF1; MARCSTATE = 0xF5; RSSI = 0xF4; LQI = 0xF3

TXBYTES = 0xFA; RXBYTES = 0xFB; PKTSTATUS = 0xF8

SRES = 0x30; SRX = 0x34; STX = 0x35; SIDLE = 0x36; SCAL = 0x33

SFRX = 0x3A; SFTX = 0x3B; SNOP = 0x3D

PATABLE_ADDR = 0x3E; TXFIFO_ADDR = 0x3F; RXFIFO_ADDR = 0x3F

WRITE_SINGLE_BYTE = 0x00; READ_SINGLE_BYTE = 0x80; WRITE_BURST = 0x40; READ_BURST = 0xC0

STATE_SLEEP = 0x00

STATE_IDLE = 0x01

STATE_XOFF = 0x02

STATE_VCOON_MC = 0x03

STATE_REGON_MC = 0x04

STATE_MANCAL = 0x05

STATE_VCOON = 0x06

STATE_REGON = 0x07

STATE_STARTCAL = 0x08

STATE_BWBOOST = 0x09

STATE_FS_LOCK = 0x0A

STATE_IFADCON = 0x0B

STATE_ENDCAL = 0x0C

STATE_RX = 0x0D

STATE_RX_END = 0x0E

STATE_RX_RST = 0x0F

STATE_TXRX_SWITCH = 0x10

STATE_RXFIFO_OVERFLOW = 0x11

STATE_FSTXON = 0x12

STATE_TX = 0x13

STATE_TX_END = 0x14

STATE_RXTX_SWITCH = 0x15

STATE_TXFIFO_UNDERFLOW = 0x16

FXOSC = 26000000

Base_frequency = 433.92

Channel = 0

SNIFF_MAX_FRAMES_TO_CAPTURE = 3

SNIFF_MAX_BITS_PER_FRAME = 24

SNIFF_MIN_PULSE_US = 150

SNIFF_MAX_PULSE_US = 2000

SNIFF_MIN_SYNC_DURATION_US = 3000

SNIFF_FRAME_TIMEOUT_MS = 300

IDEAL_T_PULSE_SHORT_US = 350

IDEAL_T_PULSE_LONG_US = 1000

SYNC_PULSE_LOW_US = 7000

Def accurate_sleep_us(delay):

Irq_state = disable_irq()

Try:

Start_wait = ticks_us()

While ticks_diff(ticks_us(), start_wait) < delay:

Pass

Finally:

```
Enable_irq(irq_state)
Import network
Import bluetooth
Def disable_radios():
Sta_if = network.WLAN(network.STA_IF)
If sta_if.active():
Sta_if.active(False)
Print("Wi-Fi STA disabled.")
Ap_if = network.WLAN(network.AP_IF)
If ap_if.active():
Ap_if.active(False)
Print("Wi-Fi AP disabled.")
Try:
Ble = bluetooth.BLE()
If ble.active():
Ble.active(False)
Print("Bluetooth disabled.")
Except Exception as e:
Print(f"Could not disable Bluetooth (or not supported): {e}")
Def enable_radios():
Pass
Class CC1101_ASK_Tool:
Def __init__(self, spi, cs_pin_id, gdo0_pin_id):
Self.spi = spi
Self.cs = Pin(cs_pin_id, Pin.OUT)
Self.cs.on()
Self.current_mode = None # "SNIFF_RX" or "ASK_TX"
If gdo0_pin_id is None:
Self.data_pin = Pin(gdo0_pin_id, Pin.IN, Pin.PULL_DOWN)
Print(f"GDO0 (data input) configured on GPIO {gdo0_pin_id}")
Self.reset()
Self.idle()
Print("CC1101 ASK Tool Initialized and Idle.")
Print("--- Basic Register Read Test ---")
Try:
Self._write_reg(CHANNR, 0xBB)
Channr_read = self._read_reg(CHANNR)
```

If channr_read == 0xBB:

Print("Basic register write/read test PASSED.")

Else:

Print(f"ERROR: Basic register write/read test FAILED! Wrote 0xBB to CHANNR, Read: 0x{channr_read:02X}")

Self._write_reg(CHANNR, 0x00)

Except Exception as e:

Print_exception(e)

Print(f"Error during basic register read test: {e}")

Def _strobe(self, cmd):

Self.cs.off(); self.spi.write(bytearray([cmd])); self.cs.on(); sleep_us(50)

Def _write_reg(self, addr, value):

Self.cs.off(); self.spi.write(bytearray([addr | WRITE_SINGLE_BYTE, value])); self.cs.on(); sleep_us(50)

Def _read_reg(self, addr):

Self.cs.off()

Wbuf = bytearray([addr | READ_SINGLE_BYTE, 0x00]); rbuf = bytearray(2)

Self.spi.write_readinto(wbuf, rbuf); val = rbuf[1]

Def _read_status_reg(self, status_addr_with_header):

Self.cs.off()

Wbuf = bytearray([status_addr_with_header, 0x00]); rbuf = bytearray(2)

Self.spi.write_readinto(wbuf, rbuf); val = rbuf[1]

Def _write_burst_reg(self, addr, data):

Self.cs.off(); self.spi.write(bytearray([addr | WRITE_BURST])); self.spi.write(bytearray(data)); self.cs.on(); sleep_us(50)

Def _read_burst_reg(self, addr, length):

Self.cs.off()

Tx_header_byte = addr | READ_BURST

Wbuf = bytearray([tx_header_byte] + [0x00] * length)

Rbuf = bytearray(1 + length)

Self.spi.write_readinto(wbuf, rbuf)

Data = rbuf[1:]

Def reset(self):

Self.cs.off(); sleep_us(10); self.cs.on(); sleep_us(45)

Self._strobe(SRES); sleep_ms(5)

Def idle(self):

Self._strobe(SIDLE)

```
Sleep_ms(1)
For i in range(150):
Marc_state = self._read_status_reg(MARCSTATE) & 0x1F
If i % 50 == 0 and i > 0 :
Sleep_us(100)
Sleep_us(50)
Print(f"Warning: CC1101 did not confirm IDLE. Last MARCSTATE: 0x{marc_state:02X}")
Def set_frequency_mhz(self, freq_mhz=433.92, channel=0):
Freq_hz = int(freq_mhz * 1_000_000 + channel * 100_000)
Freq_reg_val = int((freq_hz * (1 << 16)) / FXOSC)
F2 = (freq_reg_val >> 16) & 0xFF
F1 = (freq_reg_val >> 8) & 0xFF
F0 = freq_reg_val & 0xFF
Self._write_reg(FREQ2, f2); self._write_reg(FREQ1, f1); self._write_reg(FREQ0, f0)
Def configure_for_sniffer_rx(self):
Self.reset()
Self.idle()
Self.set_frequency_mhz(base_frequency, channel)
Self._write_reg(IOCFG0, 0x0D)
Read_iocfg0 = self._read_reg(IOCFG0)
If read_iocfg0 != 0x0D: print(f"ERROR: IOCFG0 not set to 0x0D, is 0x{read_iocfg0:02X}")
Self._write_reg(MDMCFG2, 0x30)
Self._write_reg(MDMCFG4, 0x6A)
Self._write_reg(MDMCFG3, 0x22)
Self._write_reg(DEVIATN, 0x00)
Self._write_reg(PKTCTRL0,0x30)
Self._write_reg(AGCCTRL2, 0x07)
Self._write_reg(AGCCTRL1, 0x00)
Self._write_reg(AGCCTRL0, 0xB0)
Self._write_reg(FREND1,   0x56)
Self._write_reg(MCSM0, 0x18)
Self._write_reg(MCSM1, 0x0C)
Self._strobe(SFRX)
Self._strobe(SCAL)
Sleep_ms(2)
Self._strobe(SRX)
Final_marc_state = 0x00
```

For attempt in range(10):

Current_marc_state = self._read_status_reg(MARCSTATE) & 0x1F

If current_marc_state == 0x0D:

Final_marc_state = current_marc_state

Break

Elif current_marc_state == 0x01:

Self._strobe(SRX)

Elif current_marc_state == 0x08:

Pass

Else:

Print(f"MARCSTATE is 0x{current_marc_state:02X} (unexpected) on check {attempt+1}.")

Final_marc_state = current_marc_state

If attempt < 9:

Sleep_ms(1)

Else:

Print(f"ERROR: MARCSTATE did not settle to RX (0x0D) after polling. Last state: 0x{final_marc_state:02X}.")

Print("Attempting full recovery sequence (IDLE, SFRX, SCAL, SRX)...")

Self.idle()

Self._strobe(SFRX)

Self._strobe(SCAL)

Sleep_ms(2)

Self._strobe(SRX)

Final_marc_state = self._read_status_reg(MARCSTATE) & 0x1F

If final_marc_state == 0x0D:

Sleep_ms(100)

Self.current_mode = "SNIFF_RX"

Else:

Print(f"CRITICAL ERROR: CC1101 failed to enter RX mode. Final MARCSTATE: 0x{final_marc_state:02X}")

Self.current_mode = "ERROR_SNIFF_CONFIG"

Def configure_for_ask_tx(self, pa_table_val=0xC0):

Self.reset()

Self.idle()

Self.set_frequency_mhz(base_frequency, channel)

Self._write_reg(PATABLE_ADDR, pa_table_val)

Self._write_reg(MDMCFG2, 0x30) # ASK/OOK

Self._write_reg(DEVIATN, 0x00)

Self._write_reg(PKTCTRL0, 0x02) # Synchronous serial mode. Packet handler off.

Self._write_reg(PKTLEN, 1)   # Set a dummy packet length. Required when PKTCTRL0[1:0] != 0b11.

Self._write_reg(MDMCFG4, 0xA8)

Self._write_reg(MDMCFG3, 0x93)

Self._write_reg(FSCTRL1, 0x06)

Self._write_reg(MCSM1, 0x01)   # After TX (pulse): Go to FSTXON

Self._write_reg(MCSM0, 0x09)   # XOSC alw

Python Radio 4: Sending Music

Simon Quellen Field

Simon Quellen Field

Follow

3 min read

Aug 21, 2024

Listen

Share

More

Or at least ringtones…

Press enter or click to view image in full size

MidJourney

In the previous project:

Python Radio: Simple Beginnings

The simplest digital radio mode

Medium.com

We used the PWM feature of the ESP8266 to send a 1,000 Hertz tone over the air to form the dots and dashes of Morse code.

That was as high a frequency as our little ESP8266 can manage with the PWM feature (unlike its big brother, the ESP32 which can manage up to 40 megahertz).

But we can send tones of a lower pitch if we like. This allows us to play the kind of music we used to hear on early computer games and phones.

Ringtones contain notes that are higher than 1,000 Hertz, but we can shift them two octaves down easily by dividing the frequency by 4. The program to send them over the air using the same simple FS1000A and XY-MK-5V circuits that we used when sending Morse looks like this:

PIN_D4   = 2

Key = PWM(Pin(PIN_D4))

Def play_tone(freq, msec):

If freq > 0:

Key.freq(int(freq/4)) # Set frequency (divide by 4 because we can’t go higher than 1000 Hz)

Key.duty(512) # 50% duty cycle

While(True):

With open(’tunes.txt’) as f:

For song in f:

Print(song.split(”:”)[0])

Tune = RTTTL(song)

For freq, msec in tune.notes():

Play_tone(freq, msec)

Sleep(1)

We create a function play_tone() that uses the PWM pin to send the note or a quiet period.

The main code loops forever, opening the file tunes.txt from the file system in flash on the ESP8266.

That file contains several ringtones, one per line, in the RTTTL format.

All of the files needed can be found here: . There is a setup.cmd file to put MicroPython onto the ESP8266 and copy the files onto it.

Set comport=%1

Esptool –port %comport% erase_flash

Esptool.exe –port %comport% write_flash -fm dio -fs 16MB 0 firmware.bin

Send_files %comport%

Putty -load %comport%

To use it, save it as setup.cmd and type:

Setup com3

If you just wish to repurpose the previous project, you can simply use the send_files.cmd script to put the files in place, since MicroPython will already be loaded onto the board.

Set comport=%1

Ampy -p %comport% put webrepl_cfg.py

Ampy -p %comport% put boot.py

Ampy -p %comport% put main.py

Ampy -p %comport% put rtttl.py

Save it as send_files.cmd and type:

Send_files com3

The tunes.txt file contains a short list (about 39) of ringtones, selected from a list of 10,000 of them that can be found here: .

#.,2g,2f#,f.,2e,d#.,d.,2c#,1b5,1c,1c#,1d#,1e,2f#,2d#,1e,2f#,2d#,2e,2d#,2e,2d#,2e,2d#,2e,2d#,1e.

.,e.,8c,1d,2a5,2c6,2d6,8d,d,f,a.,8g,8a,2g.,8p,8f,8g,f.,d,c,d,2d,8f,8g,2f.,8d,c,8d,1d,2a5,2c6,d6

,8p,8f,p,f,8c,d,8c,f,f,8p,8f,p,8c,d,8c,f,1p,8p,8f,f,f,8p,8f,p,8c,d,8f,1p,p,8p,8g,1p,p,8p,g#.,1f

4p,p,a,c6,a,f,a,4f,2g#.,4p,p,a,c6,a,f,a,4f,4g#,4p,c,d,f,d,4g#,d,2f,4p,p,d#6,d#6,p,d#6,d#6,p,2f6

6,b=225:a#.5,g.,d#.,a#.5,c.,g#.,1f.,d.,a#.,g.,f.,d#.,c.7,1g#.,a#.5,g.,d#.,a#.5,c.,f.,d.,f.,1d#.

,16g#6,16d#7,8c#7,16p,8d#7,16c#7,16a#6,16p,16a#6,8p,16f#6,16g#6,16d#7,8c#7,16p,8d#7,16c#7,16a#6

p,32p,32e,p,32p,8e,8d,4p,8b5,8e,8g,p,32p,4a,8g,8f#,2e,8p,p,8e,8p,8e,4b,4a,8e,4c,2b5,b,g,e,c,2b5

16g#6,8d#7,8c#7,16f#7,16g#7,8d#7,8c#7,8d#7,16a#6,16c#7,16c#7,16f#6,16g#6,16f#6,16f#6,16c#6,8c#6

,f#,4g,32p,16g,32g,32f#,e,4d#,e,f#,4b5,c#,d#,4e,d,c,4b5,a5,g5,32f#5,32e5,32f#5,4f#5,32p,g5,2g.5

#.5,b5,32a#5,8g#.5,d#,32c#,e.,d#.,c#,32c#,d#,32d#,a5,32a5,d#,c#.,32f#,f#,32f#,f#.,f.,d#,f.,4g#5

g,e,g,a,g,2e,2d,e,d,1c,c.,8c,e,g,1c6,a.,8a,c6,a,1g,g,g,8e,8e,8g,8g,a,g,2e,d,8e,8f,e,8e,8d,2c,2p

,p.,16a#,p.,2c6.,p.,16c6,p.,16c#6,p.,16c6.,p.,16a#,p.,4c6.,p.,16a#,p.,16g#,p.,4a#.,16g,16f.,d#.

a6,16p,16d6,16p,16d6,16d6,16d6,16d6,16d6,16d6,16d6,16a,16p,16c6,16d6,16p,16d6,16c6,16b,16a#,16a

00:8f#6,16p,16d#6,16p,16c#6,16p,8f#6,16p,8f#6,16p,16d#6,16p,8c#6,8p,16p,16e#6,16p,8f#6,16p,8f#6

,d,d,c,d,d,c,16d#,d#,d,d,c,c,a5,a5,c,d,c,16c,4g5,d,d,c,d,d,c,16d#,d#,16d,c,c,a5,a5,c,c,f,16g,4f

16e.,8e,p,16d#.,8e,p,4g#.,8c#,4d#,2e.,16p,p,8c#,p,16b.5,8c#,p,4b5,4e,16b.5,4a5,8g#5,p,4e5,2f#.5

16p,16g#6,16a#6,16a#6,16d#7,16c#7,16c#6,16c#6,16c#7,16a#6,8g#6,16p,16c#6,16d#6,16e#6,16a#6,8g#6

p,f,f,e,e,d,d,p,4d.,d,e,4f.,p,f,f,e,e,d,d,p,4d.,d,e,4f.,p,f,f,e,e,d,d,p,4d.,d,e,4f.,p,f,f,e,e,d

#5,16a#5,16a5,16g#5,32f#5,16d#5,16d#5,32f#5,32f5,32c#5,16d#5,16a#5,16a5,16g#5,32f#5,16d#5,16d#5

,e,8p,8f,8g,8p,1c6,p,d6,8p,8e6,1f6,g,8p,8g,e.6,8p,d6,8p,8g,e.6,8p,d6,8p,8g,f.6,8p,e6,8p,8d6,1c6

,8d#5,32g#4,32c#.5,a#4,16c#5,16e5,32c,8b5,32e5,32g.5,g#5,32e.5,32f#5,32e5,8d#5,32g#4,32c.5,c#.5

b=100:c,c,c,8f.,a,p,c,c,c,8f.,a,4p,f,f,e,e,d,d,8c,p,c,c,c,8e.,g,p,c,c,c,8e.,g,4p,c6,d6,c,a,g,8f

25:4d6,g,a,b,c6,4d6,4g,4g,4e6,c6,d6,e6,f6,4g6,4g,4g,4c6,d6,c6,b,a,4b,c6,b,a,g,4f#,g,a,b,g,4b,4a

,p,4g,p,4g6,d6,g,4p,4d,p,4f,p,4g,p,4a,p,4g,p,4g6,d6,g,4p,4d,p,4f,p,4g,p,d6,16p.,d6,2p.,4a,a,4g6

5:16g.5,16g.5,16g5,16g.5,32e5,8g.5,16e5,32d5,16e5,8g.5,32d,8b.5,32f#5,16e5,32f#5,16g5,32b5,32a5

2f.,32f#,32g#.,32f#.,8f.,32f#.,32g#.,32a.,32b.,32a.,32g#.,32f#.,32f#.,32f.,32f#,32g#.,32f#.,8f.

2d7,4b.7,g7,2g7,2g7,2e7,g5,b5,d,4g,g5,4f,g5,4c,g5,16c,d.,4f,g5,b5,d,4g,g5,4f,g5,4c,g5,16c,d.,4f

6,f6,e6,d6,c6.,32p,4c6,d6,e6,f6,e6,d6,c6,d6,c6,a#,a,a#,a,g,f,g,f,e,d,c,d,e,f,g,a#,a,g,4a,4f,4f.

8a5,8b5,c.,16d5,16p,8c,8c,8b5,8c,8b5,8a5,8p,a5,16d5,16p,8a5,8b5,c.,16d5,16p,8c,8c,8b5,8c,8b5,d5

5,4f5,4p,4c5,4d#5,4f.5,4d#5,4c5,2p,8p,4c5,4d#5,4f.5,4c5,4d#5,8f#5,4f5,4p,4c5,4d#5,4f.5,4d#5,4c5

f.,8d.,16p,p.,8a,8p,8d6,8p,8a,8p,8d6,8p,8a,8d6,8p,8a,8p,8g#,8a,8p,8g,8p,g.,8f#,8g,8p,8c6,a#,a,g

,8p.,g,8p,p,g,g#,g#,g#,4p,g,g#,g#,g#,4p,c6,8p.,a#,8p,p,g#,8p.,g,8p,p,g#,g#,g#,2c.6,p,1a#,1g#,1g

,16g6,8p,16a#6,16p,16c7,16p,16a#6,16g6,2d6,32p,16a#6,16g6,2c#6,32p,16a#6,16g6,2c6,16p,16a#,16c6

d=4,o=6,b=200:8b,8e,8a,8e,8b,8e,8g,8a,8e,8c7,8e,8d7,8e,8b,8c7,8e,8b,8e,8a,8e,8b,8e,8g,8a,8e,8c7

,d#,d#,8p,d#,d#,8p,d#,p,d#,f,p,g,8p,g,g,8p,g,g,8p,g,p,g,f,p,d#,8p,d#,d#,8p,d#,d#,8p,d#,p,d#,f,p

,8f#.,8d.,d,8d,d,e,p,e,p,e,p,8d.,8b5,d,d,d,d,d,e,p,e,p,e,p,8f#.,8d.,d,8d,d,e,p,8f#,a,p,8g#.,8e.

,f,16g,c,c,16d#,16f,g,16g,a#,4g,16d#,c,d#,d#,c.,a#,16g,16f,4g.,16d#,4c,d#,d#,16c,4c,d#,16d#,2d#

5,b5,c,c,c,a#5,g5,c,e,f,g,g,g,f,d,d,g,g,f,f,f,d#,c,g5,a#5,b5,c,c,c,a#5,g5,g5,c,a#5,g5,g5,g5,4g5

6,8f#6,16d#6,8f#6,16g#6,2e6,16e6,16d#6,16c#6,16d#6,16e6,8g#6,16b6,8g#6,a#6,g#6,f#6,e6,8d#6,8c#6

Transmitter

Projects

Ringtones

Python Programming

Micropython

Python Radio 3: Text to Morse Code

Simon Quellen Field

Simon Quellen Field

Follow

5 min read

Aug 20, 2024

Listen

Share

More

Send text over the air with our little radio transmitter

Press enter or click to view image in full size

MidJourney

In our first project:

Python Radio: Simple Beginnings

The simplest digital radio mode

Medium.com

We had fun sending Morse code with the key by hand, but since we have a computer, why not let it translate the keys on the keyboard into Morse code?

We don't need to change the hardware, except to get rid of the key.

We will first need a data structure to hold the Morse code table. A Python dictionary is just the thing:

Code = {

'A': '.-',

'B': '-...',

'C': '-.-.',

'D': '-..',

'E': '.',

'F': '..-.',

'G': '--.',

'H': '....',

'I': '..',

'J': '.---',

'K': '-.-',

'L': '.-..',

‘M’: ‘—’,

‘N’: ‘-.’,

‘O’: ‘---‘,

‘P’: ‘.--.’,

‘Q’: ‘--.-‘,

‘R’: ‘.-.’,

‘S’: ‘...’,

‘T’: ‘-‘,

‘U’: ‘..-’,

‘V’: ‘...-’,

‘W’: ‘.—’,

‘X’: ‘-..-’,

‘Y’: ‘-.—’,

‘Z’: ‘--..’,

We will put that into a file called the_code.py.

Class Morse:

Character_speed = 18

Def __init__(self, pin):

Self.key = PWM(Pin(pin, Pin.OUT))

Self.key.freq(300)

Def speed(self, overall_speed):

If overall_speed >= 18:

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 >= 18:

Self.LETTER_SPACE = int(3 * self.DOT) – self.CYPHER_SPACE

Self.WORD_SPACE = int(7 * self.DOT) – self.CYPHER_SPACE

Else:

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):

For c in str:

If c == ‘ ‘:

Self.key.duty(0)

Sleep_ms(self.WORD_SPACE)

Else:

Cyphers = code[c.upper()]

For x in cyphers:

If x == ‘.’:

Self.key.duty(512)

Sleep_ms(self.DOT)

Else:

Self.key.duty(512)

Sleep_ms(self.DASH)

Self.key.duty(0)

Sleep_ms(self.CYPHER_SPACE)

Self.key.duty(0)

Sleep_ms(self.LETTER_SPACE)

The main routine is fairly simple:

Def main():

PIN_D4 = 2

Morse = Morse(PIN_D4)

Print(”Morse code transmitter”)

While(True):

Wpm = input(”How many words per minute? “)

If int(wpm) > 0 and int(wpm) < 50:

Morse.speed(int(wpm))

Str = input(”Enter the message to send: “)

Morse.send(str)

Else:

Print(”Try a more reasonable speed.”)

Main()

We put it into main.py

When all the files are loaded onto the ESP8266 (using the send_files.cmd script), we are ready to run it.

The terminal emulator connects to the USB port that powers the ESP8266:

Putty -load com3

At first, the putty screen is blank. If you hit the Enter key, you get a prompt from the REPL of three greater-than signs. To do a soft reboot (which then runs the program) type a control-D.

Here is what the screen looks like after entering a speed and a message:

MPY: soft reboot

WebREPL server started on

Started webrepl in normal mode

Morse code transmitter

How many words per minute? 20

Enter the message to send: Hello

How many words per minute?

Radio Transmitter

Python Programming

Morse Code

Python Radio 35: You've got mail…

Follow

11 min read

Mar 18, 2025

Listen

Share

More

Get a Wi-Fi notification of snail-mail

Press enter or click to view image in full size

Image of A.G. Bell's telephone patent

The transmitter and receiver used the same design. An electromagnet sat next to a diaphragm that had a bit of metal attached. As the metal moved back and forth in the magnetic field, it induced a signal in the coil of the electromagnet.

When this signal reached the receiver, the magnetic field in the coil varied with the sound, moving the metal and thus the diaphragm, producing sound.

These days, we have speakers that operate on a similar principle. A thin coil of wire is attached to a diaphragm. As the coil moves back and forth next to a permanent magnet, an electric signal is produced.

If we connect two speakers together with a pair of long wires, one person can talk into one speaker, and another person can hear the voice in the second speaker. Just like the tin-can telephones we made as kids.

Using a speaker as a microphone

We can use this trick to make our tiny computers sensitive to sound.

We connect one speaker terminal to the ground pin of the microcontroller.

We connect the other pin to an analog input pin.

The microcontroller.

For this project, I chose the ESP32-C3 Supermini. Mainly because it is so darn cute.

Press enter or click to view image in full size

The ESP32-C3 Supermini

That tiny screen is 0.42 inches diagonally (10.66 mm). It can display 9 characters across four lines.

Why on earth did you do that?

I live on a 20-acre hobby farm called The Lakeview Birdfarm. We have a lot of birds. Including an emu.

But all that space means that my mailbox is 600 feet from the main house. That's 2 American football fields. Almost two if we're talking about the football the rest of the world thinks of.

The mailbox is a big steel box with a door that slams itself shut, making a nice loud noise. So a computer that can radio home when it hears a loud noise would be a nice thing to park on top of the mailbox.

The code in the Supermini

The display on the Supermini uses the SH1106 driver. I'll add that later without discussion, but it is available on GitHub.

Next, we set up the Wi-Fi connection. We have excellent Wi-Fi on the farm, so the mailbox has good coverage. The SSID is BirdfarmOffice2, and there is no password. We let anyone passing by use our 5 gigabit Internet connection. Our neighbours are a long way away.

The server in my office has the IP address 10.90.20.10. We will run a Python server on that machine listening on port 8143.

When the Supermini hears a noise, it prints Hit! (although nobody will be reading the serial port). Then it connects to the server using requests.get().

We don't need to send any data, we just want to connect. The server will know that a connection means there has been a noise.

We also flash the blue LED, so I can tell if the noise was loud enough without running 600 feet back to my office to look at the server output.

Then main() listens to the speaker. The read_uv() method returns microvolts, even though the last three digits are always zero, so it could have been millivolts.

Here is where you tune the volume we need to hear to decide to send an alert. By dropping the whole project on the desktop from about half an inch up, I chose 50,000 microvolts. I may change that if I am getting to many or too few alerts.

The Server

The server side is a little simpler because the Microdot module does all the heavy lifting. You install it using "pip install microdot".

If a connection comes in on port 8143 it prints a message with a timestamp.

The output looks like this:

C:\simon\mailbox_alarm>server.py
2025-03-16 13:35:06.311245 Device at 10.90.20.74 heard a loud noise. You may have mail.

2025-03-16 13:35:12.346001 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-16 13:35:15.622433 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-16 13:42:19.284935 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-16 15:23:26.683053 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-16 15:50:36.045345 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-16 17:23:06.027225 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-16 17:23:09.308156 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-16 17:23:34.948135 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-16 17:23:37.562645 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-16 17:23:44.837076 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-16 17:23:46.984213 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-16 17:26:10.905210 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-17 09:24:51.373658 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-17 13:08:23.083649 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-17 16:25:10.423883 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-17 16:54:01.483339 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-17 16:54:39.730194 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-17 16:55:16.245313 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-17 16:55:39.365604 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-17 16:55:59.530717 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-17 16:56:01.336591 Device at 10.90.20.74 heard a loud noise. You may have mail.
2025-03-17 17:10:04.631543 Device at 10.90.20.74 heard a loud noise. You may have mail.

Python

Micropython

Radio

Demystifying SDR Hacking: A Deep Dive into Wireless Protocols Part:3

KISHORERAM

KISHORERAM

Follow

7 min read

Oct 7, 2023

Listen

Share

More

Aircraft Communication

Press enter or click to view image in full size

Photo by Bao Menglong on Unsplash

Common Aircraft Communication Frequencies

TWR (Tower Frequency: 118.15 MHz): This frequency is used by the control tower at an airport to provide instructions to aircraft in the vicinity of the airport.

HDSDR

Press enter or click to view image in full size

To listen to aircraft communication using HDSDR, follow these steps:

Open HDSDR and set the mode to AM.

Set the frequency manager to Air.

Enter the frequency of the aircraft communication that you want to hear.

Click the Start button to start listening.

You should now be able to hear the aircraft communication. You may need to adjust the volume and other settings to get the best sound quality.

Replay Attacks in Wireless Devices with RTL-SDR

Wireless devices such as bells, switches, and car remotes have become an integral part of our daily lives due to their convenience. However, these devices can be susceptible to a type of security vulnerability known as a replay attack.A replay attack involves capturing a valid transmission and retransmitting it at a later time. This could potentially allow unauthorized access or control over the wireless device.For replaying first we need to identify the frequency it uses. This can be achieved using an RTL-SDR dongle, a cost-effective device capable of receiving radio signals. Once the frequency is identified, the signal can be recorded using the RTL-SDR dongle. After the signal has been recorded, it can be replayed using a Raspberry Pi as a transmitter.However, it’s important to note that many modern wireless devices employ security measures to prevent such attacks. One such measure is rolling code security, which is commonly used in car remotes. This security feature changes the code transmitted by the remote every time it’s used, preventing unauthorized individuals from replaying the signal to gain access.

Press enter or click to view image in full size

Source:https://www.pngegg.com/

Jamming Signals Using a HackRF One

What is Jamming?

Press enter or click to view image in full size

Source:https://www.everythingrf.com/

To jam signals using a HackRF One, you will need the following:

A HackRF One

GNU Radio

The gr-osmosdr package

The osmocom_siggen_nogui utility

Commands

Sudo apt install gnuradio

Sudo apt install gr-osmosdr

Osmocom_siggen_nogui -h

The following command will generate a continuous wave signal at 100 MHz with a power of 10 dBm:

Osmocom_siggen_nogui -a hackrf -f 100e6 –sweep -x 2e6 -y 10 -v

Osmocom_siggen_nogui: This is the command to run the signal generator application. Osmocom-siggen is a versatile signal generator tool that can be used to generate a variety of signals, including constant waves, sinusoidal signals, uniform noise, Gaussian noise, frequency sweeps, GSM bursts, and two-tone signals.

-f 100e6: This option sets the frequency to 100 MHz. The frequency of the signal is the number of times per second that the signal oscillates. The frequency of the signal you want to jam will depend on the type of signal you are targeting. For example, if you are targeting a GSM signal, you would need to set the frequency to 900 MHz.

- sweep: This option indicates that a frequency sweep will be performed. This means that the signal will sweep through a range of frequencies. Frequency sweeps can be used to interfere with a wider range of signals, but they can be less effective at jamming specific signals.

-x 2e6: This option specifies the start frequency of the sweep, which is 2 MHz.

-y 10: This option sets the stop frequency of the sweep, which is 10 MHz.

-v: This option enables verbose mode, providing additional information about the signal generation process.

To jam a GSM signal at 900 MHz, you would use the following command:

Osmocom_siggen_nogui -a hackrf -f 900e6 –sweep -x 890e6 -y 910e6 -v

This command would sweep the signal from 890 MHz to 910 MHz, which would interfere with the GSM signal at 900 MHz.

Limitations

The primary limitation of jamming signals using a HackRF One is the strength of the signal. The HackRF One is a relatively low-power device, so it can only jam signals that are close to the receiver. Additionally, the HackRF One is susceptible to interference from other radio signals, so it may not be effective in noisy environments. It is important to note that jamming signals is illegal.

GPS Spoofing with HackRF One: A Deep Dive into Location Deception

Press enter or click to view image in full size

Photo by Tobias Rademacher on Unsplash

Step #1: Install HackRF One

Step #2: Install GPS Spoofing Software

Create a Directory:

Kali > mkdir GPS_SPOOF

Navigate to the New Directory:

Kali > cd GPS_SPOOF

Download GPS Spoofing Software:

Kali > sudo git clone

Navigate to the Software Directory:

Kali > cd gps-sdr-sim

Compile the Software:

Compile the gpssim.c file to create an executable named gps-sdr-sim:

Kali > sudo gcc gpssim.c -lm -O3 -o gps-sdr-sim -DUSER_MOTION_SIZE=4000

Step #3: Locate the Satellite

Proceed with GPS spoofing, you need information about GPS satellites positions. This information is obtained from GPS broadcast ephemeris files, which can be downloaded from sources like NASAs CDDIS archive. Make sure to download the most recent daily file.

NASA CDDIS Ephemeris Files:

Select a Location. Choose the location you want to spoof. You can use services like Google Maps to obtain GPS coordinates.

Start GPS Spoofing

To initiate GPS spoofing, use the following command, providing the ephemeris file and GPS coordinates:

Kali > sudo ./gps-sdr-sim -b 8 -e <ephemeris_file> -l <latitude>, <longitude>, <altitude>

This command creates a simulation file named gpssim.bin containing spoofed GPS data.

Transmit the Spoofed GPS Signal

Now, you can transmit the spoofed GPS signal using the HackRF One:

Kali > sudo hackrf_transfer -t gpssim.bin -f 1575420000 -s 2600000 -a 1 -x 0

This command sends the spoofed GPS signal, making any GPS receiver tracking it believe that it's at the specified location.

GPS spoofing can be used to conceal our location and prevent tracking by governments and malicious actors.

Thanks For Reading

If you have any further questions or would like to connect, feel free to reach out to me

My LinkedIn handle:

Demystifying SDR Hacking: A Deep Dive into Wireless Protocols Part:6

Follow

10 min read

Oct 14, 2023

Listen

Share

More

HackrfOne

Source:

Firmware Update HackrfOne

https://github.com/mossmann/hackrf/releases/
taf –xvf (upzip file)
cd firmware-bin
hackrf_spiflash -w hackrf_one_usb.bin
hackrf_cpldjtag -x firmware/cpld/sgpio_if/FILENAME.xsvf

SSTV Broadcast

Slow Scan Television (SSTV) is a method used by ham radio operators to send images over radio frequencies. With a Raspberry Pi and HackRF One, you can set up your own SSTV station! The Raspberry Pi can be used to transmit SSTV signals. A project called Pi-SSTV used to transmit images in the SSTV.

Press enter or click to view image in full size

Projects

https://github.com/AgriVision/pisstv
sudo apt-get install python-setuptools
sudo apt-get install python-imaging
sudo easy_install pip
sudo pip install setuptools --no-use-wheel --upgrade
sudo pip install PySSTV
sudo apt-get install libgd2-xpm-dev
sudo apt-get install libmagic-dev
gcc -lm -lgd -lmagic -o pisstv pisstv.c
sudo ./pisstv image.png 22050

DragonOS

Press enter or click to view image in full size

https://cemaxecuter.com/ DOWNLOAD USING THIS LINK

LimeSDR

LimeSDR Types

LimeSDR-Mini: This is a software-defined radio (SDR) board that uses a USB3 interface.

LimeSDR-USB: Similar to the Mini, this SDR board also uses a USB3 interface.

LimeNET-Micro: This SDR board comes with a Raspberry Pi (Compute Module) CM3 and uses a USB2 interface.

LimeSDR-PCIe: This SDR board uses a PCIe (1.0 x4) interface.

LimeSDR-QPCIe: This SDR board also uses a PCIe (1.0 x4) interface but comes with two LMS7002M transceivers.

Press enter or click to view image in full size

GSM NETWORK

BTS (Base Transceiver Station) A BTS is like a Wi-Fi access point that communicates with a centralized controller, the BSC (Base Station Controller). The BTS handles the transmission and reception of baseband data, getting most of its commands from the BSC.

Media Gateway (MGW) A Media Gateway is a translation device or service that converts media streams between different telecommunications technologies such as 2G, 2.5G, 3G. One of its main functions is to convert between different transmission and coding techniques.

Press enter or click to view image in full size

Source:

Communication between SIM and HLR

Press enter or click to view image in full size

Source:

The Home Location Register (HLR) is a functional unit that manages mobile subscribers. It stores data such as:

International Mobile Subscriber Identity (IMSI)

Mobile Subscriber ISDN Number (MSISDN)

Authentication keys

Service profiles

The HLR communicates with the SIM card in the following ways:

The HLR identifies the last known location of the device.

The HLR transfers the list of services to VLR/MSC.

The roaming network uses the information to allow or disallow the call.

The HLR updates the VLR address when the subscriber moves from one VLR to another.

Each mobile network operator has its own HLR.

Iridium Satellites

Iridium satellites were built by Motorola .There are 66 active satellites across the globe which covers the entire Earth surface .These satellites are in low Earth orbit at a height of approximately 781 kilometers . The Iridium system was launched on November 1, 1998, and it has changed global communications. The Iridium satellites provides L band voice and data information.The satellites are cross-linked and operate as a fully meshed network. They are the largest constellation and orbit closer to Earth than other networks. The Iridium network covers the entire Earth, including poles, oceans, and airways.

Press enter or click to view image in full size

Source:

The Iridium satellites have the following characteristics:

They are low-earth orbiting (LEO)

They orbit in an 86.4° inclined orbit

They take about 100 minutes to orbit the Earth, and about 10 minutes from horizon to horizon

They use GSM-based telephony architecture

They provide global roaming

https://www.iridium.com/

Press enter or click to view image in full size

Services Provided by Iridium Satellites

Paging

Global Burst Data

Voice / SMS

Short Burst Data

Time and Locations services

Applications of Iridium Satellite

Tracking

Mobile Data/Voice

Emergency Services

Aircraft communication

Covert Operations

Receiving data using gr-iridium

GitHub - muccc/gr-iridium: Iridium burst detector and demodulator.

Iridium burst detector and demodulator. Contribute to muccc/gr-iridium development by creating an account on GitHub.

github.com

iridium-extractor -D 4 DEVICE.conf | grep "A:OK" > FILE_NAME.bits

Decoding data using iridium toolkit

GitHub - muccc/iridium-toolkit: A set of tools to parse Iridium frames

A set of tools to parse Iridium frames. Contribute to muccc/iridium-toolkit development by creating an account on…

github.com

pypy iridium-parser.py -p FILE_NAME.bits > FILE_NAME.parsed

Decoding Voice data using iridium toolkit

GitHub - muccc/iridium-toolkit: A set of tools to parse Iridium frames

A set of tools to parse Iridium frames. Contribute to muccc/iridium-toolkit development by creating an account on…

github.com

Decoding other data using iridium toolkit

GitHub - muccc/iridium-toolkit: A set of tools to parse Iridium frames

A set of tools to parse Iridium frames. Contribute to muccc/iridium-toolkit development by creating an account on…

github.com

sudo ./reassembler.py -i FILE_NAME.parsed -m <mode>
ida - outputs Um Layer 3 messages as hex
lap - GSM-compatible L3 messages as GSMtap compatible .pcap
page - paging requests (Ring Alert Channel)
msg - Pager messages

Press enter or click to view image in full size

Inmarsat Satellite

Press enter or click to view image in full size

Source:

Inmarsat System

The Inmarsat System is a complex network of components that work together to provide global mobile services.

Satellite Control Center (SCC): The SCC manages the satellites in the Inmarsat system. It controls the positioning of the satellites and monitors their health and performance.

Network Coordination Station (NCS): The NCS keeps track of all Inmarsat C transceivers in its region and broadcasts information such as navigational warnings, weather reports, and news. There is one NCS in each region.

Mobile Earth Station (MES): The MES is a portable or mobile terminal that communicates with the Inmarsat satellites. It can be used to send and receive voice and data services.

Land Earth Station (LES): The LES provides the link between the MES and the terrestrial telecommunications networks via satellite. There are several LESs in each region.

Projects

Press enter or click to view image in full size

Source:

The Inmarsat C system divides the world into four regions and each region is covered by its own satellite.

In each region, there is one NCS and several LESs. The NCS keeps track of all Inmarsat C transceivers in its region and broadcasts information such as navigational warnings, weather reports, and news. The LESs provide the link between the MES and the terrestrial telecommunications networks via satellite.

Setup Inmarsat Decoding with Scytale-C

Data Receiving with SDR Sever
https://airspy.com/directory/

Press enter or click to view image in full size

Press enter or click to view image in full size

Downloads

Edit description

bitbucket.org

Select Enabled and also Auto Tracking option.
Decoding Data of Inmarsat with scytale
In radio use USB and BW of 4,000 and make sure snap to grid is checked.

Press enter or click to view image in full size

Press enter or click to view image in full size

WebSDR

Press enter or click to view image in full size

Press enter or click to view image in full size

GMRS COIMBATORE -

List of active webSDRs available in India

Thanks For Reading :)

If you have any further questions or would like to connect, feel free to reach out to me

My LinkedIn handle:

Hacking

Cybersecurity

Blog

Demystifying SDR Hacking: A Deep Dive into Wireless Protocols Part:2

Follow

7 min read

Oct 5, 2023

Listen

Share

More

Projects

Signal hunting

Installing SDR#
Download SDR#
Pre-requisites: .NET 5 Desktop x86 Runtime
ps://dotnet.microsoft.com/download/dotnet/thank-you/runtime-desktop-5.0.2-windows-x86-installer

Frequencies and SDR Servers
Check different frequencies around the World

SDR Servers
/

Install VB-CABLE
Download:
Virtual Audio Cable is a software that allows a user to transfer audio streams from one application to another.

Raspberry PI Installation
Tool Required
Raspberry PI OS:
Angry IP Scanner:
Putty:
Setup SSH and Wi-Fi Connection on Raspberry PI

Raspberry Pi (SDR Server & Transmitter)

Transmit Radio Signals with Raspberry Pi: Radio Data System data generated in real time

sudo apt-get install libsndfile1-dev

git clone

cd PiFmRds/src

make clean

make

Command

sudo ./pi_fm_rds [-freq freq] [-audio file] [-pi pi_code] [-ps ps_text] [-rt rt_text]

Source:

VNC on Raspberry Pi

sudo apt-get update
sudo apt-get install realvnc-vnc-server

Listen Radio

RTL-SDR is a popular software-defined radio dongle that can be used to receive a wide range of radio signals, including FM radio stations. To listen to FM radio using an RTL-SDR dongle, you will need to connect the dongle to your computer and install SDR software, such as SDR# or Gqrx. Once you have installed the software, you can tune to the frequency of the radio station that you want to listen to and click the “Play” button.

Press enter or click to view image in full size

Some of the frequencies of popular Tamil FM radio stations:

Radio City: 91.1 MHz

Aahaa FM: 91.9 MHz

Suriyan FM: 93.5 MHz

Radio Mirchi: 98.3 MHz

Hello FM: 106.4 MHz

You can listen FM in online in many websites

Aircraft Tracking

Press enter or click to view image in full size

Press enter or click to view image in full size

Press enter or click to view image in full size

Press enter or click to view image in full size

Press enter or click to view image in full size

Press enter or click to view image in full size

Dump 1090

Virtual Radar Server

Virtual Radar Server

A stand-alone .NET application that displays output from an SBS-1 ADS-B receiver on a Google Maps web page.

www.virtualradarserver.co.uk

Listen International Space Station

The International Space Station (ISS) is a large artificial satellite orbiting the Earth as a microgravity and space environment research laboratory. It is jointly owned and operated by the United States, Russia, Japan, Europe, and Canada. The ISS runs an amateur radio service under the Amateur Radio on International Space Station (ARISS) program. This service enables amateur radio operators on Earth to communicate with astronauts aboard the ISS. ARISS also transmits slow scan television (SSTV) signals, which can be received by amateur radio operators using relatively inexpensive equipment. To receive ARISS SSTV signals, you will need an amateur radio receiver and a computer with SSTV decoding software.

Frequency: 145.800MHz

Determining When the Space Station is Overhead

To find when the International Space Station will be over your location, you can go to

MMSSTV (Decoding ISS Data)

Press enter or click to view image in full size

ISS Detector App(Playstore)

Press enter or click to view image in full size

Press enter or click to view image in full size

Source: (Audio file)

Upload your received and decoded image

Claim your SSTV reception award

Use virtual cable audio input in the MMSSTV and also choose the audio output as virtual cable from which software you are using.

Projects

Press enter or click to view image in full size

MMSSTV Output

Press enter or click to view image in full size

Audio player Input

Thanks For Reading :)

If you have any further questions or would like to connect, feel free to reach out to me

My LinkedIn handle:

Demystifying SDR Hacking: A Deep Dive into Wireless Protocols Part:4

KISHORERAM

KISHORERAM

Follow

8 min read

Oct 14, 2023

Listen

Share

More

GSM HACKING

Understanding GSM Architecture:

Press enter or click to view image in full size

Photo by Kabiur Rahman Riyad on Unsplash

GSM comprises three essential subsystems:

Base Station Subsystem (BSS): BSS manages traffic and signaling between mobile phones and the network switching subsystem. It consists of two key components, the Base Transceiver Station (BTS) and the Base Station Controller (BSC).

Press enter or click to view image in full size

Components of GSM Subsystems:

Base Transceiver Station (BTS): BTS, found in every tower, facilitates wireless communication between user equipment and the network.

Base Station Controller (BSC): BSC serves as a local exchange, managing multiple towers and their respective BTS units.

Mobile Switching Center (MSC): MSC handles communication switching functions, including call setup, release, and routing, as well as services like call tracing and call forwarding. VLR, HLR, AUC, EIR, and PSTN are integral components of MSC.

Visitor Location Register (VLR): VLR maintains the real-time location data of mobile subscribers within the service area of the MSC, ensuring seamless mobility across regions.

Home Location Register (HLR): HLR is a database containing subscriber-specific information, such as plan details, caller tunes, and identity data.

Authentication Center (AUC): AUC authenticates mobile subscribers seeking to connect to the network.

Equipment Identity Register (EIR): EIR maintains records of allowed and banned devices, ensuring network security and integrity.

Public Switched Telephone Network (PSTN): PSTN connects to MSC and has evolved from analog to a mostly digital network, encompassing mobile and fixed telephones.

Operation Maintenance Center (OMC): OMC plays a pivotal role in monitoring and maintaining the performance of mobile stations, BSCs, and MSCs in the GSM system.

Understanding GSM Sniffing and IMSI Numbers

What is GSM Sniffing?

GSM sniffing is a method used to intercept and decode the communication between mobile devices and cellular networks.

Press enter or click to view image in full size

Source :https://www.pentotest.com/

What is an IMSI Number?

The International Mobile Subscriber Identity (IMSI) is a unique number that identifies every user of a cellular network. It is usually presented as a 15-digit number. The IMSI number is crucial in GSM sniffing as it allows the identification of individual mobile devices within the network.

Sourcehttps://www.tech-invite.com/

IMSI Catcher

An IMSI Catcher, also known as an International Mobile Subscriber Identity catcher, is a device that intercepts mobile phone communications and tracks the location data of mobile phone users. It operates by posing as a fake mobile phone tower, creating a connection between the target mobile phone and the service provider’s actual towers, making it a man-in-the-middle (MITM) attack.While 3G or 4G wireless cellular networks require mutual authentication from both the handset and the network, sophisticated IMSI catchers may have the capability to downgrade 3G and LTE to non-LTE network services. These services do not require mutual authentication, making them vulnerable to interception.

Gqrx

Gqrx is a software-defined radio (SDR) receiver that can control and use a variety of SDR hardware. To install Gqrx on Debian-based systems, you can use the following command:

Sudo apt-get install gqrx

Kalibrate-rtl

Sudo apt-get install kalibrate-rtl

OR

Sudo apt-get update

Git clone

Cd kalibrate-rtl

./bootstrap && CXXFLAGS=’-W -Wall -O3’

./configure

Make

Sudo make install

To find gsm frequency

Kal -g 40 -s GSM900

Press enter or click to view image in full size

Gr-gsm

Gr-gsm is a set of tools designed to understand and demodulate GSM signal. To install gr-gsm, you can follow these commands:

Sudo apt install python3-numpy python3-scipy python3-scapy

Sudo apt-get install -y \

Cmake \

Autoconf \

Libtool \

Pkg-config \

Build-essential \

Python-docutils \

Libcppunit-dev \

Swig \

Doxygen \

Liblog4cpp5-dev \

Gnuradio-dev \

Gr-osmosdr \

Libosmocore-dev \

Liborc-0.4-dev \

Swig

Git clone

Cd gr-gsm

Mkdir build

Cd build

Cmake ..

Make

Sudo make install

Sudo ldconfig

Install IMSI Catcher :

Git clone

Cd IMSI-catcher

Sudo apt install python3-numpy python3-scipy python3-scapy

USAGE

We use grgsm_livemon to decode GSM signals and simple_IMSI-catcher.py to find IMSIs.

Python3 simple_IMSI-catcher.py -h

Usage: simple_IMSI-catcher.py: [options]

Options:

-h, --help show this help message and exit

-a, --alltmsi Show TMSI who haven't got IMSI (default : false)

-i IFACE, --iface=IFACE

Interface (default : lo)

-m IMSI, --imsi=IMSI IMSI to track (default : None, Example:

123456789101112 or "123 45 6789101112")

-p PORT, --port=PORT Port (default : 4729)

-s, --sniff sniff on interface instead of listening on port

(require root/suid access)

-w SQLITE, --sqlite=SQLITE

Save observed IMSI values to specified SQLite file

-t TXT, --txt=TXT Save observed IMSI values to specified TXT file

-z, --mysql Save observed IMSI values to specified MYSQL DB (copy

.env.dist to .env and edit it)

Capturing or Intercept of GSM traffic :

Open 2 terminals.

In terminal 1

Sudo python3 simple_IMSI-catcher.py -s

In terminal 2

Grgsm_livemon

Now, change the frequency until it display, in terminal, something like that :

15 06 21 00 01 f0 2b 2b 2b 2b 2b 2b 2b 2b 2b 2b 2b 2b 2b 2b 2b 2b 2b

25 06 21 00 05 f4 f8 68 03 26 23 2b 2b 2b 2b 2b 2b 2b 2b 2b 2b 2b 2b

49 06 1b 95 cc 02 f8 02 01 9c c8 03 1e 57 a5 01 79 00 00 1c 13 2b 2b

Grgsm_livemon -f 942.4M -g 45

IMSI Catchers: Detection Techniques

Using a signal detector, a portable device that can detect suspicious wireless signals in the vicinity. It can detect the presence of IMSI catchers by picking up on their GSM or LTE signals. Another project, known as AIMSICD (Android IMSI-Catcher Detector), attempts to detect IMSI-Catchers through various detection methods such as checking tower information consistency, checking LAC/Cell ID consistency, checking neighboring cell info, preventing silent app installations, monitoring signal strength, detecting silent SMS, and detecting FemtoCells.

While these methods can help in detecting IMSI Catchers, they may not provide complete protection against all types of IMSI Catcher attacks.

Decoding Morse Code with RTL-SDR

Morse code is a system of encoding messages using dots and dashes. It was developed in the early 1800s by Samuel Morse and Alfred Vail. Morse code is named after Samuel Morse, who was one of the inventors of the telegraph. Morse code can encode the 26 basic Latin letters A to Z, Arabic numerals, and a small set of punctuation and procedural signals.

Source:https://en.wikipedia.org/

What is RTL-SDR?

What is CwGet?

CwGet is a program that decodes Morse code (CW) via a sound card to text. It can work as a narrow-band sound DSP-filter also. No additional hardware is required — you need only receiver and computer with a sound card.

To decode Morse code using RTL-SDR and CwGet, you would first need to set up your RTL-SDR dongle to receive the desired frequency range. Then, you would use CwGet to decode the Morse code signals received by the RTL-SDR. The decoded text would then be displayed in the CwGet interface.

Press enter or click to view image in full size

Amateur Radio Frequency Bands in India

Amateur radio, often referred to as ham radio, is a popular pastime among more than 16,000 licensed enthusiasts in India. The Wireless and Planning and Coordination Wing (WPC), a division of the Ministry of Communications and Information Technology, grants licenses. The Indian Wireless Telegraphs (Amateur Service) Rules, 1978 initially listed five license categories. To earn a license, applicants must pass the Amateur Station Operator's Certificate examination administered by the WPC. Amateur radio operators in India have access to a range of frequencies. For instance, they can operate on 0.1357–0.1378 MHz (2200 m wavelength) in the LF band, 0.472–0.479 MHz (630 m) in the MF band, and 1.800–1.825 MHz (160 m) in the MF band, among others. Each band corresponds to the International Telecommunication Union (ITU) radio band designation.

Press enter or click to view image in full size

Source:https://hamradioprep.com/

Thanks For Reading

If you have any further questions or would like to connect, feel free to reach out to me

My LinkedIn handle:

Hacking

Cybersecurity

Sdr