Python Radio 9: A 40-meter CW transmitter
Simon Quellen Field
Simon Quellen Field
Follow
11 min read
.
Aug 26, 2024
Listen
Share
More
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 ha 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, 1844 1849710, 1855072, 1860465, 1865889, 1871345, 1876832, 1882352, 1887905, 1893491, 1899109, 19 1910447, 1916167, 1921921, 1927710, 1933534, 1939393, 1945288, 1951219, 1957186, 1963190, 19 1975308, 1981424, 1987577, 1993769, 2000000
80-meter band: 3506849, 3516483, 3526170, 3535911, 3545706, 3555555, 3565459, 3575418, 35854 3595505, 3605633, 3615819, 3626062, 3636363, 3646723, 3657142, 3667621, 3678160, 3688760, 36 3710144, 3720930, 3731778, 3742690, 3753665, 3764705, 3775811, 3786982, 3798219, 3809523, 38 3832335, 3843843, 3855421, 3867069, 3878787, 3890577, 3902439, 3914373, 3926380, 3938461, 39 3962848, 3975155, 3987538, 4000000
40-meter band: 7013698, 7032967, 7052341, 7071823, 7091412, 7111111, 7130919, 7150837, 71708 7191011, 7211267, 7231638, 7252124, 7272727, 7293447, 7314285
30-meter band: 10118577, 10138613, 10158730
20-meter band: 14027397, 14065934, 14104683, 14143646, 14182825, 14222222, 14261838, 143016 14341736, 14382022
17-meter band: 18091872, 18156028, 18220640
15-meter band: 21026694, 21069958, 21113402, 21157024, 21200828, 21244813, 21288981, 213333 21377870, 21422594, 21467505
12-meter band: 24914841, 24975609, 25036674
10-meter band: 28054794, 28131868, 28209366, 28287292, 28365650, 28444444, 28523676, 286033 28683473, 28764044, 28845070, 28926553, 29008498, 29090909, 29173789, 29257142, 29340974, 2 29510086, 29595375, 29681159, 29767441
Frequencies shown above in bold are available for CW to the entry-level classes: Novice, Technician, and Technician Plus.
I have omitted the 60-meter band since it is channelized and does not allow arbitrary frequencies.
The program for demonstrating these frequencies is here:

```
From machine import Pin, PWM
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()
```

If you don’t (yet) have an amateur radio license, let me take this opportunity to encourage you to get one, since if you have read this far, you obviously have an interest, and the license is very easy to get (Google for “amateur radio license”). The tests are simple, all the questions and answers are available online, there are many free tutorial guides, and unlimited online practice tests, and you only have to get 70% of the questions right to pass.

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 sma 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:

```
From machine import Pin, PWM
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 timing from
Farnsworth_spacing = (60000 * self.character_speed – 37200 * overall_speed) / (overall_speed *
self.character_speed)
Farnsworth_spacing *= 60000/68500    # A fudge factor to get the ESP8266 timing closer to correct
Self.LETTER_SPACE = int((3 * farnsworth_spacing) / 19) – self.CYPHER_SPACE
Self.WORD_SPACE = int((7 * farnsworth_spacing) / 19) – self.CYPHER_SPACE
Def send(self, str):
From the_code import code
From time import sleep_ms
For c in str:
If c == ‘ ‘:
Self.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:

```
From cwmorse import CWMorse
From machine import Pin, PWM
From time import sleep
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()
```

We import our modified CWMorse module. We will use pin 18 for our PWM output pin, and we will transmit on a frequency of 7,032,967 Hertz. I chose to set the frequency to one Hertz below that, knowing that the chip would do the arithmetic and pick the next available frequency.

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.

The rest of the program should be familiar. We create the cw object, passing in the frequency. We set the Morse code speed to 20 words per minute. Then we loop, sending out the Morse-coded messag and then sleeping for 5 seconds before repeating. What we have built is what amateur radio operators call a beacon.

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,
1000, 2200, 3300, 4700, 6800, 10000, 22000, 33000, 47000, 100000, 1000000]
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,
47, 68, 100, 120, 150, 220, 330, 470, 680, 1000, 10000, 22000, 33000, 47000, 68000, 100000]
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
while newQ * newQ + 1 <= hi / lo:
newQ += 0.1
squared = (hi / lo) / (newQ * newQ + 1 – (hi / lo))
while squared <= 0:
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 our filter, we input 50 ohms for the input and output impedances, 1 for the Q, and 7032967 for the center frequency. We get:

```
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
```

Note that although the frequency came out a little higher than we asked for when we used standard values, and the output impedance is not quite 50 ohms, the results are close enough to make a decent filter. You can use online programs to design filters with more elements if you want even better performance.

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.

High frequencies are blocked by the coil. But we also have the two capacitors. They act like short circuits to ground for high frequencies. High frequencies do not make it out of the antenna.

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 13 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, 1923 1937984, 1953125, 1968504, 1984127, 2000000,
80-meter band: 3521127, 3571429, 3623188, 3676471, 3731343, 3787879, 3846154, 3906250, 39682
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.
Since we can change the system clock frequency, we can hunt for values of the clock frequency that give us legal radio frequencies. A program to do this looks like this:

```
from machine import Pin, PWM, freq
from time import ticks_ms, ticks_diff
def main():
pin = Pin(15, Pin.OUT)
pwm = PWM(pin, freq=1800000)
pwm.duty_u16(32025)
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
( 50000000,  54000000), # 6 meters
(144000000, 148000000)  # 2 meters
)
t = ticks_ms()
results = {}
guess = 0
old_len = 0
for x in bands:
f_lo, f_hi = x
```

```
print(f_lo)
for f in range(f_lo, f_hi):
for clock_mul in range(2, 21):
freq(12000000 * clock_mul)
pwm.freq(f)
actual = pwm.freq()
if actual != guess and actual > f_lo and actual < f_hi:
results[str(actual)] = clock_mul
if len(results) != old_len:
print(str(round(ticks_diff(ticks_ms(), t) / 1000.0)) + ":", results)
old_len = len(results)
guess = actual
print(results)
main()
```

After running for over five minutes, it gives us this line:

325: {'1815126': 18, '1806452': 14, '1821429': 17, '1824000': 19, '1833333': 11, '1837838': 17, '1838710': 19, '1809524': 19, '1811321': 8, '1830508': 18, '1846154': 8, '1808219': 11, '1805310': 17, '1835294': 13, '1804511': 20, '1828571': 16, '1813953': 13, '1822785': 12, '1826087': 14, '1818182': 10, '1832061': 20, '1836735': 15}

This requires a bit of explanation. We can't just set the system clock to any frequency willy-nilly. The board has a 12 MHz crystal as a timebase and the chip multiplies that by some integer factor between 2 and 21 to create the system clock. That output line tells us that if we set the system clock to 18 times 12 MHz, we can set the PWM to 1815126 and it will stay there. The same goes for the other 21 values.

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:

```
class RP_CW:
def __init__(self, pin):
from machine import Pin
from rp2 import PIO, StateMachine, asm_pio
@asm_pio(set_init=PIO.OUT_LOW)
def square():
wrap_target()
set(pins, 1)
set(pins, 0)
wrap()
self.pin = Pin(pin, Pin.OUT)
self.f = 7030000
self.sm = rp2.StateMachine(0, square, freq=2*self.f, set_base=self.pin)
def on(self):
self.sm.active(1)
def off(self):
self.sm.active(0)
def frequency(self, frq):
self.f = frq
def main():
from time import sleep_ms
cw = RP_CW(15)
while True:
```

```
cw.on()
sleep_ms(100)
cw.off()
sleep_ms(100)
main()
```

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.

There are two main parts to explain. The @asm_pio and the square method that follows it define two instructions for the state machine. The first sets the pin to high. The second sets it too low. That is all that happens there.

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:

```
from machine import Pin, PWM
from rp2 import PIO, StateMachine, asm_pio
class RP_CW:
def __init__(self, pin):
from machine import Pin
from rp2 import PIO, StateMachine, asm_pio
@asm_pio(set_init=PIO.OUT_LOW)
def square():
wrap_target()
set(pins, 1)
set(pins, 0)
wrap()
self.pin = Pin(pin, Pin.OUT)
self.f = 7030000
self.sm = rp2.StateMachine(0, square, freq=2*self.f, set_base=self.pin)
def on(self):
self.sm.active(1)
def off(self):
self.sm.active(0)
def frequency(self, frq):
self.f = frq
class CWMorse:
character_speed = 18
def __init__(self, pin, freq):
self.cw = RP_CW(pin)
self.cw.frequency(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 timing from “https://www.arrl.org/files/file/Technology/x9004008.pdf”
farnsworth_spacing = (60000 * self.character_speed - 37200 * overall_speed) / (overall_speed *
self.character_speed)
farnsworth_spacing *= 60000/68500    # A fudge factor to get the ESP8266 timing closer to correct
self.LETTER_SPACE = int((3 * farnsworth_spacing) / 19) - self.CYPHER_SPACE
self.WORD_SPACE = int((7 * farnsworth_spacing) / 19) - self.CYPHER_SPACE
def send(self, str):
from the_code import code
from time import sleep_ms
for c in str:
if c == ‘ ‘:
self.cw.off()
sleep_ms(self.WORD_SPACE)
else:
cyphers = code[c.upper()]
for x in cyphers:
if x == ‘.’:
self.cw.on()
sleep_ms(self.DOT)
else:
self.cw.on()
sleep_ms(self.DASH)
self.cw.off()
sleep_ms(self.CYPHER_SPACE)
self.cw.off()
sleep_ms(self.LETTER_SPACE)
from time import sleep
frequency = 7030000
def main():
cw = CWMorse(15, frequency)
cw.speed(5)
print(”CW transmitter”)
msg = “AB6NY testing RP2040 as a 40 meter transmitter sending on “ + str(frequency) + “ Hertz.”
while True:
print(msg)
cw.send(msg)
sleep(5)
main()
```

My RP2040 without any amplification was putting out 3.8 milliwatts (5.8 dBm) after the low pass filter. That tiny signal could be heard several miles away, even though the antenna was just a 6-foot whip tuned for 40 meters.

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.
There is a written test of the rules you have to follow when transmitting. There are free simple study guides for the test. All the questions and answers are available online. You only need to get 74% right to pass (26 questions right out of 35). You can practice the test online as many times as you like (also here, and here, and here, and many more).
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)
The code for this project is a slight modification of code we used in previous projects. The

cwmorse.py module looks like this:

```
From machine import Pin
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 timing from
Farnsworth_spacing = (60000 * self.character_speed – 37200 * overall_speed) / (overall_speed *
self.character_speed)
Farnsworth_spacing *= 60000/68500    # A fudge factor to get the ESP8266 timing closer to correct
Self.LETTER_SPACE = int((3 * farnsworth_spacing) / 19) – self.CYPHER_SPACE
Self.WORD_SPACE = int((7 * farnsworth_spacing) / 19) – self.CYPHER_SPACE
Def send(self, str):
From the_code import code
From time import sleep_ms
For c in str:
If c == ‘ ‘:
Self.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.

Big 1,500-watt transmitters are expensive and heat the room. Let’s start with something affordable

that can still reach over a hundred miles using a cheap wire antenna (in my case, an end-fed half-wave antenna for 40 meters for $7.05 at AliExpress.com).
The Pixie 2 transceiver is easy to find with a Google search. You can get it in kit form for $3.55 on AliExpress.com, or fully built and ready to use in a transparent acrylic case for $12.88 on eBay.
Press enter or click to view image in full size
The RP2040 connected to a Pixie 2 transceiver.
Photo by author
That's it. You are on the air for about $30.
Our main.py module is simple:

```
From cwmorse import CWMorse
From time import sleep
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()
```

This sets up a beacon for testing the range of your transmitter. Now you can use a good receiver and another EFHW (end-fed half-wave) antenna as you drive around (tossing one end of the antenna wire u into trees).
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’: ‘--..’,
‘0’: ‘-----‘,
‘1’: ‘.----‘,
‘2’: ‘..---‘,
‘3’: ‘...—‘,
‘4’: ‘....-‘,
‘5’: ‘.....’,
‘6’: ‘-....’,
‘7’: ‘--...’,
‘8’: ‘---..’,
‘9’: ‘----.’,
‘.’: ‘.-.-.-‘,
‘,’: ‘--..—‘,
‘?’: ‘..--..’,
‘\’’: ‘.----.’,
‘!’: ‘-.-.—‘,
‘/’: ‘-..-.’,
‘(‘: ‘-.--.’,
‘)’: ‘-.--.-‘,
‘&’: ‘.-...’,
‘:’: ‘---...’,
‘;’: ‘-.-.-.’,
‘=’: ‘-...-‘,
‘+’: ‘.-.-.’,
‘-‘: ‘-....-‘,
‘_’: ‘..--.-‘,
‘”’: ‘.-..-.’,
‘$’: ‘...-..-‘,
‘@’: ‘.--.-.’,
}

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

If the 1.2 watts of the Pixie 2 is not enough, you can go for 5 watts:

Press enter or click to view image in full size

The NS-40+ QRP transmitter.

Image by author

For $30 you can get the NS-40+ transmitter kit. The NS stands for “None Simpler” because all of the coils in the circuit are printed right on the circuit board. There are only 16 parts to solder. The kit goes together in minutes.

Connect to the same EFHW antenna and as much as 12 volts (shown above using a 9-volt battery, whi 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?

None of the links in this article are affiliate links. I only make money when you clap for this article.

Radio

Decoding POCSAG using Gqrx & RTL-SDR

Noë Flatreaud

Noë Flatreaud

Follow
4 min read
.
Oct 6, 2024
Listen
Share
More
Recent events, involving Hezbollah, Mossad and the explosion of thousands of trapped pagers, sparke
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
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 fre
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.
In principle, any software defined radio (SDR) covering a frequency range up to 800 MHz should be
suitable to monitor POCSAG communication. This also includes cheap RTL-SDR USB sticks. The RTL
was initially produced as DVB-T tuner and is available for around 25€.
Gqrx: An open-source software-defined radio receiver.
Sox: A command-line audio processing tool.
Multimon-ng: A tool for decoding various digital radio protocols, including POCSAG.
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 Sock
on port 7355.
Multimon-ng helps us identify and decode the POCSAG signals in various speeds (512,1200 and 2400
bps), sox resamples our audio signal from 44100 to 48000 bauds for signal processing.
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 --timest
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.
When using two processors at the same time, there are occasions when they need to communicate wit one another. They can do this by sharing a bit of memory. To make the sharing happen properly, they must agree to take turns, so that one of them isn't reading the memory while the other is not finished writing to it, or (worse) they aren't both writing at the same time.
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.
To illustrate how to use both processors and a lock to protect their communication, we will start with a simple example. We will have one processor blink the onboard LED while the second processor reads the count of how many times the LED blinked. Here is the code:
From _thread import start_new_thread, allocate_lock
From time import sleep
From machine import Pin
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()
```

This code is for the ESP32. If you are using the RP2040, the LED is on pin 25, not pin 2.

We import two methods from the _thread module: start_new_thread(), and allocate_lock().

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).

To make handling the radios easy, we build a module and a class:

```
From machine import UART, Pin
From time import sleep_ms
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 bytes(b for b in data if b < 128)
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)
Self.uart.flush()  # Wait for all writing to finish
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 constructor for the HC12 class creates the UART object using arguments passed in for the pins

and baud rate. It also sets the radio into long-distance mode (1200 baud).
The command() method handles setting and clearing the SET bit to issue the AT commands to the radi
The radio has a habit of sending characters like 0x80 and 0xFF which can't be decoded into utf-8 for printing, so the cleanup() method is used to remove anything with the high bit set.
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:

```
From _thread import start_new_thread, allocate_lock
From machine import Pin
From hc12 import HC12
LED = Pin(25, Pin.OUT)
From whoami import WhoAmI
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”:
From machine import Timer
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”)
# Give the board 5 seconds after a reset to allow us to replace main.py during debugging
From time import sleep
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 second bit of housekeeping is only used for the debugging phase. If two processors are both running when we try to send new files to the RP2040, the upload hangs. So we delay by five seconds after a reset, so we can quickly start loading files.

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.

The transmit() thread loops, getting the lock, sending the data, and flashing the LED while it transmits. On the RP2040 the LED is on when you send it a one, but on the ESP32 it is on when you send a zero. So on my RP2040 repeater, the LED is off when it is transmitting. You can (of course) change this if you like.

The Ping class handles the timer. The timeout() method sends the text.

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.

Instead of the Ping class sending automatically, you can have the radio read from the USB port and

send the text it reads. This lets you use your laptop keyboard to send and receive.
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.
The radios have a connector for an external antenna if you want even more range. The repeater will want an omnidirectional antenna (the ground plane antenna we built earlier is a good choice).
The rovers can use directional antennas, such as a dish or a Yagi-Uda. You can build these yourself or purchase them (commercial antennas for the 70-centimeter band are easy to find). Of course, they can also use the ground plane antenna or just the tiny spring antenna.
As promised, the whoami.cfg file for rover1 looks like this:
{“name”:”rover1”,”tx_channel”:”C001”,”rx_channel”:”C004”}
And for the repeater it looks like this:
{“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):
Self.me = {}
Try:
With open(“whoami.cfg”,”rb”) as f:
Line = f.read(1024)
From json import loads
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.
Python is not known for its speed. However, when we need speed, we often find that there are built-in functions that run as fast as the machine can go. These functions are written in C, the language that Python is written in.
In this chapter, we will build an SDR AM radio receiver (and transmitter) in MicroPython for the Raspberry Pi Pico 2 (the RP2350).
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 G 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 MicroPytho firmware. That means we need to write a Python module in C and link it in.
Adding simple C modules to MicroPython is easy. The developers have given us a nice mechanism for doing this. Unfortunately, ours is not a simple module. It needs to access subroutines from the Pico SDK and from the Arm CMSIS system. The easy mechanism can't do that.
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.
MicroPython is built under Linux. I have many Linux machines, but my big, beefy, fast server is running Windows. So I use Windows System for Linux to do the build. It runs Linux under Windows. Yo type this into a CMD window:
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):

```
#!/bin/bash
Set -e # Exit immediately if any command fails
# --- Configuration ---
MPY_VERSION="v1.25.0"
BOARD="RPI_PICO2"
PROJECT_ROOT=~/AM_sdr_pico2_final
# This is the directory where you have staged all your working, vendored files.
USER_SOURCE_DIR=~/sdr_radio
# --- Sanity Check ---
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) ---”
# --- STEPS 1-2: SETUP & VENDORING ---
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
# Add pico-extras, which is separate
Git submodule add  lib/pico-extras
git submodule update --init lib/pico-extras
# --- BRUTE-FORCE VENDORING of CMSIS ---
# The git submodule process is unreliable. We will download and place the files manually.
Echo “Manually downloading and vendoring CMSIS-DSP library...”
# Create the target directory structure
Mkdir -p ./lib/vendor/CMSIS_5
# Download a known-good version of the library as a ZIP file
Wget -O cmsis.zip
# Unzip it into a temporary directory
Unzip -q cmsis.zip -d ./lib/vendor/
# Move the contents into our final location
Mv ./lib/vendor/CMSIS_5-5.9.0/* ./lib/vendor/CMSIS_5/
# Clean up
Rm cmsis.zip
Rm -rf ./lib/vendor/CMSIS_5-5.9.0/
# --- VERIFICATION STEP ---
# Check the path where we downloaded the files.
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
# --- STEP 3: CREATE THE SELF-CONTAINED SDR MODULE ---
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}/
# --- STEP 4: MODIFY BUILD FILES (THE DEFINITIVE FIX) ---
Echo “--- [4/5] Configuring the MicroPython build... ---”
Cd ./ports/rp2
# 1. Reset and activate the module in the C preprocessor.
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
```

```
# 2. Reset and inject the complete module configuration into CMake.
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# --- Customization for sdr_radio module ---\n\
# We will now build the CMSIS-DSP sources directly into the firmware.\n\
\n\
# Part 1: Add all necessary include paths.\n\
Include_directories(\n\
\${MICROPY_DIR}/extmod/sdr_radio \n\
\${MICROPY_DIR}/py \n\
\${MICROPY_DIR}/ports/rp2 \n\
# Include paths for CMSIS-DSP Public API, Private Helpers, and Core types.\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\
# Your other existing include paths\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\
# \n\
# Part 2: Create a list containing ONLY the main 'roll-up' source files.\n\
Set(CMSIS_DSP_SOURCES\n\
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/BasicMathFunctions/BasicMathFunction
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/CommonTables/CommonTables.c\"\n\
\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/ComplexMathFunctions/ComplexMathFun
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/ControllerFunctions/ControllerFunctions
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/FastMathFunctions/FastMathFunctions.c
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/FilteringFunctions/FilteringFunctions.c\"\
\"\${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
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/SupportFunctions/SupportFunctions.c\"\
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/TransformFunctions/TransformFunctions
ROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/QuaternionMathFunctions/QuaternionMathFunct
)\n\
\n\
# Part 3: Add our module's C file AND the curated CMSIS-DSP SOURCE FILES to MicroPython's build
list.\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\
# --- End of customizations ---\n"
Awk -v block="$CUSTOM_BLOCK_SOURCES" -v target="$TARGET_LINE_SOURCES" 'index($0, tar
block} 1' CMakeLists.txt > CMakeLists.txt.new && mv CMakeLists.txt.new CMakeLists.txt
### sed -i '/Execute _boot.py to set up the filesystem/a \        mp_printf(MP_PYTHON_PRINTER,
```

“Kilroy with Micropython threads and ADC fix\\n”);’ main.c
--- STEP 5: BUILD THE FIRMWARE ---
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 ---”
# Check the final ELF for the module symbol. This will now pass.
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:
#include “py/runtime.h”
#include “py/mphal.h”
#include <math.h>
#include <string.h>
#include “hardware/dma.h”
#include “hardware/adc.h”
#include “hardware/irq.h”
#include “hardware/sync.h”
#include “hardware/resets.h”
#include <float.h>
#include “hardware/clocks.h”
#include “hardware/pwm.h”
#include “arm_math.h”
#include “pico/multicore.h”
#define ADC_SAMPLE_RATE 500000
#define AUDIO_SAMPLE_RATE 22050
#define mult_q31(a, b) ((q31_t)(((int64_t)(a) * (b)) >> 31))
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
// --- State for the Iterative NCO (Mixer) ---
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)
// --- State for the fixed-point RF DC Blocker ---
Q31_t dc_block_i_x1;
Q31_t dc_block_i_y1;
Q31_t dc_block_q_x1;
Q31_t dc_block_q_y1;
// --- State for the LPF (Cascaded EMA) ---
Q31_t ema_i_s1, ema_i_s2, ema_i_s3;
Q31_t ema_q_s1, ema_q_s2, ema_q_s3;;
Q31_t demod_mag_x1;
// --- State for the Audio HPF (DC Blocker) ---
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.
#define MAX_CAPTURE_BUFFER_SIZE 8192
Static int adc_dma_chan_A = -1;
Static int adc_dma_chan_B = -1;
// Internal ping-pong buffers for the DMA
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.
// e.g., for a 190kHz station, remainder is 190k. We tune to 190k.
Nco_tune_freq_hz = remainder;
} else {
// If it’s in the upper half, the alias is mirrored from the top.
// e.g., for an 810kHz station, remainder is 310k. We tune to 500k-310k = 190k.
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) {
// ============================================================
//  FAST PATH for AM MODE (No RF DC Blocker)
// ============================================================
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);
Q31_t q_raw = mult_q31(sample, nco_s); // Use positive sine for Q
// 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_LP
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_
```

```
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_
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_LP
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 {
// ================================================================
//  FAST PATH for CW/SSB MODE (with BFO)
// ================================================================
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:

```
from machine import PWM, ADC, Pin
import array
from sdr_radio import SDR_Radio, configure_transmitter_pwm, audio_play_chunk
from sdr_radio import audio_wait_done, audio_deinit, deinit_capture, audio_configure
machine.freq(250_000_000)
# --- Configuration ---
TRANSMIT_FREQ = 600_000
AUDIO_SAMPLE_RATE = 22050
PWM_UPDATE_RATE = 5_000_000
PWM_TOP = 49 # 250MHz / (5MHz * (49+1)) = 1.0 divider
AUDIO_BUFFER_SAMPLES = 128
# Calculate how many 32-bit words we need for the PWM buffer
PWM_BUFFER_WORDS = int(AUDIO_BUFFER_SAMPLES * (PWM_UPDATE_RATE / AUDIO_SAMP
PWM_PIN = 20
ADC_PIN = 26
sdr = SDR_Radio()
sdr.set_tx_carrier(TRANSMIT_FREQ, PWM_UPDATE_RATE)
# Audio buffer is uint16_t ('H')
# This buffer will be filled by the DMA ADC
audio_in_buf = array.array('H', (0 for _ in range(AUDIO_BUFFER_SAMPLES)))
# PWM buffers are uint32_t ('L') for DMA compatibility
pwm_out_bufs = [
array.array('L', (0 for _ in range(PWM_BUFFER_WORDS))),
array.array('L', (0 for _ in range(PWM_BUFFER_WORDS)))
]
pwm = PWM(Pin(PWM_PIN))
adc = ADC(Pin(ADC_PIN))
configure_transmitter_pwm(pwm, PWM_UPDATE_RATE, PWM_TOP)
audio_configure(pwm, AUDIO_SAMPLE_RATE)
sdr.configure_and_init_capture(AUDIO_SAMPLE_RATE, AUDIO_BUFFER_SAMPLES)
sdr.start_capture()
print("Starting transmitter... Speak into the microphone.")
```

```
buf_idx = 0
try:
# --- Prime the Pump ---
# 1. Capture the first block of audio
sdr.capture_chunk(audio_in_buf)
# 2. Process it into the first PWM buffer
sdr.am_transmit_pipeline(audio_in_buf, pwm_out_bufs[0])
# 3. Start playing the first PWM buffer
audio_play_chunk(pwm, pwm_out_bufs[0])
while True:
next_buf_idx = 1 - buf_idx
# 1. Capture the next block of audio. This happens while the
#    previous PWM buffer is still being transmitted by DMA.
sdr.capture_chunk(audio_in_buf)
# 2. Process the newly captured audio into the *other* PWM buffer.
sdr.am_transmit_pipeline(audio_in_buf, pwm_out_bufs[next_buf_idx])
# 3. Wait for the PWM transmission (started in the previous loop) to complete.
audio_wait_done(pwm)
# 4. Immediately start transmitting the PWM buffer we just prepared.
audio_play_chunk(pwm, pwm_out_bufs[next_buf_idx])
buf_idx = next_buf_idx
except KeyboardInterrupt:
print("\nStopping transmitter.")
finally:
audio_deinit(pwm)
deinit_capture()
```

Like the receiver, we use ping-pong buffers to allow the DMA to happen in the background while we process the inputs on the CPU.

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).

We will run our PWM at 5 MHz. This will give us 8 and a third cycles of PWM for every cycle of our target frequency (600 kHz), allowing us to set our PWM duty cycle to any number between 0 and 49. With 50 levels between 0 volts and 3.3 volts, we can generate a sine wave that has only a small bit of high harmonics that are easy to filter out.

As with the receiver, the parts that need to be very fast are done in C, and linked into the Micropython runtime. Besides the DMA for the input and output, those parts are configure_transmitter_pwm(), set_tx_carrier(), and am_transmit_pipeline():

```
static mp_obj_t configure_transmitter_pwm(mp_obj_t pwm_obj, mp_obj_t update_rate_obj, mp_obj_t top_obj) {
machine_pwm_obj_t *pwm = MP_OBJ_TO_PTR(pwm_obj);
uint32_t update_rate = mp_obj_get_int(update_rate_obj);
uint32_t top = mp_obj_get_int(top_obj);
pwm_set_enabled(pwm->slice, false);
uint32_t source_hz = clock_get_hz(clk_sys);
float div = (float)source_hz / ((float)(top + 1) * (float)update_rate);
if (div < 1.0f) div = 1.0f;
pwm_set_clkdiv(pwm->slice, div);
pwm_set_wrap(pwm->slice, top);
```

```
pwm_set_chan_level(pwm->slice, pwm->channel, 0);
pwm_set_enabled(pwm->slice, true);
audio_is_configured = true;
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_3(configure_transmitter_pwm_obj, configure_transmitter_pwm
static mp_obj_t sdr_radio_set_tx_carrier(mp_obj_t self_in, mp_obj_t freq_obj, mp_obj_t pwm_rate_obj)
{
sdr_radio_obj_t *self = MP_OBJ_TO_PTR(self_in);
self->tx_carrier_freq_hz = mp_obj_get_int(freq_obj);
uint32_t pwm_update_rate = mp_obj_get_int(pwm_rate_obj);
self->tx_nco_phase_increment = (q31_t)(((uint64_t)self->tx_carrier_freq_hz << 31) /
pwm_update_rate);
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_3(sdr_radio_set_tx_carrier_obj, sdr_radio_set_tx_carrier);
# define AUDIO_GAIN 3.0f;
static mp_obj_t sdr_radio_am_transmit_pipeline(mp_obj_t self_in, mp_obj_t audio_buf_obj, mp_obj_t
pwm_buf_obj) {
sdr_radio_obj_t *self = MP_OBJ_TO_PTR(self_in);
mp_buffer_info_t audio_info; mp_get_buffer_raise(audio_buf_obj, &audio_info, MP_BUFFER_READ);
mp_buffer_info_t pwm_info;   mp_get_buffer_raise(pwm_buf_obj,   &pwm_info,   MP_BUFFER_WRITE
uint16_t *audio_in_ptr = (uint16_t *)audio_info.buf;
uint32_t *pwm_out_ptr = (uint32_t *)pwm_info.buf;
const int num_audio_samples = audio_info.len / sizeof(uint16_t);
const int num_pwm_words = pwm_info.len / sizeof(uint32_t);
// --- Constants ---
const int PWM_TOP = 49;    // Must match Python
const int PWM_CENTER = 25; // (PWM_TOP + 1) / 2
const float MODULATION_DEPTH = 0.99f;
// These must match your Python script's constants
const uint32_t PWM_UPDATE_RATE = 5000000;
const float RATIO = (float)PWM_UPDATE_RATE / (float)AUDIO_SAMPLE_RATE;
int pwm_idx = 0;
// --- Main Processing Loop ---
for (int i = 0; i < num_audio_samples; i++) {
// 1. Get current and next audio sample for linear interpolation
float audio_start = ((float)audio_in_ptr[i] - 2048.0f) / 2048.0f;
float audio_end = (i + 1 < num_audio_samples) ?
((float)audio_in_ptr[i + 1] - 2048.0f) / 2048.0f :
audio_start;
audio_start *= AUDIO_GAIN;
audio_end *= AUDIO_GAIN;
// 2. Linear Interpolation loop
// Calculate how many PWM samples this one audio sample covers
int start_j_idx = (int)(i * RATIO);
int end_j_idx = (int)((i + 1) * RATIO);
for (int pwm_sample_idx = start_j_idx; pwm_sample_idx < end_j_idx; pwm_sample_idx++) {
if (pwm_idx >= num_pwm_words) break; // Safety break
float interp_point = (float)(pwm_sample_idx - start_j_idx) / (float)(end_j_idx - start_j_idx);
```

```
float audio_interp = audio_start * (1.0f - interp_point) + audio_end * interp_point;
// 3. Generate carrier sample
q31_t carrier_q31 = arm_cos_q31(self->tx_nco_phase);
self->tx_nco_phase += self->tx_nco_phase_increment;
float carrier_float = (float)carrier_q31 / 2147483648.0f;
// 4. Modulate
float modulator = 1.0f + (audio_interp * MODULATION_DEPTH);
float am_signal = carrier_float * modulator;
// 5. Scale to PWM duty cycle
int32_t duty_cycle = (int32_t)(PWM_CENTER * (1.0f + am_signal));
// 6. Clamp
if (duty_cycle > PWM_TOP) duty_cycle = PWM_TOP;
if (duty_cycle < 0) duty_cycle = 0;
// 7. Pack two 16-bit duty cycles into one 32-bit word for the DMA
// PWM B gets silence (25).
// PWM A gets the duty_cycle
pwm_out_ptr[pwm_idx] = ((uint32_t)25 << 16) | (uint32_t)duty_cycle;
pwm_idx++;
}
}
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_3(sdr_radio_am_transmit_pipeline_obj,
sdr_radio_am_transmit_pipeline);
```

We configure the PWM for 5 MHz and 0–49 levels using configure_transmitter_pwm().

We set the target frequency for the carrier wave to 600 kHz using set_tx_carrier(). This function calculates the phase increment we will use to generate a 600 kHz sine wave.

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 center between the two resistors is where we find half the 3.3 volts (1.65 volts). We connect this to the ADC input. Now, when the sound card voltages come in, they get added and subtracted from 1.65 volts to give us a range of 0.65 to 2.65 volts, which is (nearly) perfect for the ADC.

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 wo of the two resistors.

The numbers I got were a little smaller than I liked, so I multiplied them by 3 (AUDIO_GAIN) to get them closer to 0 to 4095. I could instead have amplified the signal before giving it to the RP2350, but I was aiming for a minimal parts count.

The main loop in am_transmit_pipeline() is a linear interpolator, which smooths 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 conne 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:

```
from cwmorse import CWMorse
from time import sleep
frequency = 540000
tone = 14
carrier = 15
def main():
cw = CWMorse(carrier, tone, frequency)
cw.speed(10)
print(”CW transmitter”)
```

```
msg = “AB6NY testing RP2040 as an AM transmitter sending on “ + str(frequency) + “ Hertz.”
cw.tune(True)
sleep(30)
cw.tune(False)
while True:
print(msg)
cw.send(msg)
sleep(5)
main()
```

The cwmorse.py module looks familiar, with just a few changes:

```
from machine import Pin, PWM
from time import sleep
class RP_CW:
def __init__(self, carrier_pin, tone_pin, freq):
from machine import Pin
from rp2 import PIO, StateMachine, asm_pio
self.tone_pin = Pin(tone_pin, Pin.OUT)
self.carrier_pin = Pin(carrier_pin, Pin.OUT)
self.pwm = PWM(tone_pin, freq=500, duty_u16=0)
@asm_pio(set_init=PIO.OUT_LOW)
def square():
wrap_target()
set(pins, 1)
set(pins, 0)
wrap()
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.pwm.duty_u16(32767)
def off(self):
self.pwm.duty_u16(0)
def frequency(self, frq):
self.f = frq
class CWMorse:
character_speed = 20
def __init__(self, carrier_pin, tone_pin, freq):
self.cw = RP_CW(carrier_pin, tone_pin, freq)
self.cw.frequency(freq)
def tune( self, onoff):
if onoff:
self.cw.on()
else:
self.cw.off()
def speed(self, overall_speed):
if overall_speed >= 20:
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:
self.LETTER_SPACE = int(3 * self.DOT) - self.CYPHER_SPACE
self.WORD_SPACE = int(7 * self.DOT) - self.CYPHER_SPACE
else:
# Farnsworth timing from “https://www.arrl.org/files/file/Technology/x9004008.pdf”
farnsworth_spacing = (60000 * self.character_speed - 37200 * overall_speed) / (overall_speed *
self.character_speed)
farnsworth_spacing *= 60000/68500    # A fudge factor to get the ESP8266 timing closer to correct
self.LETTER_SPACE = int((3 * farnsworth_spacing) / 19) - self.CYPHER_SPACE
self.WORD_SPACE = int((7 * farnsworth_spacing) / 19) - self.CYPHER_SPACE
def send(self, str):
from the_code import code
from time import sleep_ms
for c in str:
if c == ‘ ‘:
self.cw.off()
sleep_ms(self.WORD_SPACE)
else:
cyphers = code[c.upper()]
for x in cyphers:
if x == ‘.’:
self.cw.on()
sleep_ms(self.DOT)
else:
self.cw.on()
sleep_ms(self.DASH)
self.cw.off()
sleep_ms(self.CYPHER_SPACE)
self.cw.off()
sleep_ms(self.LETTER_SPACE)
```

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 Hindenl 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
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.
Python is not known for its speed. However, when we need speed, we often find that there are built-in functions that run as fast as the machine can go. These functions are written in C, the language that Python is written in.
In this chapter, we will build an SDR AM radio receiver (and transmitter) in MicroPython for the Raspberry Pi Pico 2 (the RP2350).
This $5 processor will replace both the laptop computer and the RTL-SDR USB dongle we have used i 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 G 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 MicroPytho firmware. That means we need to write a Python module in C and link it in.
Adding simple C modules to MicroPython is easy. The developers have given us a nice mechanism for doing this. Unfortunately, ours is not a simple module. It needs to access subroutines from the Pico SDK and from the Arm CMSIS system. The easy mechanism can't do that.
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.
MicroPython is built under Linux. I have many Linux machines, but my big, beefy, fast server is running Windows. So I use Windows System for Linux to do the build. It runs Linux under Windows. Yo type this into a CMD window:

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):

```
#!/bin/bash
set -e # Exit immediately if any command fails
# --- Configuration ---
MPY_VERSION="v1.25.0"
BOARD="RPI_PICO2"
PROJECT_ROOT=~/AM_sdr_pico2_final
# This is the directory where you have staged all your working, vendored files.
USER_SOURCE_DIR=~/sdr_radio
# --- Sanity Check ---
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) ---"
# --- STEPS 1-2: SETUP & VENDORING ---
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} https://github.com/micropython/micropython.git
${PROJECT_ROOT}/micropython
cd ${PROJECT_ROOT}/micropython
git submodule update --init --recursive
# Add pico-extras, which is separate
git submodule add https://github.com/raspberrypi/pico-extras.git lib/pico-extras
git submodule update --init lib/pico-extras
# --- BRUTE-FORCE VENDORING of CMSIS ---
# The git submodule process is unreliable. We will download and place the files manually.
echo "Manually downloading and vendoring CMSIS-DSP library..."
# Create the target directory structure
mkdir -p ./lib/vendor/CMSIS_5
# Download a known-good version of the library as a ZIP file
wget -O cmsis.zip https://github.com/ARM-software/CMSIS_5/archive/refs/tags/5.9.0.zip
# Unzip it into a temporary directory
unzip -q cmsis.zip -d ./lib/vendor/
# Move the contents into our final location
mv ./lib/vendor/CMSIS_5-5.9.0/* ./lib/vendor/CMSIS_5/
# Clean up
rm cmsis.zip
rm -rf ./lib/vendor/CMSIS_5-5.9.0/
# --- VERIFICATION STEP ---
# Check the path where we downloaded the files.
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
# --- STEP 3: CREATE THE SELF-CONTAINED SDR MODULE ---
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}/
# --- STEP 4: MODIFY BUILD FILES (THE DEFINITIVE FIX) ---
echo "--- [4/5] Configuring the MicroPython build... ---"
cd ./ports/rp2
# 1. Reset and activate the module in the C preprocessor.
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
# 2. Reset and inject the complete module configuration into CMake.
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# --- Customization for sdr_radio module ---\n\
# We will now build the CMSIS-DSP sources directly into the firmware.\n\
\n\
# Part 1: Add all necessary include paths.\n\
include_directories(\n\
\${MICROPY_DIR}/extmod/sdr_radio \n\
\${MICROPY_DIR}/py \n\
\${MICROPY_DIR}/ports/rp2 \n\
# Include paths for CMSIS-DSP Public API, Private Helpers, and Core types.\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\
# Your other existing include paths\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\
# \n\
# Part 2: Create a list containing ONLY the main 'roll-up' source files.\n\
set(CMSIS_DSP_SOURCES\n\
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/BasicMathFunctions/BasicMathFunction
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/CommonTables/CommonTables.c\"\n\
\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/ComplexMathFunctions/ComplexMathFu
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/ControllerFunctions/ControllerFunctions
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/FastMathFunctions/FastMathFunctions.
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/FilteringFunctions/FilteringFunctions.c\"\
```

```
\"\${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
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/SupportFunctions/SupportFunctions.c\"\
\"\${MICROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/TransformFunctions/TransformFunction
ROPY_DIR}/lib/vendor/CMSIS_5/CMSIS/DSP/Source/QuaternionMathFunctions/QuaternionMathFunct
)\n\
\n\
# Part 3: Add our module's C file AND the curated CMSIS-DSP SOURCE FILES to MicroPython's build
list.\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\
# --- End of customizations ---\n"
awk -v block="$CUSTOM_BLOCK_SOURCES" -v target="$TARGET_LINE_SOURCES" 'index($0, tar
block} 1' CMakeLists.txt > CMakeLists.txt.new && mv CMakeLists.txt.new CMakeLists.txt
### sed -i '/Execute _boot.py to set up the filesystem/a \        mp_printf(MP_PYTHON_PRINTER,
"Kilroy with Micropython threads and ADC fix\\n");' main.c
# --- STEP 5: BUILD THE FIRMWARE ---
echo "--- [5/5] Starting the final MicroPython build ---"
make -j4 BOARD=${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 ---"
# Check the final ELF for the module symbol. This will now pass.
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:

```
#include "py/runtime.h"
#include "py/mphal.h"
#include <math.h>
#include <string.h>
#include "hardware/dma.h"
#include "hardware/adc.h"
#include "hardware/irq.h"
```

```
#include "hardware/sync.h"
#include "hardware/resets.h"
#include <float.h>
#include "hardware/clocks.h"
#include "hardware/pwm.h"
#include "arm_math.h"
#include "pico/multicore.h"
#define ADC_SAMPLE_RATE 500000
#define AUDIO_SAMPLE_RATE 22050
#define mult_q31(a, b) ((q31_t)(((int64_t)(a) * (b)) >> 31))
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
// --- State for the Iterative NCO (Mixer) ---
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)
// --- State for the fixed-point RF DC Blocker ---
q31_t dc_block_i_x1;
q31_t dc_block_i_y1;
q31_t dc_block_q_x1;
q31_t dc_block_q_y1;
// --- State for the LPF (Cascaded EMA) ---
q31_t ema_i_s1, ema_i_s2, ema_i_s3;
q31_t ema_q_s1, ema_q_s2, ema_q_s3;;
q31_t demod_mag_x1;
// --- State for the Audio HPF (DC Blocker) ---
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.
```

```
#define MAX_CAPTURE_BUFFER_SIZE 8192
static int adc_dma_chan_A = -1;
static int adc_dma_chan_B = -1;
// Internal ping-pong buffers for the DMA
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.
// e.g., for a 190kHz station, remainder is 190k. We tune to 190k.
nco_tune_freq_hz = remainder;
} else {
// If it's in the upper half, the alias is mirrored from the top.
// e.g., for an 810kHz station, remainder is 310k. We tune to 500k-310k = 190k.
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) {
```

```
// ================================================================
//  FAST PATH for AM MODE (No RF DC Blocker)
// ================================================================
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);
q31_t q_raw = mult_q31(sample, nco_s); // Use positive sine for Q
// 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_LP
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_A
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_L
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
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 {
// ================================================================
//  FAST PATH for CW/SSB MODE (with BFO)
// ================================================================
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;
q31_t i_raw = mult_q31(sample, nco_c);
q31_t q_raw = mult_q31(sample, nco_s);
// Step 3: RF DC Blocker
q31_t i_blocked = i_raw - self->dc_block_i_x1 + mult_q31(DC_BLOCK_R, self->dc_block_i_y1);
self->dc_block_i_x1 = i_raw; self->dc_block_i_y1 = i_blocked;
q31_t q_blocked = q_raw - self->dc_block_q_x1 + mult_q31(DC_BLOCK_R, self->dc_block_q_y1);
self->dc_block_q_x1 = q_raw; self->dc_block_q_y1 = q_blocked;
// 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_blocked,
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_A
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_blocked,
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
self->ema_q_s3 = q_filtered;
// Step 5: Decimation and Audio Path
if (++decimation_counter >= DECIMATION_FACTOR) {
decimation_counter = 0;
if (audio_idx < num_audio_samples) {
```

```
// Step 6: BFO Mixing for CW/SSB Demodulation
q31_t bfo_c = arm_cos_q31(self->bfo_phase);
q31_t bfo_s = arm_sin_q31(self->bfo_phase);
self->bfo_phase += self->bfo_phase_increment;
// This is a complex multiplication that shifts the signal by the BFO frequency.
// For SSB, this is single-sideband demodulation.
// For CW, this shifts the 0 Hz DC signal up to the audible BFO frequency.
q31_t demodulated_signal = mult_q31(i_filtered, bfo_c) - mult_q31(q_filtered, bfo_s);
// Step 7: Audio HPF to remove any remaining DC
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 = demodulated_signal;
self->audio_hpf_y1 = audio_sample;
temp_audio_buf[audio_idx++] = audio_sample;
}
}
}
}
// ============================================================
//  Sample-by-Sample AGC (Common to both paths)
// ============================================================
const q31_t AGC_ATTACK_ALPHA = 0x01000000;
const q31_t AGC_DECAY_ALPHA  = 0x00100000;
for (int i = 0; i < audio_idx; i++) {
q31_t current_sample = temp_audio_buf[i];
q31_t current_abs = (current_sample > 0) ? current_sample : -current_sample;
if (current_abs > self->agc_smoothed_peak) {
self->agc_smoothed_peak = mult_q31(self->agc_smoothed_peak, (0x7FFFFFFF - AGC_ATTACK_ALP
mult_q31(current_abs, AGC_ATTACK_ALPHA);
} else {
self->agc_smoothed_peak = mult_q31(self->agc_smoothed_peak, (0x7FFFFFFF - AGC_DECAY_ALP
mult_q31(current_abs, AGC_DECAY_ALPHA);
}
int32_t gain_shifts = 0;
if (self->agc_smoothed_peak > 1000) {
gain_shifts = __builtin_clz(self->agc_smoothed_peak) - 2;
}
if (gain_shifts < 0) gain_shifts = 0;
q31_t final_audio;
if (gain_shifts > 0) {
final_audio = __SSAT(((int64_t)current_sample << gain_shifts), 32);
} else {
final_audio = current_sample;
}
int32_t scaled_sample = (final_audio >> 23) + 128;
if (scaled_sample > 255) scaled_sample = 255; else if (scaled_sample < 0) scaled_sample = 0;
pwm_out_ptr[i] = (128 << 16) | (uint32_t)scaled_sample;
}
return mp_const_none;
```

```
}
static MP_DEFINE_CONST_FUN_OBJ_2(fast_sdr_pipeline_obj, fast_sdr_pipeline);
static mp_obj_t sdr_radio_set_bfo(mp_obj_t self_in, mp_obj_t freq_obj) {
sdr_radio_obj_t *self = MP_OBJ_TO_PTR(self_in);
// Get the frequency as an integer from the Python object
int bfo_freq_hz = mp_obj_get_int(freq_obj);
// Calculate the phase increment for the BFO.
// NOTE: This calculation uses AUDIO_SAMPLE_RATE because the BFO
// operates on the decimated, audio-rate signal.
// It also uses "<< 31" because arm_cos_q31 expects a signed Q31 input.
self->bfo_phase_increment = (q31_t)(((uint64_t)bfo_freq_hz << 31) / AUDIO_SAMPLE_RATE);
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_2(sdr_radio_set_bfo_obj, sdr_radio_set_bfo);
static bool consumer_wants_buffer_A = true;
uint32_t sum_a, sum_b;
static mp_obj_t sdr_radio_capture_chunk(mp_obj_t self_in, mp_obj_t buf_obj) {
sdr_radio_obj_t *self = MP_OBJ_TO_PTR(self_in);
mp_buffer_info_t bufinfo;
mp_get_buffer_raise(buf_obj, &bufinfo, MP_BUFFER_WRITE);
uint32_t *src_buf_to_copy = NULL;
if (consumer_wants_buffer_A) {
while (dma_channel_hw_addr(adc_dma_chan_A)->transfer_count > 0) {
// Busy-wait
}
dma_channel_acknowledge_irq0(adc_dma_chan_A);
src_buf_to_copy = capture_buf_A;
dma_channel_set_write_addr(adc_dma_chan_B, capture_buf_B, true); // true = trigger now
} else {
while (dma_channel_hw_addr(adc_dma_chan_B)->transfer_count > 0) {
// Busy-wait
}
dma_channel_acknowledge_irq0(adc_dma_chan_B);
src_buf_to_copy = capture_buf_B;
dma_channel_set_write_addr(adc_dma_chan_A, capture_buf_A, true); // true = trigger now
}
consumer_wants_buffer_A = !consumer_wants_buffer_A;
uint32_t *dma_src = (uint32_t *)src_buf_to_copy;
uint16_t *py_dest = (uint16_t *)bufinfo.buf;
for (uint32_t i = 0; i < (self->capture_num_samples); ++i) {
py_dest[i] = dma_src[i] & 0xFFFF;
}
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_2(sdr_radio_capture_chunk_obj, sdr_radio_capture_chunk);
static mp_obj_t sdr_radio_deinit_capture() {
// Check if channels were claimed before trying to use them
if (adc_dma_chan_A != -1) {
dma_channel_abort(adc_dma_chan_A);
dma_channel_unclaim(adc_dma_chan_A);
```

```
}
if (adc_dma_chan_B != -1) {
dma_channel_abort(adc_dma_chan_B);
dma_channel_unclaim(adc_dma_chan_B);
}
adc_run(false);
adc_dma_chan_A = -1;
adc_dma_chan_B = -1;
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_0(sdr_radio_deinit_capture_obj, sdr_radio_deinit_capture);
// ======================================================================
// 1. THE PWM OBJECT DEFINITION
// ======================================================================
typedef struct _machine_pwm_obj_t {
mp_obj_base_t base;
uint8_t slice;
uint8_t channel;
uint8_t invert;
uint8_t duty_type;
mp_int_t duty;
bool is_streaming;
int stream_dma_chan;
} machine_pwm_obj_t;
// Our own state for the DMA channel.
static bool audio_is_configured = false;
static mp_obj_t sdr_radio_configure_and_init_capture(mp_obj_t self_in, mp_obj_t rate_obj, mp_obj_t
size_obj) {
sdr_radio_obj_t *self = MP_OBJ_TO_PTR(self_in);
self->capture_sample_rate = mp_obj_get_int(rate_obj);
self->capture_num_samples = mp_obj_get_int(size_obj);
reset_block(RESETS_RESET_ADC_BITS | RESETS_RESET_DMA_BITS);
unreset_block_wait(RESETS_RESET_ADC_BITS | RESETS_RESET_DMA_BITS);
adc_init();
adc_gpio_init(26);
adc_select_input(0);
adc_fifo_setup(true, true, 1, false, false);
float div = 48000000.0f / (float)self->capture_sample_rate;
adc_set_clkdiv(div);
uint32_t save_em[6];
for (int i = 0; i < 6; ++i) {
save_em[i] = dma_claim_unused_channel(true);
}
adc_dma_chan_A = dma_claim_unused_channel(true);
adc_dma_chan_B = dma_claim_unused_channel(true);
for (int i = 0; i < 6; ++i) {
dma_channel_unclaim(save_em[i]);
}
dma_channel_config cA = dma_channel_get_default_config(adc_dma_chan_A);
channel_config_set_transfer_data_size(&cA, DMA_SIZE_32);
```

```
channel_config_set_read_increment(&cA, false);
channel_config_set_write_increment(&cA, true);
channel_config_set_dreq(&cA, DREQ_ADC);
channel_config_set_irq_quiet(&cA, false);
dma_channel_configure(adc_dma_chan_A, &cA, capture_buf_A, &adc_hw->fifo, self->capture_num_s
false);
mp_hal_delay_ms(1);
dma_channel_config cB = dma_channel_get_default_config(adc_dma_chan_B);
channel_config_set_transfer_data_size(&cB, DMA_SIZE_32);
channel_config_set_read_increment(&cB, false);
channel_config_set_write_increment(&cB, true);
channel_config_set_dreq(&cB, DREQ_ADC);
channel_config_set_irq_quiet(&cB, false);
dma_channel_configure(adc_dma_chan_B, &cB, capture_buf_B, &adc_hw->fifo, self->capture_num_s
false);
mp_hal_delay_ms(1);
mp_hal_delay_ms(1);
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_3(sdr_radio_configure_and_init_capture_obj,
sdr_radio_configure_and_init_capture);
static mp_obj_t audio_configure(mp_obj_t pwm_obj, mp_obj_t sample_rate_obj) {
machine_pwm_obj_t *pwm = MP_OBJ_TO_PTR(pwm_obj);
if (!audio_is_configured) {
pwm->stream_dma_chan = -1;
}
mp_int_t sample_rate = mp_obj_get_int(sample_rate_obj);
// Configure PWM slice basics
pwm_set_enabled(pwm->slice, false);
pwm_set_wrap(pwm->slice, 255);
uint32_t source_hz = clock_get_hz(clk_sys);
float div = (float)source_hz / (256.0f * (float)sample_rate);
if (div < 1.0f) div = 1.0f;
pwm_set_clkdiv(pwm->slice, div);
// Enable the PWM to send DREQ signals to the DMA
hw_set_bits(&pwm_hw->slice[pwm->slice].csr, 1 << 3); // Set DMAEN bit
// Set initial level and enable the PWM
pwm_set_both_levels(pwm->slice, 128, 128);
pwm_set_enabled(pwm->slice, true);
if (pwm->stream_dma_chan < 0) {
pwm->stream_dma_chan = dma_claim_unused_channel(true);
if (pwm->stream_dma_chan < 0) {
mp_raise_msg(&mp_type_RuntimeError, MP_ERROR_TEXT("Failed to claim a DMA channel for audio
}
}
dma_channel_config c = dma_channel_get_default_config(pwm->stream_dma_chan);
channel_config_set_transfer_data_size(&c, DMA_SIZE_32);
channel_config_set_read_increment(&c, true);
channel_config_set_write_increment(&c, false);
channel_config_set_dreq(&c, pwm_get_dreq(pwm->slice));
```

```
dma_channel_configure(
pwm->stream_dma_chan, &c,
&pwm_hw->slice[pwm->slice].cc,
NULL, // Source address will be set by audio_play_chunk
0,    // Transfer count will be set by audio_play_chunk
false // Do not trigger now
);
audio_is_configured = true;
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_2(audio_configure_obj, audio_configure);
static mp_obj_t audio_play_chunk(mp_obj_t pwm_obj, mp_obj_t buf_obj) {
machine_pwm_obj_t *pwm = MP_OBJ_TO_PTR(pwm_obj);
if (!audio_is_configured || pwm->stream_dma_chan < 0) {
mp_raise_msg(&mp_type_RuntimeError, MP_ERROR_TEXT("Audio not configured or DMA channel n
claimed"));
}
mp_buffer_info_t bufinfo;
mp_get_buffer_raise(buf_obj, &bufinfo, MP_BUFFER_READ);
if (bufinfo.typecode != 'L') {
mp_raise_ValueError(MP_ERROR_TEXT("Buffer must be of typecode 'L'."));
}
dma_channel_abort(pwm->stream_dma_chan);
// 2. Get a clean, default configuration block.
dma_channel_config c = dma_channel_get_default_config(pwm->stream_dma_chan);
// 3. Re-populate the entire configuration.
channel_config_set_transfer_data_size(&c, DMA_SIZE_32);
channel_config_set_read_increment(&c, true);
channel_config_set_write_increment(&c, false);
channel_config_set_dreq(&c, pwm_get_dreq(pwm->slice));
// 4. Atomically apply the full configuration and trigger the transfer.
dma_channel_configure(
pwm->stream_dma_chan,
&c,
&pwm_hw->slice[pwm->slice].cc, // Write address
bufinfo.buf,                   // Read address (the new buffer)
bufinfo.len / 4,               // Transfer count
true                          // Trigger immediately
);
dma_channel_set_read_addr(pwm->stream_dma_chan, bufinfo.buf, false);
dma_channel_set_trans_count(pwm->stream_dma_chan, bufinfo.len / 4, true); // true = trigger now
pwm->is_streaming = true;
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_2(audio_play_chunk_obj, audio_play_chunk);
static mp_obj_t audio_wait_done(mp_obj_t pwm_obj) {
machine_pwm_obj_t *pwm = MP_OBJ_TO_PTR(pwm_obj);
if (pwm->stream_dma_chan >= 0 && dma_channel_is_busy(pwm->stream_dma_chan)) {
dma_channel_wait_for_finish_blocking(pwm->stream_dma_chan);
}
```

```
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_1(audio_wait_done_obj, audio_wait_done);
static mp_obj_t audio_deinit(mp_obj_t pwm_obj) {
machine_pwm_obj_t *pwm = MP_OBJ_TO_PTR(pwm_obj);
// Wait for any final transfer to complete.
if (pwm->stream_dma_chan >= 0) {
dma_channel_wait_for_finish_blocking(pwm->stream_dma_chan);
}
// Unclaim the channel ONLY when we are finished ---
if (pwm->stream_dma_chan >= 0) {
dma_channel_unclaim(pwm->stream_dma_chan);
pwm->stream_dma_chan = -1;
}
if (pwm->is_streaming) {
pwm_set_chan_level(pwm->slice, pwm->channel, 128); // Set to silence
pwm->is_streaming = false;
}
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_1(audio_deinit_obj, audio_deinit);
static mp_obj_t sdr_radio_start_capture(mp_obj_t self_in) {
sdr_radio_obj_t *self = MP_OBJ_TO_PTR(self_in);
(void)self;
adc_fifo_drain();
dma_start_channel_mask(1u << adc_dma_chan_A);
adc_run(true);
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_1(sdr_radio_start_capture_obj, sdr_radio_start_capture);
static mp_obj_t configure_transmitter_pwm(mp_obj_t pwm_obj, mp_obj_t update_rate_obj, mp_obj_t
top_obj) {
machine_pwm_obj_t *pwm = MP_OBJ_TO_PTR(pwm_obj);
uint32_t update_rate = mp_obj_get_int(update_rate_obj);
uint32_t top = mp_obj_get_int(top_obj);
pwm_set_enabled(pwm->slice, false);
uint32_t source_hz = clock_get_hz(clk_sys);
float div = (float)source_hz / ((float)(top + 1) * (float)update_rate);
if (div < 1.0f) div = 1.0f;
pwm_set_clkdiv(pwm->slice, div);
pwm_set_wrap(pwm->slice, top);
pwm_set_chan_level(pwm->slice, pwm->channel, 0);
pwm_set_enabled(pwm->slice, true);
audio_is_configured = true;
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_3(configure_transmitter_pwm_obj, configure_transmitter_pwm
static mp_obj_t sdr_radio_set_tx_carrier(mp_obj_t self_in, mp_obj_t freq_obj, mp_obj_t pwm_rate_obj)
{
sdr_radio_obj_t *self = MP_OBJ_TO_PTR(self_in);
```

```
self->tx_carrier_freq_hz = mp_obj_get_int(freq_obj);
uint32_t pwm_update_rate = mp_obj_get_int(pwm_rate_obj);
self->tx_nco_phase_increment = (q31_t)(((uint64_t)self->tx_carrier_freq_hz << 31) /
pwm_update_rate);
return mp_const_none;
}
static MP_DEFINE_CONST_FUN_OBJ_3(sdr_radio_set_tx_carrier_obj, sdr_radio_set_tx_carrier);
static mp_obj_t sdr_radio_am_transmit_pipeline(mp_obj_t self_in, mp_obj_t audio_buf_obj, mp_obj_t
pwm_buf_obj) {
sdr_radio_obj_t *self = MP_OBJ_TO_PTR(self_in);
mp_buffer_info_t audio_info; mp_get_buffer_raise(audio_buf_obj, &audio_info, MP_BUFFER_READ);
mp_buffer_info_t pwm_info;   mp_get_buffer_raise(pwm_buf_obj,   &pwm_info,   MP_BUFFER_WRITE
uint16_t *audio_in_ptr = (uint16_t *)audio_info.buf;
uint32_t *pwm_out_ptr = (uint32_t *)pwm_info.buf;
const int num_audio_samples = audio_info.len / sizeof(uint16_t);
const int num_pwm_words = pwm_info.len / sizeof(uint32_t);
// --- Constants ---
const int PWM_TOP = 49;    // Must match Python
const int PWM_CENTER = 25; // (PWM_TOP + 1) / 2
const float MODULATION_DEPTH = 0.95f;
// These must match your Python script's constants
const uint32_t PWM_UPDATE_RATE = 5000000;
// const uint32_t AUDIO_SAMPLE_RATE = 22050;
const float RATIO = (float)PWM_UPDATE_RATE / (float)AUDIO_SAMPLE_RATE;
int pwm_idx = 0;
// --- Main Processing Loop ---
for (int i = 0; i < num_audio_samples; i++) {
// 1. Get current and next audio sample for linear interpolation
float audio_start = ((float)audio_in_ptr[i] - 2048.0f) / 2048.0f;
float audio_end = (i + 1 < num_audio_samples) ?
((float)audio_in_ptr[i + 1] - 2048.0f) / 2048.0f :
audio_start;
// 2. Linear Interpolation loop
// Calculate how many PWM samples this one audio sample covers
int start_j_idx = (int)(i * RATIO);
int end_j_idx = (int)((i + 1) * RATIO);
for (int pwm_sample_idx = start_j_idx; pwm_sample_idx < end_j_idx; pwm_sample_idx++) {
if (pwm_idx >= num_pwm_words) break; // Safety break
float interp_point = (float)(pwm_sample_idx - start_j_idx) / (float)(end_j_idx - start_j_idx);
float audio_interp = audio_start * (1.0f - interp_point) + audio_end * interp_point;
// 3. Generate carrier sample
q31_t carrier_q31 = arm_cos_q31(self->tx_nco_phase);
self->tx_nco_phase += self->tx_nco_phase_increment;
float carrier_float = (float)carrier_q31 / 2147483648.0f;
// 4. Modulate
float modulator = 1.0f + (audio_interp * MODULATION_DEPTH);
float am_signal = carrier_float * modulator;
// 5. Scale to PWM duty cycle
int32_t duty_cycle = (int32_t)(PWM_CENTER * (1.0f + am_signal));
// 6. Clamp
```

```
if (duty_cycle > PWM_TOP) duty_cycle = PWM_TOP;
if (duty_cycle < 0) duty_cycle = 0;
// 7. Pack two 16-bit duty cycles into one 32-bit word for the DMA
pwm_out_ptr[pwm_idx] = ((uint32_t)duty_cycle << 16) | (uint32_t)duty_cycle;
pwm_idx++;
}
}
return mp_const_none;
}
// CORRECTED: Macro for a function with 3 args (self, audio_buf, pwm_buf)
static MP_DEFINE_CONST_FUN_OBJ_3(sdr_radio_am_transmit_pipeline_obj,
sdr_radio_am_transmit_pipeline);
static const mp_rom_map_elem_t sdr_radio_locals_dict_table[] = {
{ MP_ROM_QSTR(MP_QSTR_tune), MP_ROM_PTR(&sdr_radio_tune_obj) },
{ MP_ROM_QSTR(MP_QSTR_reset_state), MP_ROM_PTR(&sdr_radio_reset_state_obj) },
{ MP_ROM_QSTR(MP_QSTR_set_mode), MP_ROM_PTR(&sdr_radio_set_mode_obj) },
{ MP_ROM_QSTR(MP_QSTR_fast_sdr_pipeline), MP_ROM_PTR(&fast_sdr_pipeline_obj) },
{ MP_ROM_QSTR(MP_QSTR_set_bfo), MP_ROM_PTR(&sdr_radio_set_bfo_obj) },
{ MP_ROM_QSTR(MP_QSTR_capture_chunk), MP_ROM_PTR(&sdr_radio_capture_chunk_obj) },
{ MP_ROM_QSTR(MP_QSTR_start_capture), MP_ROM_PTR(&sdr_radio_start_capture_obj) },
{ MP_ROM_QSTR(MP_QSTR_configure_and_init_capture),
MP_ROM_PTR(&sdr_radio_configure_and_init_capture_obj) },
{ MP_ROM_QSTR(MP_QSTR_set_tx_carrier), MP_ROM_PTR(&sdr_radio_set_tx_carrier_obj) },
{ MP_ROM_QSTR(MP_QSTR_am_transmit_pipeline), MP_ROM_PTR(&sdr_radio_am_transmit_pipeli
};
static MP_DEFINE_CONST_DICT(sdr_radio_locals_dict, sdr_radio_locals_dict_table);
const mp_obj_type_t sdr_radio_SDR_Radio_type;
MP_DEFINE_CONST_OBJ_TYPE( sdr_radio_SDR_Radio_type, MP_QSTR_SDR_Radio, MP_TYPE_
sdr_radio_make_new, locals_dict, &sdr_radio_locals_dict );
static const mp_rom_map_elem_t sdr_radio_module_globals_table[] = {
{ MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_sdr_radio) },
{ MP_ROM_QSTR(MP_QSTR_SDR_Radio), MP_ROM_PTR(&sdr_radio_SDR_Radio_type) },
{ MP_ROM_QSTR(MP_QSTR_deinit_capture), MP_ROM_PTR(&sdr_radio_deinit_capture_obj) },
{ MP_ROM_QSTR(MP_QSTR_configure_transmitter_pwm), MP_ROM_PTR(&configure_transmitter_p
{ MP_ROM_QSTR(MP_QSTR_audio_configure), MP_ROM_PTR(&audio_configure_obj) },
{ MP_ROM_QSTR(MP_QSTR_audio_play_chunk), MP_ROM_PTR(&audio_play_chunk_obj) },
{ MP_ROM_QSTR(MP_QSTR_audio_wait_done), MP_ROM_PTR(&audio_wait_done_obj) },
{ MP_ROM_QSTR(MP_QSTR_audio_deinit), MP_ROM_PTR(&audio_deinit_obj) },
};
static MP_DEFINE_CONST_DICT(sdr_radio_module_globals, sdr_radio_module_globals_table);
const mp_obj_module_t sdr_radio_user_cmodule = { .base = { &mp_type_module }, .globals =
(mp_obj_dict_t *)&sdr_radio_module_globals, };
#if MICROPY_PY_SDR_RADIO
MP_REGISTER_MODULE(MP_QSTR_sdr_radio, sdr_radio_user_cmodule);
#endif
```

That’s a lot of code to go over. But much of it is boilerplate used to connect C to MicroPython. As a guide to explaining the C code, we will walk through the Python code for the radio. As you would expect, it is much smaller and simpler:

```
from array import array
from machine import Pin, ADC, PWM, freq
```

```
from sdr_radio import SDR_Radio
from sdr_radio import deinit_capture
from sdr_radio import audio_configure, audio_play_chunk, audio_wait_done, audio_deinit
from sys import print_exception
freq(250_000_000)
ADC_PIN = 26
PWM_PIN = 20
BUFFER_SIZE_SAMPLES = 8192
ADC_SAMPLE_RATE = 500000
AUDIO_SAMPLE_RATE = 22050
TARGET_FREQ = 810_000.0
BUFFERS_PER_SECOND = ADC_SAMPLE_RATE / BUFFER_SIZE_SAMPLES
led = Pin("LED", Pin.OUT)
led.off()
adc = ADC(Pin(ADC_PIN))
pwm = PWM(Pin(PWM_PIN))
audio_configure(pwm, AUDIO_SAMPLE_RATE)
DECIMATION_FACTOR = int(ADC_SAMPLE_RATE // AUDIO_SAMPLE_RATE)
audio_rate_len = BUFFER_SIZE_SAMPLES // DECIMATION_FACTOR
sdr = SDR_Radio()
sdr.configure_and_init_capture(ADC_SAMPLE_RATE, BUFFER_SIZE_SAMPLES)
sdr.start_capture()
def radio(f):
sdr.set_mode(True)
sdr.tune(int(f))
adc_buf = array('H', (0 for _ in range(BUFFER_SIZE_SAMPLES)))
pwm_long_bufs = [
array('L', (0 for _ in range(audio_rate_len))),
array('L', (0 for _ in range(audio_rate_len)))
]
scratch_buf = array('l', (0 for _ in range(BUFFER_SIZE_SAMPLES * 2)))
buf_idx = 0
sdr.capture_chunk(adc_buf)
sdr.fast_sdr_pipeline([adc_buf, pwm_long_bufs[0], scratch_buf])
audio_play_chunk(pwm, pwm_long_bufs[0])
while True:
next_buf_idx = 1 - buf_idx
sdr.capture_chunk(adc_buf)
sdr.fast_sdr_pipeline([adc_buf, pwm_long_bufs[buf_idx], scratch_buf])
audio_play_chunk(pwm, pwm_long_bufs[buf_idx])
buf_idx = next_buf_idx
def main():
try:
radio(810_000)
except KeyboardInterrupt:
print("\nUser interrupt.")
except Exception as e:
print("An exception occurred:")
print_exception(e)
finally:
```

```
print("Cleaning up...")
# Wait for any final chunk to finish before stopping
audio_wait_done(pwm)
audio_deinit(pwm)
# This will stop the ADC and release DMA channels
deinit_capture()
led.value(0)
print("Done.")
main()
```

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.

Now we have an audio signal, but it is sampled at 500 kHz, which is much too fast for our PWM, which would like 22050 Hz. So we "decimate". We just keep every 22nd sample, and we're at 22050.

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.

Now we are ready for the Automatic Gain Control (AGC). We want weak faraway stations to sound as loud as strong nearby stations. If the signal is too strong, we reduce it. Too weak, and we increase it.

Now we want to send it to the PWM. The PWM wants to see 32-bit integers, where the high 16 bits are for the B channel, and the low 16 bits are for the A channel. Only the low byte is actually used for either channel. We send 128 to B, and our sample to A. This keeps the B channel quiet.

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.

Then it calls sdr.start_capture() to start filling the two "ping-pong" buffers for the DMA output.

The ADC DMA is now running in the background, without bothering the CPU, filling first one buffer and then the other. This allows the CPU to read from the full buffer while the other one is being filled. Thus "ping pong".

Next, we create a buffer for the ADC and two buffers for the PWM (so we can do a ping-pong with those in Python).
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().
Once primed, we enter the main loop. We capture, pipeline, wait for the PWM to finish playing the primed audio, and then hand it another buffer to play.
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.
The code I presented above also includes the ability to handle CW and single-sideband reception, and it has an AM transmitter. But as this has been a long slog for you to read, we’ll play with those another day.
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

9 min read
.
Dec 1, 2024

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
One of the least expensive computers that can run Micropython is the ESP8266–01S. I get them from AliExpress literally by the handful — they cost $11.68 for ten of them, and they arrive in about ten days.
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.

That's a lot of computer for $1.17.

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 wa 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

To program the 01S, we will need a separate USB to UART adapter. Search for "ESP8266–01 program You can find them on AliExpress for less than a dollar. I got a couple from Amazon for ten times that, but they arrived the next day.

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 –veri
```

Micropython

There are three versions of the ESP8266 Micropython. One is for the boards that have 4 megabits of flash, one for those with 512 kilobits of flash, and one for those (like our 01S) that have a megabit of flash.

Download that last one, and rename it esp8266_mpy_1M.bin, and then run "setup com3". Your com po will probably be some other number.

The Python code

```
Def beep(f):
From machine import PWM, Pin
From time import sleep_ms
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():
From network import WLAN, STA_IF, AP_IF, AUTH_OPEN
```

```
From time import sleep_ms
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():
Machine.idle() # save power while waiting
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 second Wi-Fi connection is a station. We scan for access points looking for any that start with "Fox ". When we find a fox, we connect to it.

In this way, the same code works for both the fox and the fox hunter (the hound).

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

The hardware part of the receiver is pretty simple. We build a little wiring harness for the computer, battery, and speaker:

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
The wiring for the transmitter is even simpler.
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.
When you plug in the IO0 wire on the receiver, it takes a little bit of time to find the signal from the fox and connect to it. After that, it starts beeping. If your body is between the receiver and the transmitter, the pitch will drop, so you know which direction to walk. The players don't know what you hid the transmitter in, so they may hunt around for a while, even if the pill bottle is in plain sight.
A poor man's dish
Blocking the signal with your body is a time-tested radio direction finding trick. But we can do better.
We discussed how to make Yagi-Uda directional antennas in an earlier part of this series. We could do that for 2.4 gigahertz, but instead, let's do something new.
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

Hunting Wi-Fi transmitters is fun, and might even be useful if you want to locate the best place to sit in a cafe with free Wi-Fi (you just change the “Fox ” in the code to the SSID you are looking for).

But Wi-Fi only goes so far. To make the game more challenging, we can use a transmitter that can be heard for a mile. Our simple Morse Code transmitter from earlier in the series is just the thing.

It uses the FS1000A transmitter, and optionally a ground plane antenna. We change the main.py code just a little:

```
From morse import Morse
From machine import Pin
From network import AP_IF, STA_IF, WLAN
From esp import deepsleep, sleep_type, SLEEP_MODEM
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
# Now the user must ground the reset pin
# After reset, we get here with the WIFI radio turned off to save power.
# Power went from 80 mA down to 18.85 (and 21 mA when transmitting using FS1000A)
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.

Thankfully, any UHF FM radio receiver or transceiver can pick up the signal and give us an RSSI reading, usually as a number of bars on the screen. The BAOFENG transceiver is only $25, and works fine for this task.

We can use our Yagi-Uda antenna, or we can use a more robust commercial antenna with its much low 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
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.
There is a very nice little remote control available on AliExpress.com for three dollars. It comes with the receiver. Look for “433 MHz Wireless RF Remote Control Switch EV1527 Learning Code 4CH Relay Receiver Module and On Off Transmitter For DIY Kit”.
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.
Python Code for the RTL-SDR
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():
From rtlsdr import RtlSdr
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
# print(sdr.valid_gains_db)
Sdr.gain = 49.6
# print(sdr.gain)
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”)
# Look for a 8,000 sample quiet period
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.xticks([x*2048 for x in range(50)], [x for x in range(50)])
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.xticks([x*2048 for x in range(50)], [x for x in range(50)])
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’: “01100000110000111111110000”,
‘B’: “01100000110000111111101000”,
‘C’: “01100000110000111111100100”,
‘D’: “01100000110000111111100010”,
}
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
Let’s build a program for the ESP32C3 Super Mini and the CC1101 to both sniff the data from the remote and send the four codes to the receiver:

```
Import gc
From machine import Pin, SPI, disable_irq, enable_irq
From time import sleep, sleep_ms, sleep_us, ticks_diff, ticks_ms, ticks_us
From sys import print_exception
From whoami import WhoAmI
BIT_DECODE_SHORT_MIN = 200  # Min duration for a short pulse
BIT_DECODE_SHORT_MAX = 700  # Max duration for a short pulse
BIT_DECODE_LONG_MIN  = 800  # Min duration for a long pulse
BIT_DECODE_LONG_MAX  = 1300 # Max duration for a long pulse
REMOTE_BUTTON_CODES = {
‘A’: “01100000110000111111110000”,
‘B’: “01100000110000111111101000”,
‘C’: “01100000110000111111100100”,
‘D’: “01100000110000111111100010”,
}
# Configuration Registers
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
# Status Registers
PARTNUM = 0xF0; VERSION = 0xF1; MARCSTATE = 0xF5; RSSI = 0xF4; LQI = 0xF3
TXBYTES = 0xFA; RXBYTES = 0xFB; PKTSTATUS = 0xF8
# Strobe Commands
SRES = 0x30; SRX = 0x34; STX = 0x35; SIDLE = 0x36; SCAL = 0x33
SFRX = 0x3A; SFTX = 0x3B; SNOP = 0x3D
# PATABLE and FIFO Addresses
PATABLE_ADDR = 0x3E; TXFIFO_ADDR = 0x3F; RXFIFO_ADDR = 0x3F
# SPI Header Bits
WRITE_SINGLE_BYTE = 0x00; READ_SINGLE_BYTE = 0x80; WRITE_BURST = 0x40; READ_BURS
# States:
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:
Raise ValueError(“GDO0 pin is required for sniffing mode.”)
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]
Self.cs.on(); sleep_us(50); return val
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]
Self.cs.on(); sleep_us(50); return val
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:]
Self.cs.on(); sleep_us(50); return data
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 marc_state == 0x01: return
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)
# print(f"Frequency is {freq_hz}")
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:0
Self.current_mode = “ERROR_SNIFF_CONFIG”
Def configure_for_ask_tx(self, pa_table_val=0xC0):
Print(“Configuring CC1101 for ASK/OOK Bit-Bang Transmission (PKTCTRL0=0x02)...”)
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.

There is a format that phones used to use for storing their ringtones. It is called the Ring Tone Text Transfer Language, or RTTTL. Dave Hylands wrote a parser for this language for MicroPython, which you can see here.

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:

```
From rtttl import RTTTL
From time import sleep_ms, sleep
From machine import Pin, PWM
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
Sleep_ms(int(0.9 * msec))     # Play for a number of msec
Key.duty(0)                 # Stop playing for gap between notes
Sleep_ms(int(0.1 * msec))     # Pause for a number of msec
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 import Dave’s RTTTL class from the module rtttl.py.
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.
Dave’s RTTTL class hands us pairs of frequency and time to hand to our play_tone function.
We sleep for a second in between songs.
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: .
,a.,g#.,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.
c,d,f.,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
,8c,f,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
2g#.,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
=2,o=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#.
16f#6,16g#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
,32e,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,2b
6c#7,16g#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
32p,e,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.
b.5,a#.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
d.,g,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
16c6.,p.,16a#,p.,2c6.,p.,16c6,p.,16c#6,p.,16c6.,p.,16a#,p.,4c6.,p.,16a#,p.,16g#,p.,4a#.,16g,16f.,d#.
b6,16a6,16p,16d6,16p,16d6,16d6,16d6,16d6,16d6,16d6,16d6,16d6,16a,16p,16c6,16d6,16p,16d6,16c6,16b,
5,b=100:8f#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#6,16
6g,4f,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
2c#.,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#.
8e#6,16p,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#6,1
,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,d,p,4d.,d,e,4f.,p,f,f,e,e,d
5,16d#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
6,2e6,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
,32e5,8d#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
,o=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

5,b=125:4d6,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,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
6,b=45:16g.5,16g.5,16g5,16g.5,32e5,8g.5,16e5,32d5,16e5,8g.5,32d,8b.5,32f#5,16e5,32f#5,16g5,32b5
f#.,32f.,32f#,32g#.,32f#.,8f.,32f#.,32g#.,32a.,32b.,32a.,32g#.,32f#.,32f#.,32f.,32f#,32g#.,32f#.,8f.
,2e7,2d7,4b.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
,4g,g6,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.
,16p,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
5,8f#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#5,4c
,8f#,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
p.,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
g6,8p,16g6,8p,16a#6,16p,16c7,16p,16a#6,16g6,2d6,32p,16a#6,16g6,2c#6,32p,16a#6,16g6,2c6,16p,1
larB: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,
d#,8p,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
p,e,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.
,a#,g,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#
g5,a#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
,16d#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
Transmitter
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’: ‘--..’,
‘0’: ‘-----’,
‘1’: ‘.----’,
‘2’: ‘..---‘,
‘3’: ‘...—‘,
‘4’: ‘....-‘,
‘5’: ‘.....’,
‘6’: ‘-....’,
‘7’: ‘--...’,
‘8’: ‘---..’,
‘9’: ‘----.’,
‘.’: ‘.-.-.-‘,
‘,’: ‘--..—‘,
‘?’: ‘..--..’,
‘\’’: ‘·————·’,
‘!’: ‘–·–·—‘,
‘/’: ‘–··–·’,
‘(‘: ‘–·—·’,
‘)’: ‘–·—··–‘,
‘&’: ‘·–···’,
‘:’: ‘———···’,
‘;’: ‘–·–·–·’,
‘=’: ‘–···–‘,
‘+’: ‘·–·–·’,
‘-‘: ‘–····–‘,
‘_’: ‘··——·–‘,
‘”’: ‘·–··–·’,
‘$’: ‘···–··–‘,
‘@’: ‘·——·–·’,
}
We will put that into a file called the_code.py.

Now we create a class to hold all the data and methods needed to make our Morse code transmitter function:

```
From machine import Pin, PWM
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 timing from
Farnsworth_spacing = (60000 * self.character_speed – 37200 * overall_speed) / (overall_speed * self.character_speed)
Farnsworth_spacing *= 60000/68500    # A fudge factor to get the ESP8266 timing closer to correct
Self.LETTER_SPACE = int((3 * farnsworth_spacing) / 19) – self.CYPHER_SPACE
Self.WORD_SPACE = int((7 * farnsworth_spacing) / 19) – self.CYPHER_SPACE
Def send(self, str):
From the_code import code
From time import sleep_ms
For c in str:
If c == ‘ ‘:
Self.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)
```

Let’s look at the send() method first. It accepts the string of characters we wish to transmit. For each character in the string, we look it up in the code table and get the string of dots and dashes for that letter.

Then we walk through the string of dots and dashes, turning on the PWM pin for the length of a dot

or a dash.
After each dot or dash, we turn off the PWM pin and wait for a time equal to a dot.
When we are done with the string of dots and dashes, we wait for the proper time to indicate that we are done with the letter.
Half of the code in the class worries about setting the proper delays for the right speed. This is because humans find it easiest to hear Morse characters by the rhythm of the dots and dashes, and that is actually easier when the dots and dashes are happening quickly. For those who have not learned Morse at higher speeds, we add a delay between the characters to give the person time to recall what letter that sound was.
This is called the Farnsworth method for learning Morse code. Letters are always sent at least 18 words per minute, even if the overall speed is only something like 5 words per minute or less.
We put the Morse class into a file called morse.py.
The main routine is fairly simple:

```
From morse import Morse
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.
Since it asks for input, we will need to access the Python REPL (Read, Evaluate, Print Loop) using a terminal emulator. I use the Putty program on Windows, but Linux and Macs have their own terminal emulators.
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?
```

Note that it says it has started the WebREPL. This is because of the web_repl.py file we loaded onto the ESP8266. Because of this, you can connect to our little computer over Wi-Fi. Look for a Wi-Fi SSID that looks like MicroPython-xxxxxx where the x characters are hexadecimal characters. Connect to that Wi-Fi SSID, and you will be asked for a password. The web_repl.py file has the password in

it. Enter the password, and now your web browser can control the ESP8266 in the same way the terminal emulator does. So, you can talk to it even if it is battery-powered up on a pole or on the roof.
As before, you can hear the Morse code coming out of the speaker on the receiver we built in the previous Morse code project. This is one way to learn Morse code. You can send whole files and the little computer will send the Morse to anyone listening on their receivers. A whole class or a whole neighborhood can listen in on their receivers, which they can build for about a dollar.
The files for this project can be found here: .
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
In 1876, Alexander Graham Bell patented his system for sending voice information over a wire. One wire. The "return path" was the earth.
The earth acts like a great big bucket of electrons. It can both source and "sink" any amount of current. So while the current in the wire carried the signal, the earth at each end acted like a spring, able to push or pull electrons in either direction.
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.
That's it. Except for a bit of code.
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
The little board is tiny. The size of my thumbnail. You can see the USB-C connector on it for a size reference. Those holes for the pins are 1/10th of an inch apart. The speaker is 2 inches across (50 millimeters).

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.

Out here in the mountains, the mail delivery has no reliable schedule. I thought it would be nice to know when my latest parts for my next project have arrived, so I can dive right in and solder something up.

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

```
from machine import Pin, I2C, ADC
from SH1106 import SH1106_I2C
import urequests as requests
from network import WLAN, STA_IF
from time import sleep
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 = Display()
sensor = ADC(4)
gnd    = Pin(7, Pin.OUT)
led    = Pin(8, Pin.OUT)
gnd.value(0)
led.value(0)
station = WLAN(STA_IF)
station.active(True)
station.connect("BirdfarmOffice2", "")
def noise():
print("Hit!")
while True:
try:
requests.get(url="http://10.90.20.10:8143/")
break
except Exception as e:
print(e)
led.value(0)
sleep(.2)
```

```
led.value(1)
sleep(.2)
led.value(0)
sleep(.2)
led.value(1)
sleep(.2)
def main():
while station.isconnected() == False:
pass
d.oled("Mailbox:", 0, 0)
ip = station.ifconfig()[0]
gateway = station.ifconfig()[2]
print(station.ifconfig())
last = ip[9:]
d.oled(ip, 1, 0)
d.oled(last, 2, 0)
print(ip)
old_microvolts = 0
while True:
microvolts = sensor.read_uv()
if microvolts > 50000:
print(microvolts)
noise()
old_microvolts = microvolts
main()
```

The display on the Supermini uses the SH1106 driver. I'll add that later without discussion, but it is available on GitHub.

Our Display class handles all the peculiarities of the display. And it is definitely peculiar. The visible part of the display is a rectangle inside the frame buffer of the driver. The oled() method handles all the arithmetic to deal with that. It takes a string, a line, and a column as arguments.

We connect the speaker between pins 4 and 7. Pin 7 we will set at ground. Pin 4 is the ADC pin that will listen to the speaker. I didn't need pin 7 for anything, and it was more convenient than soldering to the GND pin (which I planned to use for the battery).

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.

The main routine waits for a Wi-Fi connection, then prints Mailbox on the screen, along with the IP address it has been given. The address is longer than 9 characters, so I had to split it into two lines.

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”.

```
from microdot import Microdot
from datetime import datetime
app = Microdot()
@app.route('/')
async def hit(r):
print(datetime.now(), "Device at", r.client_addr[0], "heard a loud noise. You may have mail.")
return "OK"
app.run(port=8143)
```

If a connection comes in on port 8143 it prints a message with a timestamp.

The timestamp lets us decide if the alarm is the mail or some other event that made a loud noise. We could add some checks to see if the alert is in the time window that makes sense for the mail, but I think it would be fun to know how many times a loud noise happened out there.

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.
```

The gadget has been running for a couple of days on the 1800 milliampere battery. I could run it from a USB-C charger, but my long-term plan is to run it on solar power. I am just waiting for my solar battery charger boards to arrive. With a big enough solar panel, I wouldn’t need a battery, since it would be enough to run the gadget on a cloudy day, and the mail doesn’t arrive at night.

I promised the code for the SS1106 driver:

```
#
# MicroPython SH1106 OLED driver, I2C and SPI interfaces
#
```

```
# The MIT License (MIT)
#
# Copyright (c) 2016 Radomir Dopieralski (@deshipu),
#               2017-2021 Robert Hammelrath (@robert-hh)
#               2021 Tim Weber (@scy)
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
#
# Sample code sections for ESP8266 pin assignments
# ------------ SPI ------------------
# Pin Map SPI
#   - 3v - xxxxxx   - Vcc
#   - G  - xxxxxx   - Gnd
#   - D7 - GPIO 13  - Din / MOSI fixed
#   - D5 - GPIO 14  - Clk / Sck fixed
#   - D8 - GPIO 4   - CS (optional, if the only connected device)
#   - D2 - GPIO 5   - D/C
#   - D1 - GPIO 2   - Res
#
# for CS, D/C and Res other ports may be chosen.
#
# from machine import Pin, SPI
# import sh1106
# spi = SPI(1, baudrate=1000000)
# display = sh1106.SH1106_SPI(128, 64, spi, Pin(5), Pin(2), Pin(4))
# display.sleep(False)
# display.fill(0)
# display.text('Testing 1', 0, 0, 1)
# display.show()
#
# ---------- I2C ------------------
#
# Pin Map I2C
#   - 3v - xxxxxx   - Vcc
```

```
#   - G  - xxxxxx   - Gnd
#   - D2 - GPIO 5   - SCK / SCL
#   - D1 - GPIO 4   - DIN / SDA
#   - D0 - GPIO 16  - Res
#   - G  - xxxxxx     CS
#   - G  - xxxxxx     D/C
#
# Pin's for I2C can be set almost arbitrary
#
# from machine import Pin, I2C
# import sh1106
#
# i2c = I2C(scl=Pin(5), sda=Pin(4), freq=400000)
# display = sh1106.SH1106_I2C(128, 64, i2c, Pin(16), 0x3c)
# display.sleep(False)
# display.fill(0)
# display.text('Testing 1', 0, 0, 1)
# display.show()
from micropython import const
import utime as time
import framebuf
# a few register definitions
_SET_CONTRAST        = const(0x81)
_SET_NORM_INV        = const(0xa6)
_SET_DISP            = const(0xae)
_SET_SCAN_DIR        = const(0xc0)
_SET_SEG_REMAP       = const(0xa0)
_LOW_COLUMN_ADDRESS  = const(0x00)
_HIGH_COLUMN_ADDRESS = const(0x10)
_SET_PAGE_ADDRESS    = const(0xB0)
class SH1106(framebuf.FrameBuffer):
def __init__(self, width, height, external_vcc, rotate=0):
self.width = width
self.height = height
self.external_vcc = external_vcc
self.flip_en = rotate == 180 or rotate == 270
self.rotate90 = rotate == 90 or rotate == 270
self.pages = self.height // 8
self.bufsize = self.pages * self.width
self.renderbuf = bytearray(self.bufsize)
self.pages_to_update = 0
if self.rotate90:
self.displaybuf = bytearray(self.bufsize)
# HMSB is required to keep the bit order in the render buffer
# compatible with byte-for-byte remapping to the display buffer,
# which is in VLSB. Else we'd have to copy bit-by-bit!
super().__init__(self.renderbuf, self.height, self.width,
framebuf.MONO_HMSB)
else:
self.displaybuf = self.renderbuf
```

```
super().__init__(self.renderbuf, self.width, self.height,
framebuf.MONO_VLSB)
# flip() was called rotate() once, provide backwards compatibility.
self.rotate = self.flip
self.init_display()
def init_display(self):
self.reset()
self.fill(0)
self.show()
self.poweron()
# rotate90 requires a call to flip() for setting up.
self.flip(self.flip_en)
def poweroff(self):
self.write_cmd(_SET_DISP | 0x00)
def poweron(self):
self.write_cmd(_SET_DISP | 0x01)
if self.delay:
time.sleep_ms(self.delay)
def flip(self, flag=None, update=True):
if flag is None:
flag = not self.flip_en
mir_v = flag ^ self.rotate90
mir_h = flag
self.write_cmd(_SET_SEG_REMAP | (0x01 if mir_v else 0x00))
self.write_cmd(_SET_SCAN_DIR | (0x08 if mir_h else 0x00))
self.flip_en = flag
if update:
self.show(True) # full update
def sleep(self, value):
self.write_cmd(_SET_DISP | (not value))
def contrast(self, contrast):
self.write_cmd(_SET_CONTRAST)
self.write_cmd(contrast)
def invert(self, invert):
self.write_cmd(_SET_NORM_INV | (invert & 1))
def show(self, full_update = False):
# self.* lookups in loops take significant time (~4fps).
(w, p, db, rb) = (self.width, self.pages,
self.displaybuf, self.renderbuf)
if self.rotate90:
for i in range(self.bufsize):
db[w * (i % p) + (i // p)] = rb[i]
if full_update:
pages_to_update = (1 << self.pages) - 1
else:
pages_to_update = self.pages_to_update
#print("Updating pages: {:08b}".format(pages_to_update))
for page in range(self.pages):
if (pages_to_update & (1 << page)):
self.write_cmd(_SET_PAGE_ADDRESS | page)
```

```
self.write_cmd(_LOW_COLUMN_ADDRESS | 2)
self.write_cmd(_HIGH_COLUMN_ADDRESS | 0)
self.write_data(db[(w*page):(w*page+w)])
self.pages_to_update = 0
def pixel(self, x, y, color=None):
if color is None:
return super().pixel(x, y)
else:
super().pixel(x, y , color)
page = y // 8
self.pages_to_update |= 1 << page
def text(self, text, x, y, color=1):
super().text(text, x, y, color)
self.register_updates(y, y+7)
def line(self, x0, y0, x1, y1, color):
super().line(x0, y0, x1, y1, color)
self.register_updates(y0, y1)
def hline(self, x, y, w, color):
super().hline(x, y, w, color)
self.register_updates(y)
def vline(self, x, y, h, color):
super().vline(x, y, h, color)
self.register_updates(y, y+h-1)
def fill(self, color):
super().fill(color)
self.pages_to_update = (1 << self.pages) - 1
def blit(self, fbuf, x, y, key=-1, palette=None):
super().blit(fbuf, x, y, key, palette)
self.register_updates(y, y+self.height)
def scroll(self, x, y):
# my understanding is that scroll() does a full screen change
super().scroll(x, y)
self.pages_to_update =  (1 << self.pages) - 1
def fill_rect(self, x, y, w, h, color):
super().fill_rect(x, y, w, h, color)
self.register_updates(y, y+h-1)
def rect(self, x, y, w, h, color):
super().rect(x, y, w, h, color)
self.register_updates(y, y+h-1)
def register_updates(self, y0, y1=None):
# this function takes the top and optional bottom address of the changes made
# and updates the pages_to_change list with any changed pages
# that are not yet on the list
start_page = max(0, y0 // 8)
end_page = max(0, y1 // 8) if y1 is not None else start_page
# rearrange start_page and end_page if coordinates were given from bottom to top
if start_page > end_page:
start_page, end_page = end_page, start_page
for page in range(start_page, end_page+1):
self.pages_to_update |= 1 << page
```

```
def reset(self, res):
if res is not None:
res(1)
time.sleep_ms(1)
res(0)
time.sleep_ms(20)
res(1)
time.sleep_ms(20)
class SH1106_I2C(SH1106):
def __init__(self, width, height, i2c, res=None, addr=0x3c,
rotate=0, external_vcc=False, delay=0):
self.i2c = i2c
self.addr = addr
self.res = res
self.temp = bytearray(2)
self.delay = delay
if res is not None:
res.init(res.OUT, value=1)
super().__init__(width, height, external_vcc, rotate)
def write_cmd(self, cmd):
self.temp[0] = 0x80  # Co=1, D/C#=0
self.temp[1] = cmd
self.i2c.writeto(self.addr, self.temp)
def write_data(self, buf):
self.i2c.writeto(self.addr, b'\x40'+buf)
def reset(self):
super().reset(self.res)
class SH1106_SPI(SH1106):
def __init__(self, width, height, spi, dc, res=None, cs=None,
rotate=0, external_vcc=False, delay=0):
dc.init(dc.OUT, value=0)
if res is not None:
res.init(res.OUT, value=0)
if cs is not None:
cs.init(cs.OUT, value=1)
self.spi = spi
self.dc = dc
self.res = res
self.cs = cs
self.delay = delay
super().__init__(width, height, external_vcc, rotate)
def write_cmd(self, cmd):
if self.cs is not None:
self.cs(1)
self.dc(0)
self.cs(0)
self.spi.write(bytearray([cmd]))
self.cs(1)
else:
self.dc(0)
```

```
self.spi.write(bytearray([cmd]))
def write_data(self, buf):
if self.cs is not None:
self.cs(1)
self.dc(1)
self.cs(0)
self.spi.write(buf)
self.cs(1)
else:
self.dc(1)
self.spi.write(buf)
def reset(self):
super().reset(self.res)
```

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

Aircraft communication is a critical aspect of aviation that ensures the safe and efficient operation of aircraft. The International Telecommunication Union (ITU) has assigned aircraft analog voice dialogue in the High Frequency (HF) band between 3–30MHz and in the Very High Frequency (V band at 118–137 Mhz.VHF signals are only line-of-sight but offer much higher audio quality. This makes them ideal for aircraft communication where clear and immediate transmission is vital. In this context, VHF is often preferred despite its shorter range.

Press enter or click to view image in full size

Photo by Bao Menglong on Unsplash

Common Aircraft Communication Frequencies

APP (Approach Control Frequency: 120.05 MHz): This frequency is used by the approach control department to guide aircraft as they approach for a landing.

ATIS (Automatic Terminal Information Service: 128.05 MHz): This is a continuous broadcast of recorded non-control information in busier terminal areas. It’s essential for arriving and departing aircraft as it provides information like weather conditions, active runways, available approaches, and other necessary data.

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.

These frequencies is for Coimbatore International Airport .To find the aircraft control tower frequency at your local airport, go to the airport’s website and look for the air traffic control or pilot information section.

HDSDR

Press enter or click to view image in full size

HDSDR (High Definition Software Defined Radio) is a freeware Software Defined Radio (SDR) program

for Microsoft Windows. It can be used to listen to VHF aircraft communication.
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 suc 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?
Jamming is the act of intentionally interfering with a radio signal. It can be used to prevent a signal from being received, to corrupt the signal, or to make the signal difficult to understand. Jamming can be used for legitimate purposes, such as to prevent interference between different radio systems, but it can also be used for malicious purposes, such as to disrupt communications or to prevent navigation.
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-sigge 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, an two-tone signals.
-a hackrf: This option specifies the hardware to use, in this case, the HackRF One. The HackRF One is a software-defined radio (SDR) device that can be used to transmit and receive radio signals. It is a popular choice for jamming because it is relatively inexpensive and easy to use.

-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
GPS spoofing is a fascinating and somewhat controversial topic in the world of software-defined radio (SDR) and radio communications.GPS spoofing is the act of manipulating GPS signals to deceive a receiver into believing that it is located at a different location than it actually is. This can be done for a variety of purposes, such as to evade tracking or to disrupt critical infrastructure.
One way to spoof GPS signals is to use a software-defined radio (SDR) device such as the HackRF On
The HackRF One is a low-cost SDR device that can be used to transmit and receive radio signals across a wide frequency range. To spoof GPS signals with the HackRF One, you will need to install the gps-sdr-sim software. This software can be used to generate simulated GPS signals that can then be transmitted by the HackRF One. To start spoofing GPS signals, you will need to compile the gpssim.c file. Once the gpssim.c file has been compiled, you can use it to generate a simulated GPS signal file. This file can then be transmitted by the HackRF One using the hackrf_transfer command.
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 CDI archive. Make sure to download the most recent daily file.
NASA CDDIS Ephemeris Files:
Once you have the ephemeris file, you can use it to generate a simulated pseudorange and Doppler for the satellites in your range.
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
Don’t miss out on my upcoming articles! Follow me on Medium for more insightful content. Clap and share this article to spread the knowledge among fellow bug bounty hunters and cybersecurity enthusiasts.
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
HackrfOne is an open-source Software Defined Radio (SDR) peripheral that is capable of both transmitting and receiving radio signals. It operates in a wide frequency range from 1 MHz to 6 GHz, making it highly versatile for a variety of radio technologies.It can be used for replay attacks,Jamming Signals,Sniffing,Spoofing.
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 Raspb 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
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
DragonOS is a Linux distribution that's designed for software-defined radio (SDR) exploration. It comes with many open-source SDR programs pre-installed, including:
Kismet, Kismon, GNU Radio 3.10, GR-Iridium, GR-Tempest, GR-RDS.
DragonOS supports SDRs like the: RTL-SDR, HackRF, LimeSDR.
Press enter or click to view image in full size
https://cemaxecuter.com/ DOWNLOAD USING THIS LINK
LimeSDR
LimeSDR is a low-cost, open-source, apps-enabled software-defined radio (SDR) platform that can be used to support just about any type of wireless communication standard. It can transmit and receive UMTS, LTE, GSM, LoRa, Bluetooth, Zigbee, RFID, and Digital Broadcasting, among others. While mos SDRs have remained in the domain of RF and protocol experts, LimeSDR is usable by anyone familiar with the idea of an app store. This means you can easily download new LimeSDR apps from developer around the world. If you're a developer yourself, you can share and/or sell your LimeSDR apps through Snappy Ubuntu Core as well. The LimeSDR platform gives system developers, inventors, and even students an intelligent and flexible device for manipulating wireless signals, so they can learn, experiment, and develop products and applications.
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.
LimeSDR GPIO Board: This is an expansion board that provides individually settable, bi-directional level-shifted I/O for FPGA GPIO 0–7.
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.
BSC (Base Station Controller) The BSC acts as a primary controller for one or more BTS. It configures most of the parameters on the BTS and brings each one up on air after they're ready.
MSC (Mobile Switching Center) The MSC is responsible for routing voice calls and SMS. It sets up and releases end-to-end connections, handles mobility and hand-over requirements during the call, and manages billing and real-time prepaid account monitoring.
Media Gateway (MGW) A Media Gateway is a translation device or service that converts media stream between different telecommunications technologies such as 2G, 2.5G, 3G. One of its main functions is

to convert between different transmission and coding techniques.
Home Location Register (HLR) The HLR is a main database in a GSM network which saves all perman
information about a subscriber, for example, billing details, subscriber identity, current status in
the network, and many other things.
Press enter or click to view image in full size
Source:
Communication between SIM and HLR
The network and SIM both contain a secret key (K) for authentication. The key is never exposed to
the subscriber and never transmitted over the air. When a user wants to authenticate, the HLR
generates a RAND key, encrypts it with the (K) key, and generates a signed response known as SRES
The HLR also sends the same RAND key to the subscriber. The SIM encrypts this RAND key with the (
key and generates SRES. The SIM then sends the generated SRES to the network. If the SRES from t
SIM matches with the network-generated SRES, then the SIM is authenticated.
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 is updated each time a device moves to a new location. It also facilitates SMS by scanning
for the mobile switching center (MSC) used by the receiving party.
The HLR communicates with the SIM card in the following ways:
The HLR identifies the last known location of the device.
The HLR stores a teleservice list for voice and SMS.
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
Path Setup
export PATH=$PATH:/usr/src/iridium-toolkit
Command
./stats-voc.py FILE_NAME.parsed
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
Inmarsat is a British company that provides global mobile services through 14 geostationary satellites. These satellites allow for constant communication between the satellite and its corresponding ground station. Inmarsat’s services are vital for industries, governments, and aid agencies that need to communicate in remote areas or where there is no reliable terrestrial network. It’s especially valuable for the shipping, airline, and mining industries. Inmarsat was originally established as an intergovernmental organization in 1979 and was privatized in the late 1990s. Despite these changes, Inmarsat continues to play a crucial role in global communications. In

addition to its current fleet of satellites, Inmarsat plans to launch another seven satellites to further enhance its network. This will allow it to offer even more reliable and comprehensive services to its users around the globe. Inmarsat also provides services that support email, internet, video conferencing, and in-flight Wi-Fi. For example, the Inmarsat-5 (I-5) satellite is used primarily for mobile broadband communications for deep-sea vessels and in-flight connectivity for airline passengers. In November 2021, a deal was announced between Inmarsat's owners and Vias an American communications company. Viasat completed the acquisition of Inmarsat in May 2023.
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.
Operation Control Center (OCC): The OCC is responsible for the overall operation of the Inmarsat network. It monitors the performance of the network and coordinates with other components to ensure smooth operation.
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.
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
Download Scytale-C Plugin for SDR#
Downloads
Edit description
bitbucket.org
The Scytale-C Plugin for SDR# is a tool developed by Microp11 for decoding Inmarsat STD-C signals. It's currently in the pre-alpha stages, which means it may still be missing some functionality and could be buggy. However, it is functional at this point in time.
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

WebSDR, or Web Software-Defined Radio, is a revolutionary technology that allows multiple listeners to tune into a radio receiver connected to the internet simultaneously. This is a significant departure from traditional receivers available online, which do not offer this level of flexibility.
With WebSDR, each listener can tune independently, meaning they can listen to different signals at the same time. This is made possible by the advancements in SDR technology.In India, there are several active WebSDRs available. These include GRMS Bengaluru, GRMS New Delhi, New Delhi web GRMS Siliguri, GRMS Dimapur, and GRMS Coimbatore. Each of these WebSDRs offers unique featur covers different frequency ranges, providing a wide array of options for listeners.
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 :)
Don’t miss out on my upcoming articles! on Medium for more insightful content. Clap and share this article to spread the knowledge among fellow bug bounty hunters and cybersecurity enthusiasts.
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
Signal hunting
A fascinating aspect of software-defined radio (SDR) hacking, involves identifying and decoding radio signals.The is a key resource in this field, offering a comprehensive database of signals, complete with example sounds and waterfall images. This database is designed to work with a wide range of software-defined radios such as the RTL-SDR, Airspy, SDRPlay, HackRF, BladeRF, Funcube Dongle, and USRP among others. Currently, the site contains , making it an extensive guide for both beginners and seasoned SDR enthusiast.
Installing SDR#
Download SDR#
Pre-requisites: .NET 5 Desktop x86 Runtime
https://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)
In the world of software-defined radio (SDR) hacking, the Raspberry Pi stands out as a multi-functional tool.It can serve dual roles as both a transmitter for sending signals and an SDR server. By connecting an antenna to a general-purpose I/O (GPIO) pin, the Raspberry Pi can wirelessly transmit data via various modulations, including FM, AM, SSB, SSTV, and FSQ signals anywhere between. This makes it a powerful tool for signal transmission. On the other hand, by installing the RTL-SDR software, it can also serve as a remote networked SDR, similar to software like rtl_tcp and Spyserver. This allows users to connect to a remote RTL-SDR running the SDR++ server on a Raspberry Pi. The server can be set up to automatically run on boot, making it easy to use and efficient. This setup is compatible with almost any SDR and enables the full range of control options for RTL-SDRs.
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
RealVNC is a tool that allows you to remotely access the desktop environment of another computer. It's particularly useful in the context of a Raspberry Pi, as it's one of the easiest ways to remotely access it. If you use Raspberry Pi OS, VNC is preinstalled so you only have to enable it to get started. You can enable VNC on your Raspberry Pi via the system configuration or . Once enabled, you can install the on your computer and type the IP address of the Raspberry Pi to get connected to it. In terms of Software Defined Radio (SDR) hacking, an RTL-SDR dongle and Raspberry Pi can be us together for various radio-related projects. For instance, with an RTL-SDR dongle, Raspberry Pi, a piece of wire, and literally no other hardware, it is possible to perform on simple digital signals like those used in 433 MHz ISM band devices. This can be used for example to control wireless home automation devices like alarms and switches. RealVNC can be useful in this context as it allows you to remotely control your Raspberry Pi while it's performing these tasks. This means you can set up your SDR hacking station in one location and control it from another, which can be particularly useful if you need to place your Raspberry Pi and SDR in a specific location for optimal signal reception.
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 hav 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
Virtual Radar Server and Dump1090 are two software tools that are often used together in the context of ADS-B decoding. Dump1090 is a command-line based ADS-B decoder for RTL-SDR (Realtek Softw Defined Radio) devices. It is specifically designed for RTLSDR devices and is considered by many to be the best ADS-B decoder available. On the other hand, Virtual Radar Server is a software that takes the data from Dump1090, decodes it and then presents it on a virtual radar screen or on a map. The combination of these two tools allows users to receive and decode ADS-B transmissions from aircraft and plot them on a map. This can be particularly useful for tracking aircraft movements in real-time.
Automatic Dependent Surveillance–Broadcast, or ADS-B, is a technology that enhances surveillance of aircraft in flight. In this system, an aircraft's onboard equipment determines its location via satellite navigation and periodically broadcasts this information. This allows the aircraft to be tracked without the need for an interrogation signal from the ground, making it a viable replacement for secondary surveillance radar. Furthermore, the ADS-B information can be shared directly between aircraft, providing situational awareness and enabling self-separation. This technology is automatic, requiring no pilot input, and dependent on data from the aircraft's navigation system.
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)
To receive and decode SSTV transmissions from the ISS, you will need an RTL-SDR dongle, a simple antenna, and a software like MMSSTV. Tune in to 145.800 MHz on your RTL-SDR dongle and open MM Enable "Auto slant" and "Auto resync" under Options->Setup MMSTV->RX for the best results. Once th ISS is transmitting SSTV images, MMSSTV will automatically decode them and display them on the screen.

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.
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 :)
Don’t miss out on my upcoming articles! on Medium for more insightful content. Clap and share this article to spread the knowledge among fellow bug bounty hunters and cybersecurity enthusiasts.
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:
GSM (Global System for Mobile Communications) is a digital mobile network that is widely used around the world. It operates across four different frequency bands: 850 MHz, 900 MHz, 1800 MHz, and 1900 MHz. GSM uses a combination of Frequency Division Multiple Access (FDMA) and Time Division Multiple Access (TDMA) to efficiently utilize these frequency bands.
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).
Network and Switching Subsystem (NSS): NSS serves as the core network, overseeing call and mobility management functions for mobile devices. It encompasses components such as the Visitor Location Register (VLR), Home Location Register (HLR), and Authentication Center (AUC).
Operating Subsystem (OSS): OSS functions as a monitoring and control system for network operators. The Operation and Maintenance Center (OMC) is a crucial part of OSS, ensuring cost-effective support for all GSM-related maintenance services.
Press enter or click to view image in full size
Components of GSM Subsystems:
Mobile Station (MS): MS, short for Mobile System, comprises user equipment and a Subscriber Identity Module (SIM). These mobile stations connect to towers through Transceivers (TRX), facilitating both

transmission and reception of signals.
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.
Tools for GSM Sniffing
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
Kalibrate-rtl, or Kal, can scan for GSM base stations in a given frequency band . To install it on Debian-based systems, use the following command:
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
25 06 21 00 05 f4 f8 68 03 26 23 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
Set the frequency for grgsm_livemon,
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?
RTL-SDR is based on the Realtek RTL2832U chipset and is widely available for purchase at a low cost of around $20-$30 USD. It can receive live radio signals in a wide frequency range, typically from 500 kHz up to 1.75 GHz.
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 dong 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 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

Don't miss out on my upcoming articles! Follow me on Medium for more insightful content. Clap and share this article to spread the knowledge among fellow bug bounty hunters and cybersecurity enthusiasts.

If you have any further questions or would like to connect, feel free to reach out to me

My LinkedIn handle:

Hacking

Cybersecurity

Sdr