Python Radio 18: DominoEX Mode
Simon Quellen Field
Simon Quellen Field
Follow
10 min read
·
Sep 5, 2024
Listen
Share
More
Perfect for weak signal communication.
Press enter or click to view image in full size
Image by the author
Another mode that is great for NVIS and weak signal work on the HF bands is DominoEX.
Like FSQ, it is a conversational mode that is easy to tune, tolerates frequency drift, has low bandwidth but high typing speed, works well for NVIS conditions, and is robust enough to fading and interference that it normally needs no forward error correction.
DominoEX (often just called Domino, since the previous versions such as DominoF are no longer in use) has 6 baud rates. The lowest, DominoEX4, runs at 3.90625 baud, uses 173 hertz of bandwidth, ar manages as much as 25 words per minute. The other rates are DominoEX5, 8, 11, and 22 (the actual baud rates are 5.3833, 7.8125, 10.766, 15.625, and 21.533). The words per minute are respectively 31, 50, 70, 100, and 140.
There is also a DominoEX Micro mode that uses 2 baud, DominoEX44, and DominEX88 that were not original specification.
Like FSQ, DominoEX uses Incremental Frequency Keying, where symbols are defined by the differenc between tones, rather than the tones themselves. This makes tuning easy (you can be 200 hertz off frequency and still decode the message) and allows it to tolerate frequency drift.
There are 18 tones, and the varicode tables are grouped into sets of three tones. The dominoex_varicode.py module looks like this:

```
Dominoex_varicode = [
# Primary alphabet
( 1,15, 9), ( 1,15,10), ( 1,15,11), ( 1,15,12), ( 1,15,13), ( 1,15,14), ( 1,15,15), ( 2, 8, 8),
( 2,12, 0), ( 2, 8, 9), ( 2, 8,10), ( 2, 8,11), ( 2, 8,12), ( 2,13, 0), ( 2, 8,13), ( 2, 8,14),
( 2, 8,15), ( 2, 9, 8), ( 2, 9, 9), ( 2, 9,10), ( 2, 9,11), ( 2, 9,12), ( 2, 9,13), ( 2, 9,14),
( 2, 9,15), ( 2,10, 8), ( 2,10, 9), ( 2,10,10), ( 2,10,11), ( 2,10,12), ( 2,10,13), ( 2,10,14),
( 0, 0, 0), ( 7,11, 0), ( 0, 8,14), ( 0,10,11), ( 0, 9,10), ( 0, 9, 9), ( 0, 8,15), ( 7,10, 0),
( 0, 8,12), ( 0, 8,11), ( 0, 9,13), ( 0, 8, 8), ( 2,11, 0), ( 7,14, 0), ( 7,13, 0), ( 0, 8, 9),
( 3,15, 0), ( 4,10, 0), ( 4,15, 0), ( 5, 9, 0), ( 6, 8, 0), ( 5,12, 0), ( 5,14, 0), ( 6,12, 0),
( 6,11, 0), ( 6,14, 0), ( 0, 8,10), ( 0, 8,13), ( 0,10, 8), ( 7,15, 0), ( 0, 9,15), ( 7,12, 0),
( 0, 9, 8), ( 3, 9, 0), ( 4,14, 0), ( 3,12, 0), ( 3,14, 0), ( 3, 8, 0), ( 4,12, 0), ( 5, 8, 0),
( 5,10, 0), ( 3,10, 0), ( 7, 8, 0), ( 6,10, 0), ( 4,11, 0), ( 4, 8, 0), ( 4,13, 0), ( 3,11, 0),
( 4, 9, 0), ( 6,15, 0), ( 3,13, 0), ( 2,15, 0), ( 2,14, 0), ( 5,11, 0), ( 6,13, 0), ( 5,13, 0),
( 5,15, 0), ( 6, 9, 0), ( 7, 9, 0), ( 0,10,14), ( 0,10, 9), ( 0,10,15), ( 0,10,10), ( 0, 9,12),
( 0, 9,11), ( 4, 0, 0), ( 1,11, 0), ( 0,12, 0), ( 0,11, 0), ( 1, 0, 0), ( 0,15, 0), ( 1, 9, 0),
( 0,10, 0), ( 5, 0, 0), ( 2,10, 0), ( 1,14, 0), ( 0, 9, 0), ( 0,14, 0), ( 6, 0, 0), ( 3, 0, 0),
( 1, 8, 0), ( 2, 8, 0), ( 7, 0, 0), ( 0, 8, 0), ( 2, 0, 0), ( 0,13, 0), ( 1,13, 0), ( 1,12, 0),
( 1,15, 0), ( 1,10, 0), ( 2, 9, 0), ( 0,10,12), ( 0, 9,14), ( 0,10,13), ( 0,11, 8), ( 2,10,15),
( 2,11, 8), ( 2,11, 9), ( 2,11,10), ( 2,11,11), ( 2,11,12), ( 2,11,13), ( 2,11,14), ( 2,11,15),
( 2,12, 8), ( 2,12, 9), ( 2,12,10), ( 2,12,11), ( 2,12,12), ( 2,12,13), ( 2,12,14), ( 2,12,15),
( 2,13, 8), ( 2,13, 9), ( 2,13,10), ( 2,13,11), ( 2,13,12), ( 2,13,13), ( 2,13,14), ( 2,13,15),
```

```
( 2,14, 8), ( 2,14, 9), ( 2,14,10), ( 2,14,11), ( 2,14,12), ( 2,14,13), ( 2,14,14), ( 2,14,15),
( 0,11, 9), ( 0,11,10), ( 0,11,11), ( 0,11,12), ( 0,11,13), ( 0,11,14), ( 0,11,15), ( 0,12, 8),
( 0,12, 9), ( 0,12,10), ( 0,12,11), ( 0,12,12), ( 0,12,13), ( 0,12,14), ( 0,12,15), ( 0,13, 8),
( 0,13, 9), ( 0,13,10), ( 0,13,11), ( 0,13,12), ( 0,13,13), ( 0,13,14), ( 0,13,15), ( 0,14, 8),
( 0,14, 9), ( 0,14,10), ( 0,14,11), ( 0,14,12), ( 0,14,13), ( 0,14,14), ( 0,14,15), ( 0,15, 8),
( 0,15, 9), ( 0,15,10), ( 0,15,11), ( 0,15,12), ( 0,15,13), ( 0,15,14), ( 0,15,15), ( 1, 8, 8),
( 1, 8, 9), ( 1, 8,10), ( 1, 8,11), ( 1, 8,12), ( 1, 8,13), ( 1, 8,14), ( 1, 8,15), ( 1, 9, 8),
( 1, 9, 9), ( 1, 9,10), ( 1, 9,11), ( 1, 9,12), ( 1, 9,13), ( 1, 9,14), ( 1, 9,15), ( 1,10, 8),
( 1,10, 9), ( 1,10,10), ( 1,10,11), ( 1,10,12), ( 1,10,13), ( 1,10,14), ( 1,10,15), ( 1,11, 8),
( 1,11, 9), ( 1,11,10), ( 1,11,11), ( 1,11,12), ( 1,11,13), ( 1,11,14), ( 1,11,15), ( 1,12, 8),
( 1,12, 9), ( 1,12,10), ( 1,12,11), ( 1,12,12), ( 1,12,13), ( 1,12,14), ( 1,12,15), ( 1,13, 8),
( 1,13, 9), ( 1,13,10), ( 1,13,11), ( 1,13,12), ( 1,13,13), ( 1,13,14), ( 1,13,15), ( 1,14, 8),
( 1,14, 9), ( 1,14,10), ( 1,14,11), ( 1,14,12), ( 1,14,13), ( 1,14,14), ( 1,14,15), ( 1,15, 8),
# Secondary alphabet
( 6,15, 9), ( 6,15,10), ( 6,15,11), ( 6,15,12), ( 6,15,13), ( 6,15,14), ( 6,15,15), ( 7, 8, 8),
( 4,10,12), ( 7, 8, 9), ( 7, 8,10), ( 7, 8,11), ( 7, 8,12), ( 4,10,13), ( 7, 8,13), ( 7, 8,14),
( 7, 8,15), ( 7, 9, 8), ( 7, 9, 9), ( 7, 9,10), ( 7, 9,11), ( 7, 9,12), ( 7, 9,13), ( 7, 9,14),
( 7, 9,15), ( 7,10, 8), ( 7,10, 9), ( 7,10,10), ( 7,10,11), ( 7,10,12), ( 7,10,13), ( 7,10,14),
( 3, 8, 8), ( 4,15,11), ( 5, 8,14), ( 5,10,11), ( 5, 9,10), ( 5, 9, 9), ( 5, 8,15), ( 4,15,10),
( 5, 8,12), ( 5, 8,11), ( 5, 9,13), ( 5, 8, 8), ( 4,10,11), ( 4,15,14), ( 4,15,13), ( 5, 8, 9),
( 4,11,15), ( 4,12,10), ( 4,12,15), ( 4,13, 9), ( 4,14, 8), ( 4,13,12), ( 4,13,14), ( 4,14,12),
( 4,14,11), ( 4,14,14), ( 5, 8,10), ( 5, 8,13), ( 5,10, 8), ( 4,15,15), ( 5, 9,15), ( 4,15,12),
( 5, 9, 8), ( 4,11, 9), ( 4,12,14), ( 4,11,12), ( 4,11,14), ( 4,11, 8), ( 4,12,12), ( 4,13, 8),
( 4,13,10), ( 4,11,10), ( 4,15, 8), ( 4,14,10), ( 4,12,11), ( 4,12, 8), ( 4,12,13), ( 4,11,11),
( 4,12, 9), ( 4,14,15), ( 4,11,13), ( 4,10,15), ( 4,10,14), ( 4,13,11), ( 4,14,13), ( 4,13,13),
( 4,13,15), ( 4,14, 9), ( 4,15, 9), ( 5,10,14), ( 5,10, 9), ( 5,10,15), ( 5,10,10), ( 5, 9,12),
( 5, 9,11), ( 3, 8,12), ( 4, 9,11), ( 4, 8,12), ( 4, 8,11), ( 3, 8, 9), ( 4, 8,15), ( 4, 9, 9),
( 4, 8,10), ( 3, 8,13), ( 4,10,10), ( 4, 9,14), ( 4, 8, 9), ( 4, 8,14), ( 3, 8,14), ( 3, 8,11),
( 4, 9, 8), ( 4,10, 8), ( 3, 8,15), ( 4, 8, 8), ( 3, 8,10), ( 4, 8,13), ( 4, 9,13), ( 4, 9,12),
( 4, 9,15), ( 4, 9,10), ( 4,10, 9), ( 5,10,12), ( 5, 9,14), ( 5,10,12), ( 5,11, 8), ( 7,10,15),
( 7,11, 8), ( 7,11, 9), ( 7,11,10), ( 7,11,11), ( 7,11,12), ( 7,11,13), ( 7,11,14), ( 7,11,15),
( 7,12, 8), ( 7,12, 9), ( 7,12,10), ( 7,12,11), ( 7,12,12), ( 7,12,13), ( 7,12,14), ( 7,12,15),
( 7,13, 8), ( 7,13, 9), ( 7,13,10), ( 7,13,11), ( 7,13,12), ( 7,13,13), ( 7,13,14), ( 7,13,15),
( 7,14, 8), ( 7,14, 9), ( 7,14,10), ( 7,14,11), ( 7,14,12), ( 7,14,13), ( 7,14,14), ( 7,14,15),
( 5,11, 9), ( 5,11,10), ( 5,11,11), ( 5,11,12), ( 5,11,13), ( 5,11,14), ( 5,11,15), ( 5,12, 8),
( 5,12, 9), ( 5,12,10), ( 5,12,11), ( 5,12,12), ( 5,12,13), ( 5,12,14), ( 5,12,15), ( 5,13, 8),
( 5,13, 9), ( 5,13,10), ( 5,13,11), ( 5,13,12), ( 5,13,13), ( 5,13,14), ( 5,13,15), ( 5,14, 8),
( 5,14, 9), ( 5,14,10), ( 5,14,11), ( 5,14,12), ( 5,14,13), ( 5,14,14), ( 5,14,15), ( 5,15, 8),
( 5,15, 9), ( 5,15,10), ( 5,15,11), ( 5,15,12), ( 5,15,13), ( 5,15,14), ( 5,15,15), ( 6, 8, 8),
( 6, 8, 9), ( 6, 8,10), ( 6, 8,11), ( 6, 8,12), ( 6, 8,13), ( 6, 8,14), ( 6, 8,15), ( 6, 9, 8),
( 6, 9, 9), ( 6, 9,10), ( 6, 9,11), ( 6, 9,12), ( 6, 9,13), ( 6, 9,14), ( 6, 9,15), ( 6,10, 8),
( 6,10, 9), ( 6,10,10), ( 6,10,11), ( 6,10,12), ( 6,10,13), ( 6,10,14), ( 6,10,15), ( 6,11, 8),
( 6,11, 9), ( 6,11,10), ( 6,11,11), ( 6,11,12), ( 6,11,13), ( 6,11,14), ( 6,11,15), ( 6,12, 8),
( 6,12, 9), ( 6,12,10), ( 6,12,11), ( 6,12,12), ( 6,12,13), ( 6,12,14), ( 6,12,15), ( 6,13, 8),
( 6,13, 9), ( 6,13,10), ( 6,13,11), ( 6,13,12), ( 6,13,13), ( 6,13,14), ( 6,13,15), ( 6,14, 8),
( 6,14, 9), ( 6,14,10), ( 6,14,11), ( 6,14,12), ( 6,14,13), ( 6,14,14), ( 6,14,15), ( 6,15, 8),
]
```

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

```
From dominoex import DOMINOEX
From radio import Radio
```

```
From time import sleep_ms, sleep
Class DominoEXConfig:
Def __init__(self, baud, frq, call, location):
Self.dds = Radio()
Self.is_beacon = False
Self.beacon_interval = 30.0
Self.message = ‘’
Self.usb_offset = 1350.0
Self.num_tones = 18
Self.incremental_tone = 0.0
Self.all_done = False
Self.r = DOMINOEX(self.send_tone, self.report_all_done)
Self.set_baud(baud)
Self.frequency = frq
Self.call = call
Self.location = location
Self.r.set_frequency(frq)
Self.r.set_call(call)
Self.r.set_location(location)
Def get_radio(self):
Return self.dss
Def send_code(self):
Self.dds.send()
Def send_tone(self, tone):
Self.incremental_tone = (self.incremental_tone + float(tone) + 2) % self.num_tones
Self.f = int(int(self.frequency) + self.usb_offset + (self.incremental_tone + 0.5) *
self.tone_spacing – self.bandwidth / 2.0)
Self.dds.set_freq(0, self.f)
Self.dds.send()
Def report_all_done(self):
Print()
Print(“All done!”)
Self.all_done = True
If self.is_beacon:
Self.r.stop()           # stop sending bits
# self.dds.off()
Sleep(float(self.beacon_interval))
Self.dds.on()
Self.r.send_code()       # Repeat for a beacon
Else:
Self.r.stop()           # stop sending bits
# self.dds.off()
Def set_baud(self, b):
Self.baud = b
If self.baud == 2:
Self.spaced = 1
Self.sample_rate = 8000.0
Self.symbol_length = 4000.0
Elif self.baud == 4:
Self.spaced = 2
```

```
Self.sample_rate = 8000.0
Self.symbol_length = 2048.0
Elif self.baud == 5:
Self.spaced = 2
Self.sample_rate = 11025.0
Self.symbol_length = 2048.0
Elif self.baud == 8:
Self.spaced = 2
Self.sample_rate = 8000.0
Self.symbol_length = 1024.0
Elif self.baud == 11:
Self.spaced = 1
Self.sample_rate = 11025.0
Self.symbol_length = 1024.0
Elif self.baud == 16:
Self.spaced = 1
Self.sample_rate = 8000.0
Self.symbol_length = 512.0
Elif self.baud == 22:
Self.spaced = 1
Self.sample_rate = 11025.0
Self.symbol_length = 512.0
Elif self.baud == 44:
Self.spaced = 2
Self.sample_rate = 11025.0
Self.symbol_length = 256.0
Elif self.baud == 88:
Self.spaced = 1
Self.sample_rate = 11025.0
Self.symbol_length = 128.0
Self.r.set_baud(self.sample_rate / self.symbol_length)
Self.r.set_bit_length(1000 / (self.sample_rate / self.symbol_length))
Self.tone_spacing = self.sample_rate * self.spaced / self.symbol_length
Self.bandwidth = self.num_tones * self.tone_spacing
Def set_message(self, msg):
Self.r.set_message(chr(0) + “\r” + msg + “\r”)
Self.dds.on()
Self.r.send_code()
Self.all_done = False
Print(”Frequency:”, self.frequency)
Print(”Baud:”, self.baud)
Print(”Beacon?:”, self.is_beacon)
Print(”Message:”, self.r.message)
Print()
Print(”Bandwidth:”, self.bandwidth)
Print(”Tone spacing:”, self.tone_spacing)
Print(”Symbol length:”, self.symbol_length)
Print(”Bit length:”, 1000 / (self.sample_rate / self.symbol_length))
Print(”Baud:”, self.sample_rate / self.symbol_length)
Def set_beacon(self, onoff, interval):
```

```
Self.is_beacon = onoff
Self.beacon_interval = interval
```

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

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

```
From machine import Timer
From dominoex_varicode import dominoex_varicode
Class DOMINOEX:
Def __init__(self, send_tone, report_message_end=None):
Self.send_tone = send_tone
Self.report_message_end = report_message_end
# self.set_baud(10.766)     # DOMINOEX 11
Self.set_baud(2)     # DOMINOEX MICRO
Self.frequency = “7104000”
Self.call = “N0CALL”
Self.location = “CM87xe”
Self.message = “{} {}   “
Self.bit_length = int(1000 / float(self.baud))
Self.timer = Timer()
Def set_call(self, call):
Self.call = call
Def set_baud(self, baud):
Self.baud = float(baud)
Def set_bit_length(self, len):
Self.bit_length = int(len)
Def set_frequency(self, frequency):
Self.frequency = frequency
Def set_location(self, location):
Self.location = location
Def set_message(self, message):
Self.message = message.format(self.call, self.location)
Def bit(self):
For letter in self.message:
Code = dominoex_varicode[ord(letter)]
Count = 0
For tone in code:
If tone & 0x8 or count == 0:
Yield tone
Count += 1
Self.report_message_end()
Def stop(self):
Self.timer.deinit()
Def send_code(self):
Self.gen = self.bit()
Self.timer.init(period=self.bit_length, mode=Timer.PERIODIC, callback=self.bit_finished)
Def send_bit(self, unused):
Try:
Tone = next(self.gen)
Except StopIteration as tone:
Return self.report_message_end()
```

Self.send_tone(tone)
Def bit_finished(self, unused):
Self.send_bit(True)
Our main.py module looks like this:
From dominoex_config import DominoEXConfig
From time import sleep
Def main():
Dex = DominoEXConfig(4, 7040000, “AB6NY”, “CM87xe”)
While True:
Dex.set_message(“{} Testing from {} using a Raspberry Pi Pico RP2040”)
Dex.send_code()
While dex.all_done == False:
Sleep(5)
Main()
We set up the baud rate (4 in this case), and the frequency, call, and location. Then we set up the message, and start sending.
The radio.py and SI5351.py modules are the same as before.
I did notice when using a baud rate of 11 (and even 8) that the first few characters in the message would often be garbled due to FLDIGI trying to be too smart about adjusting signal levels. Since the call sign was what was getting garbled (not a good thing) I added a few disposable characters in the beginning:
Dex.set_message(”   {} Testing from {} using a Raspberry Pi Pico RP2040”)
The result looked like this:
Press enter or click to view image in full size
Image by the author
Interestingly, there was less of this at the faster baud rate of 22 (140 words per minute). A few errors, but for the most part, completely readable:
Press enter or click to view image in full size
Image by the author
At 44 baud, things got progressively worse, but it was so fast that it was easy to get the whole message by reading several lines:
Press enter or click to view image in full size
Image by the author
To computer users used to a baud being a bit instead of a symbol, 44 baud sounds slow. However, since each symbol is a character in DominoEX, this is 44 characters per second or 440 bits per second. Still pathetic compared to Wi-Fi speeds, but this is five dollars’ worth of hardware capable of bouncing over the horizon. And as a chat mode, 280 words per minute is faster than I can type anyway.
Dominoex
Amateur Radio
Radio Hackers: Electromagnetic Eavesdropping & Harmonics
Investigator515
Investigator515
Follow
6 min read
·
Jan 13, 2025
Listen
Share
More

EMR eavesdropping uses receiving techniques to exploit secure information.

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

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

Known as Electromagnetic Eavesdropping, this technique would use the stray emissions broadcast by electronic devices to help compromise secure data. While today’s article will cover the concept of the field rather than breaking down a specific attack type, there’s still plenty for the RF enthusiast to take away from today’s article. Let’s take a look!

EMR & Cybersecurity

While it could be a controversial topic in the earlier days, there’s no debating that in today’s world, there is a distinct overlap between cyber / hardware security and the radio spectrum. With this involving everything from NFC cards to Bluetooth, Wi-Fi & IOT devices, the overlap has become even more pronounced, with many opportunities for research and analysis.

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

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

WTF Is a Harmonic?!

If you’re an amateur radio operator or military communications technician, you probably won’t need a refresher on the topic. For those who aren’t part of that community though, the concept of harmonic frequencies is that for every transmission on a fundamental frequency, we’ll also see “harmonic” frequencies at set points across the radio spectrum. Let’s get specific and check out what that means, using a frequency in the VHF band. 121.5mhz is well known to those in maritime & aviation as it used to be the emergency frequency before the digital shift.

So if our fundamental frequency is 121.5mhz, we’ll see our second harmonic at 243 MHz (2 × 121.5 MHz) and our third harmonic should appear at 364.5 MHz (3 × 121.5 MHz). While it sounds pretty complex for those still learning about radio, we can work these out by using some simple math.

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

The flip side of this is that while properly built transmitting devices are usually well filtered, causing no issue with harmonic frequencies, many cheap transmitter and household consumer devices use little in the way of protection or shielding.

While that can typically cause some issues with fundamental frequencies (Baofeng radios anybody?) what it also means is that much of the natural emissions produced by things like the oscillators remain unshielded as well, and in many circumstances, it’s these stray emissions that are able to be exploited for espionage purposes.

Press enter or click to view image in full size

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

What Kind Of Exploits?

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

If you aren’t familiar with some of these techniques you’d be forgiven for thinking you were reading from the set of a James Bond movie, as TEMPEST research covered light, audio, vibrations and pretty much everything in between. In its earliest stages, the research was complex and labour-intensive, with a significant entry barrier in terms of the technical skills required. While that hasn't changed in the modern world, one thing that has changed is the technology that's available to assist with this type of work. When paired with modern strategies like machine learning, there's still plenty to be gained by using the radio spectrum for research purposes.

Due to the complexity of these types of cyberattacks though there's little in the way of validated information about them occurring in the wild. Given the earliest research occurred in the 1980s, it's reasonable to assume that while some of these attack vectors nights have been used, there's little incentive for both victim and attacker to provide any form of validation.

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

How Can I Learn?

While these might be state-based attacks that are out of the realm of the hobbyist-level researcher, the reality is that for those who are interested, it's still a pretty viable field for independent research. Technology is responsible for changing much in the world of technology over the past 20 years, however, the importance of the radio spectrum in today's world is probably even more relevant than it was previously.

We can see this in the explosion of IOT devices over the past few years, many of which use some form of RF protocol to communicate. This could be purely for data transmission or it could entail some form of remote update or administrator access, similar to the Over-The-Air updates that we see in modern motor vehicles. It's fair to say that while for many, the radio spectrum is out of sight and out of mind, it's still an important domain to understand, particularly for military purposes.

Press enter or click to view image in full size

So for learning purposes, there are a few things beginners can look at to deepen their understanding of this topic. Firstly, understanding the relationship between frequencies and wavelength as well as identifying harmonic frequencies is a great place to start.

Next up understanding how different signal types propagate in different mediums is also a useful skill to understand. Grasping this will help you understand what is good for high-speed data or transmitting underwater to submarines or how to overcome propagation issues in built-up or marginal areas.

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

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

✉■ Want more content like this? Sign up for email updates
Surveillance
Radio
Information Security
Technology
Radio Hackers: Jamming & Spoofing Fundamentals
Follow
7 min read
·
Jul 9, 2024
Listen
Share
More
Understanding Spoofing and Jamming is an essential part of your offensive toolkit.
If you aren’t a medium member, you can read with no paywall via substack
We strive to provide informative articles, however, it is important for users to ensure their research is both ethical and responsible. Additionally, it is your responsibility to ensure you’re compliant with all applicable laws and regulations for your region. The information provided in this article is intended for educational purposes only.
During our Radio Hackers journey, we’ve started to lay down fundamental techniques to assist in developing a framework that will help leverage some of your new skills and information. We’ve looked at the importance of antennas as well as exploring some of the different types of signals that you might see while experimenting with the radio spectrum.
However, in Signals Intelligence, we’re going to find ourselves dealing with large amounts of data, and in the world of intelligence, we don’t automatically trust everything we discover without attempting to validate it first.
So, during the process of conducting Signals Intelligence (SIGINT), this means that determining the authenticity of a signal is an important part of the assessment process. Today, we’ll be exploring some of the fundamentals behind signal spoofing and jamming and looking at why we need to know ho to identify malicious signals. There’s also a real-life example of a spoofed signal at work.
What Is Spoofing And Jamming?
Despite both being forms of electronic attack there’s a distinct difference in the way we observe them in the wild. But, like many other radio hacking scenarios, we can often see links to traditional cyber attacks when we start examining them a little more closely. Spoofing, for instance, could be considered to be similar to an evil twin attack, where a rouge signal is impersonating a legitimate one for malicious purposes. Whereas a jamming attack is similar to a Denial of Service as the jammer's signal will deny usage of parts of the radio spectrum to anyone who is in range indiscriminately. While electronic warfare (EW) has been prevalent for military purposes since the days of World War two, the prevalence of Wi-Fi, the Internet and consumer devices means that the overlap between EW and the civilian world now is much more noticeable than it once was.
It’s also worth mentioning that these attacks can often be personalized or discriminate, where they are targeting a specific person or device of interest. We’ll mostly see that at a nation-state level. Or, they could be indiscriminate, affecting everyone within the transmitting jammers range.
The AN/ALQ-99 jamming pod on an electronic warfare aircraft is a good example of something that car work indiscriminately, providing general area effects.
Press enter or click to view image in full size
How Does It Work
Typically, a jamming attack is applied over a large chunk of the radio spectrum at high power, providing the indiscriminate area-based effects we discussed earlier. However, these attacks can

also be targeted at specific frequencies. This allows the jamming transmitter to focus its power on a particular part of the spectrum leading to a much stronger jamming effect due to higher output power from the jammer targeting that wavelength.

At its most basic level though, a jamming attack is effectively using one transmitter to overpower the other, while a spoofing attack relies on the transmission of malicious signals to achieve its end goal. And while we have used military examples to assist in explaining how things might work, it’s important to realise that we can also see similar attacks in the real world.

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

Press enter or click to view image in full size

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

Does This Even Happen?

If you aren’t studying cyber or infosec you might be reading this and thinking it’s not relevant to the civilian world and there's no way things like this can happen in our modern world. And while that might be true if you’re living in the southern hemisphere, the northern hemisphere people are regularly exposed to this type of thing and the reality is that often, it barely even makes a headline. So, to take a look at some real-world examples of this occurring we’re going to look at some infrastructure that's regularly used by many people the world over. The GPS Satellite constellation on 1575.42 and 1227.60mhz.

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

Press enter or click to view image in full size

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

It’s fair to say that most people understand that there is an active war in Europe that's been going on for many years. And while the frontlines may be a long way away for some readers, the reality is that Electronic Warfare strategies know no borders and some of this active EW has spilled out to cover parts of Europe outside of the conflict zone.

More importantly, we’ve seen both types of attacks while this has been occurring as both Spoofing and Jamming attacks have been reported on multiple occasions since the conflict escalated. To explore this in a bit more detail, we’re going to use a simple but extremely useful website that attempts to track ongoing and active attacks on civilian GPS.

Press enter or click to view image in full size

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

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

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

Press enter or click to view image in full size

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

And while you might not ever hold a pilot's license or use the GPS constellation for much more than personal navigation, it’s still plausible to see some other small-scale attacks in the real world regardless. The flipper zero made headlines recently for its ability to repeatedly spam BTLE signals (spoofing), while a simple de-auth packet transmitted repeatedly over Wi-Fi could interfere with

legitimate devices on a network (jamming).
Press enter or click to view image in full size
The Flipper Zero gained notoriety for its ability to transmit BTLE spam. Source: Github.com
In Closing.
If you'd like some more resources on the topic of spoofing and jamming attacks in the real world to help increase your knowledge level, and background then here are a few links to get you started.
While this is the first piece we've written that covers spoofing and jamming theory, we'll be exploring this a lot more in future Radio Hackers articles. We'll also be reviewing more simulated data in a controlled, lab environment using low-powered software-defined radio systems, to help further your understanding of how to detect and defend against these types of attacks.
If you haven't got a transmit-capable SDR to go with your Radio Hackers tutorials, then now might just be the time to splash out and buy one.
Medium has recently made some algorithm changes to improve the discoverability of articles like this one. These changes are designed to ensure that high-quality content reaches a wider audience, and your engagement plays a crucial role in making that happen.
If you found this article insightful, informative, or entertaining, we kindly encourage you to show your support. Clapping for this article not only lets the author know that their work is appreciated but also helps boost its visibility to others who might benefit from it.
■ Enjoyed this article? Join the community! ■
■ Join our OSINT channel for exclusive updates or
■ Follow us on
■ Articles we think you'll like:
Software Defined Radio &
OSINT Investigators Guide to
✉■ Want more content like this?
Hacking
Radio
Software Defined Radio
Learning
Python Radio 16: MFSK Mode
Follow
20 min read
·
Sep 3, 2024
Listen
Share
More
Bigger and faster…
Press enter or click to view image in full size
Photo by the author
In the last article, we hinted at the Si5351 module that will allow us to send on any frequency we like, with a resolution of less than one Hertz.
In this article, we get to use it.
We will connect the Si5351 module (available for about a dollar on AliExpress, or $8 on Amazon.com) to the RP2040 using four wires.
The module uses the I2C interface, so all it needs are five-volt power, ground, a clock signal, and a data signal. The ESP32 and ESP8266 can also use this module, as they both support the I2C interface.
Like those microprocessors, the SI5351 also produces square waves by dividing a clock by integers, but it has much better resolution. Below 18 megahertz, it will never be more than half a hertz off

of the target frequency. Even at 150 MHz the error is only 4 hertz, and 18% of the frequencies in that range don’t miss at all. The module has a nominal range of 8 kilohertz to 160 megahertz, but it can actually reach the 1.25-meter band up around 220 megahertz.

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

As an added bonus, the SI5351 puts out 50 milliwatts of power, instead of the 3.8 milliwatts we got out of our raw RP2040. You can claim the 1,000 miles per watt award by having a conversation with someone 50 miles away, with no amplifier, powered by USB. QRP is anything 5 watts or less. QRPp is less than a watt, down to 100 milliwatts. Below that is called QRPpp, and our 50-milliwatt transmitter is in that class. You can, of course, add an amplifier if you feel the need for more power.

To use the module, we need a driver. Device drivers are difficult to understand without reading the entire specification sheet for the device, and so I won’t be going into any detail here. The following code is in the module SI5351.py:

```
from machine import I2C
import math
SI5351_REGISTER_0_DEVICE_STATUS                    = 0
SI5351_REGISTER_1_INTERRUPT_STATUS_STICKY          = 1
SI5351_REGISTER_2_INTERRUPT_STATUS_MASK            = 2
SI5351_REGISTER_3_OUTPUT_ENABLE_CONTROL            = 3
SI5351_REGISTER_9_OEB_PIN_ENABLE_CONTROL           = 9
SI5351_REGISTER_15_PLL_INPUT_SOURCE                = 15
SI5351_REGISTER_16_CLK0_CONTROL                    = 16
SI5351_REGISTER_17_CLK1_CONTROL                    = 17
SI5351_REGISTER_18_CLK2_CONTROL                    = 18
SI5351_REGISTER_19_CLK3_CONTROL                    = 19
SI5351_REGISTER_20_CLK4_CONTROL                    = 20
SI5351_REGISTER_21_CLK5_CONTROL                    = 21
SI5351_REGISTER_22_CLK6_CONTROL                    = 22
SI5351_REGISTER_23_CLK7_CONTROL                    = 23
SI5351_REGISTER_24_CLK3_0_DISABLE_STATE            = 24
SI5351_REGISTER_25_CLK7_4_DISABLE_STATE            = 25
SI5351_REGISTER_42_MULTISYNTH0_PARAMETERS_1        = 42
SI5351_REGISTER_43_MULTISYNTH0_PARAMETERS_2        = 43
SI5351_REGISTER_44_MULTISYNTH0_PARAMETERS_3        = 44
SI5351_REGISTER_45_MULTISYNTH0_PARAMETERS_4        = 45
SI5351_REGISTER_46_MULTISYNTH0_PARAMETERS_5        = 46
SI5351_REGISTER_47_MULTISYNTH0_PARAMETERS_6        = 47
SI5351_REGISTER_48_MULTISYNTH0_PARAMETERS_7        = 48
SI5351_REGISTER_49_MULTISYNTH0_PARAMETERS_8        = 49
SI5351_REGISTER_50_MULTISYNTH1_PARAMETERS_1        = 50
SI5351_REGISTER_51_MULTISYNTH1_PARAMETERS_2        = 51
SI5351_REGISTER_52_MULTISYNTH1_PARAMETERS_3        = 52
SI5351_REGISTER_53_MULTISYNTH1_PARAMETERS_4        = 53
SI5351_REGISTER_54_MULTISYNTH1_PARAMETERS_5        = 54
SI5351_REGISTER_55_MULTISYNTH1_PARAMETERS_6        = 55
SI5351_REGISTER_56_MULTISYNTH1_PARAMETERS_7        = 56
SI5351_REGISTER_57_MULTISYNTH1_PARAMETERS_8        = 57
SI5351_REGISTER_58_MULTISYNTH2_PARAMETERS_1        = 58
SI5351_REGISTER_59_MULTISYNTH2_PARAMETERS_2        = 59
```

```
SI5351_REGISTER_60_MULTISYNTH2_PARAMETERS_3          = 60
SI5351_REGISTER_61_MULTISYNTH2_PARAMETERS_4          = 61
SI5351_REGISTER_62_MULTISYNTH2_PARAMETERS_5          = 62
SI5351_REGISTER_63_MULTISYNTH2_PARAMETERS_6          = 63
SI5351_REGISTER_64_MULTISYNTH2_PARAMETERS_7          = 64
SI5351_REGISTER_65_MULTISYNTH2_PARAMETERS_8          = 65
SI5351_REGISTER_66_MULTISYNTH3_PARAMETERS_1          = 66
SI5351_REGISTER_67_MULTISYNTH3_PARAMETERS_2          = 67
SI5351_REGISTER_68_MULTISYNTH3_PARAMETERS_3          = 68
SI5351_REGISTER_69_MULTISYNTH3_PARAMETERS_4          = 69
SI5351_REGISTER_70_MULTISYNTH3_PARAMETERS_5          = 70
SI5351_REGISTER_71_MULTISYNTH3_PARAMETERS_6          = 71
SI5351_REGISTER_72_MULTISYNTH3_PARAMETERS_7          = 72
SI5351_REGISTER_73_MULTISYNTH3_PARAMETERS_8          = 73
SI5351_REGISTER_74_MULTISYNTH4_PARAMETERS_1          = 74
SI5351_REGISTER_75_MULTISYNTH4_PARAMETERS_2          = 75
SI5351_REGISTER_76_MULTISYNTH4_PARAMETERS_3          = 76
SI5351_REGISTER_77_MULTISYNTH4_PARAMETERS_4          = 77
SI5351_REGISTER_78_MULTISYNTH4_PARAMETERS_5          = 78
SI5351_REGISTER_79_MULTISYNTH4_PARAMETERS_6          = 79
SI5351_REGISTER_80_MULTISYNTH4_PARAMETERS_7          = 80
SI5351_REGISTER_81_MULTISYNTH4_PARAMETERS_8          = 81
SI5351_REGISTER_82_MULTISYNTH5_PARAMETERS_1          = 82
SI5351_REGISTER_83_MULTISYNTH5_PARAMETERS_2          = 83
SI5351_REGISTER_84_MULTISYNTH5_PARAMETERS_3          = 84
SI5351_REGISTER_85_MULTISYNTH5_PARAMETERS_4          = 85
SI5351_REGISTER_86_MULTISYNTH5_PARAMETERS_5          = 86
SI5351_REGISTER_87_MULTISYNTH5_PARAMETERS_6          = 87
SI5351_REGISTER_88_MULTISYNTH5_PARAMETERS_7          = 88
SI5351_REGISTER_89_MULTISYNTH5_PARAMETERS_8          = 89
SI5351_REGISTER_90_MULTISYNTH6_PARAMETERS          = 90
SI5351_REGISTER_91_MULTISYNTH7_PARAMETERS          = 91
SI5351_REGISTER_092_CLOCK_6_7_OUTPUT_DIVIDER         = 92
SI5351_REGISTER_165_CLK0_INITIAL_PHASE_OFFSET        = 165
SI5351_REGISTER_166_CLK1_INITIAL_PHASE_OFFSET        = 166
SI5351_REGISTER_167_CLK2_INITIAL_PHASE_OFFSET        = 167
SI5351_REGISTER_168_CLK3_INITIAL_PHASE_OFFSET        = 168
SI5351_REGISTER_169_CLK4_INITIAL_PHASE_OFFSET        = 169
SI5351_REGISTER_170_CLK5_INITIAL_PHASE_OFFSET        = 170
SI5351_REGISTER_177_PLL_RESET                        = 177
SI5351_REGISTER_183_CRYSTAL_INTERNAL_LOAD_CAPACITANCE   = 183
SI5351_CRYSTAL_FREQ_25MHZ = 25000000
SI5351_CRYSTAL_FREQ_27MHZ = 27000000
SI5351_CRYSTAL_LOAD_6PF  = 1<<6
SI5351_CRYSTAL_LOAD_8PF  = 2<<6
SI5351_CRYSTAL_LOAD_10PF = 3<<6
si5351_15to92 =
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
SI5351_MULTISYNTH_DIV_4  = 4
SI5351_MULTISYNTH_DIV_6  = 6
```

```
SI5351_MULTISYNTH_DIV_8  = 8
class SI5351:
def __init__(self, i2c, address=0x60, crystalFreq=25000000):
self.i2c = i2c
self.address = address
self.initialized     = False
self.crystalFreq     = crystalFreq
self.crystalLoad     = SI5351_CRYSTAL_LOAD_10PF
self.crystalPPM      = 30
self.plla_configured = False
self.plla_freq       = 0
self.pllb_configured = False
self.pllb_freq       = 0
return
def write8(self, register, value):
ret = True
self.i2c.start()   # only available in SoftI2C
buffera = bytearray(1)
buffera[0] = value & 0xff
try:
self.i2c.writeto_mem(self.address, register, buffera)
except Exception as e:
print(”Exception”, e, “when writing to Si5351”)
print(”Address: “, self.address)
print(”Scan: “, self.i2c.scan())
ret = False
self.i2c.stop()   # only available in SoftI2C
return ret
def read8(self, register, value):
ret = True
self.i2c.start()   # only available in SoftI2C
buffera = bytearray(1)
try:
self.i2c.readfrom_mem_into(self.address, register, buffera)
except Exception as e:
print(”Exception”, e, “when writing to Si5351”)
print(”Address: “, self.address)
print(”Scan: “, self.i2c.scan())
ret = False
self.i2c.stop()   # only available in SoftI2C
return ret
def begin(self):
self.write8(SI5351_REGISTER_3_OUTPUT_ENABLE_CONTROL, 0xFF)
# Power down all output drivers */
self.write8(SI5351_REGISTER_16_CLK0_CONTROL, 0x80)
self.write8(SI5351_REGISTER_17_CLK1_CONTROL, 0x80)
self.write8(SI5351_REGISTER_18_CLK2_CONTROL, 0x80)
self.write8(SI5351_REGISTER_19_CLK3_CONTROL, 0x80)
self.write8(SI5351_REGISTER_20_CLK4_CONTROL, 0x80)
self.write8(SI5351_REGISTER_21_CLK5_CONTROL, 0x80)
```

```
self.write8(SI5351_REGISTER_22_CLK6_CONTROL, 0x80)
self.write8(SI5351_REGISTER_23_CLK7_CONTROL, 0x80)
#  Set the load capacitance for the XTAL */
self.write8(SI5351_REGISTER_183_CRYSTAL_INTERNAL_LOAD_CAPACITANCE,
self.crystalLoad)
# Set interrupt masks as required (see Register 2 description in AN619).
# By default, ClockBuilder Desktop sets this register to 0x18.
# Note that the least significant nibble must remain 0x8, but the most
# significant nibble may be modified to suit your needs.
# Reset the PLL config fields just in case we call init again
self.plla_configured = False
self.plla_freq = 0
self.pllb_configured = False
self.pllb_freq = 0
# All done!
self.initialized = True
return
def setClockBuilderData(self ):
i = 0
# Make sure we've called init first
assert self.initialized == True, "you have not initialized the object"
# Disable all outputs setting CLKx_DIS high
self.write8(SI5351_REGISTER_3_OUTPUT_ENABLE_CONTROL, 0xFF)
# Writes configuration data to device using the register map contents
# generated by ClockBuilder Desktop (registers 15-92 + 149-170)
for i, x in enumerate(range(15,93)):
#print(x,  si5351_15to92[i] )
self.write8(x, si5351_15to92[i] )
for i in range(149, 171):
self.write8(i, 0x00)
# Apply soft reset
self.write8(SI5351_REGISTER_177_PLL_RESET, 0xAC)
# Enabled desired outputs (see Register 3)
self.write8(SI5351_REGISTER_3_OUTPUT_ENABLE_CONTROL, 0x00)
return None
def setupPLL(self, mult, num, denom, pllsource = 'A', reset=True):
assert self.initialized == True, "you have not initialized the object"
assert ((mult > 14) and (mult < 91) ), "invalid mult parameter"
assert denom > 0, "denom must be > 0"
assert num <= 0xfffff, "invalid parameter num"
assert denom <= 0xfffff, "invalid parameter denom"
if num ==0:
P1 = 128*mult -512
P2 = num
P3 = denom
else:
P1 = 128*mult + math.floor( 128 * num/denom ) -512
P2 = 128*num - denom * math.floor( 128 * num/denom)
P3 = denom
if pllsource == 'A':
```

```
baseaddr = 26
else:
baseaddr = 34
P1 = int(P1)
self.write8( baseaddr,   (P3 & 0x0000FF00) >> 8)
self.write8( baseaddr+1, (P3 & 0x000000FF))
self.write8( baseaddr+2, (P1 & 0x00030000) >> 16)
self.write8( baseaddr+3, (P1 & 0x0000FF00) >> 8)
self.write8( baseaddr+4, (P1 & 0x000000FF))
self.write8( baseaddr+5, ((P3 & 0x000F0000) >> 12) | ((P2 & 0x000F0000) >> 16) )
self.write8( baseaddr+6, (P2 & 0x0000FF00) >> 8)
self.write8( baseaddr+7, (P2 & 0x000000FF))
if reset:
self.write8(SI5351_REGISTER_177_PLL_RESET, (1<<7) | (1<<5) )
if pllsource ==’A’:
fvco = self.crystalFreq*( mult + num/denom)
self.plla_configured = True
self.plla_freq = int(math.floor( fvco))
else:
fvco = self.crystalFreq*(mult + num/denom)
self.pllb_configured = True
self.pllb_freq = int(math.floor(fvco))
return None
def setupRdiv( self, output, div):
assert output in [0,1,2], “output value invalid”
assert div in [1,2,4,8,16,32,64,128], “div invalid”
divdict = {1: 0, 2: 1, 4: 2, 8: 3, 16: 4, 32: 5, 64: 6, 128: 7}
registers = [ 44, 52, 60]
Rreg = registers[output]
buf = bytearray( 1)
self.read8(Rreg, buf)
regval = buf[0] & 0x0F
divider = divdict[div]
divider &= 0x07
divider <<= 4
regval |= divider
self.write8(Rreg, regval)
return None
def setupMultisynth( self, output, div, num, denom, pllsource, power=3):
assert self.initialized  == True, “device not initialized”
assert output in [0,1,2], “output out of range”
assert div > 3, “div out of range”
assert denom >0, “denom out of range”
assert num <= 0xfffff, “num has a 20-bit limit”
assert denom <= 0xfffff, “denom as a 20-bit limit”
if pllsource==”A”:
assert self.plla_configured == True, “plla has not been configured”
else:
assert self.pllb_configured == True, ‘pllb has not been configured’
# Output Multisynth Divider Equations
```

```
# where: a = div, b = num and c = denom
#
#  P1 register is an 18-bit value using following formula:
#
# P1[17:0] = 128 * a + floor(128*(b/c)) - 512
#
# P2 register is a 20-bit value using the following formula:
#
#  P2[19:0] = 128 * b - c * floor(128*(b/c))
#
# P3 register is a 20-bit value using the following formula:
#
# P3[19:0] = c
if num==0:
# integer mode
P1 = int(128 * div - 512)
P2 = num
P3 = denom
else:
# Fractional mode */
P1 = int( 128 * div + math.floor(128 * (num/denom)) - 512 )
P2 = int( 128 * num - denom * math.floor(128 * (num/denom)))
P3 = denom
baseaddrs = [ 42, 50, 58]
baseaddr = baseaddrs[output]
self.write8( baseaddr,  (P3 & 0x0000FF00) >> 8)
self.write8( baseaddr+1, (P3 & 0x000000FF))
self.write8( baseaddr+2, (P1 & 0x00030000) >> 16) # ToDo: Add DIVBY4 (>150MHz) and R0 support
(<500kHz) later */
self.write8( baseaddr+3, (P1 & 0x0000FF00) >> 8)
self.write8( baseaddr+4, (P1 & 0x000000FF))
self.write8( baseaddr+5, ((P3 & 0x000F0000) >> 12) | ((P2 & 0x000F0000) >> 16) )
self.write8( baseaddr+6, (P2 & 0x0000FF00) >> 8)
self.write8( baseaddr+7, (P2 & 0x000000FF))
# Configure the clk control and enable the output
clkControlReg = 0x0F                    # 8mA drive strength, MS0 as CLK0 source, Clock not
inverted, powered up
if pllsource == ‘B’:
clkControlReg |= (1 << 5) # /* Uses PLLB */
if num == 0:
clkControlReg |= (1 << 6) #  Integer mode */
#
# Set power level bits
# 0: 2mA -8dB
# 1: 4mA -3dB
# 2: 6mA -1dB
# 3: 8mA -0dB (default)
#
# clkControlReg |= power & 3
if output == 0:
```

```
self.write8(SI5351_REGISTER_16_CLK0_CONTROL, clkControlReg)
if output == 1:
self.write8(SI5351_REGISTER_17_CLK1_CONTROL, clkControlReg)
if output == 2:
self.write8(SI5351_REGISTER_18_CLK2_CONTROL, clkControlReg)
def enableOutputs( self, enabled=True):
assert self.initialized == True, “Error Device not initialized”
if enabled:
ret = self.write8( SI5351_REGISTER_3_OUTPUT_ENABLE_CONTROL, 0x00)
else:
ret = self.write8( SI5351_REGISTER_3_OUTPUT_ENABLE_CONTROL, 0xff)
return ret
```

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

Here is the code:

```
import utime
class Radio:
def __init__(self):
from SI5351 import SI5351
from machine import Pin, SoftI2C
# self.i2c = SoftI2C(scl=Pin(22), sda=Pin(21), freq=400000)     # ESP32
self.i2c = SoftI2C(scl=Pin(1), sda=Pin(0), freq=400000)     # RP2040
# self.i2c = SoftI2C(scl=Pin(22), sda=Pin(21),   freq=800000)     # The ESP32 can do up to 5 MHz
best case
print( “I2C.scan():”, self.i2c.scan())
for x in range(5):
self.clockgen = SI5351(self.i2c, 0x60 + x)
status = 0
if self.clockgen.read8(0, status):
break
self.clockgen.begin()
self.clockgen.setClockBuilderData()
self.key_state = False
self.actual_freq_a = 0
self.actual_freq_b = 0
self.nominal_freq_a = 0
self.nominal_freq_b = 0
# self.last_time = utime.ticks_ms()
self.old_mult = 0
self.old_num = 0
self.old_denom = 0
self.old_src = 0
def gcd(self, x, y):
while(y):
x, y = y, x % y
return x
def send(self):
pass
```

```
def info(self):
print( “I2C.scan():”, self.i2c.scan())
def on(self):
if self.clockgen.enableOutputs(True):
self.key_state = True
def off(self):
if self.clockgen.enableOutputs(False):
self.key_state = False
def key_down(self):
if self.clockgen.enableOutputs(True):
self.key_state = True
def key_up(self):
if self.clockgen.enableOutputs(False):
self.key_state = False
def get_freq(self, which):
if which == 0:
return self.nominal_freq_a
else:
return self.nominal_freq_b
def set_freq(self, which, f):
f = float(f)
div = int(900000000.0 / f)              # Values under a megahertz need an extra divide step
r = 1
while div > 900:
r *= 2
div /= 2
if div % 2:                             # Make sure it is an even number
div -= 1
pllFreq = div * r * f
xtal_freq = 25000000                    # Our board uses a 25 MHz crystal
fmult = pllFreq / xtal_freq             # The full multiplier
mult = int(fmult)                       # The integer part of the multiplier
frac = fmult - mult
off = int(frac * xtal_freq)
divisor = self.gcd(off, xtal_freq)
num = int(off / divisor)
denom = int(xtal_freq / divisor)
if num > 0xFFFFF or denom > 0xFFFFF:
denom = 0xFFFFF                         # Use the maximum value for the denominator
num = int((pllFreq % xtal_freq) * denom / xtal_freq)
# Below 18 MHz, we will never be more than half a Hertz off
# Below 37.5 MHz, we will never be more than a Hertz off
# Below 75 MHz, we will never be more than two Hertz off
# Below 112.5 MHz, we will never be more than three Hertz off
# Below 150 MHz, we will never be more than four Hertz off
# Below 222 MHz, we will never be more than six Hertz off
# A little over 18% of the frequencies were right on the money
# This is better than the frequency stability of a temperature controlled crystal oscillator
# so any failure of accuracy here will be swamped by the variability in the oscillator
# Of course, if you have a nice OCXO crystal oscillator in an oven that has a parts-per-billion
```

```
accuracy
# then you might want to know that you will never be off by more than 4 Hz in the 200 Hz wide WSPR
window
# in the 2 meter band.
# All that assumes that we have double precision arithmetic. Micropython on the ESP8266 only has
single precision 32 bit floats.
# So we can think we are off by as much as 5 Hz in the 40-meter band, when the Si5351 is actually
much more accurate.
# On the ESP32, I have built special micropython firmware that supports double precision
artithmetic.
# If r is 1, we will never be less than our target frequency
if which == 0:
self.actual_freq_a = (mult * xtal_freq + xtal_freq * num / denom) / div * r
self.nominal_freq_a = f
src = "A"
else:
self.actual_freq_b = (mult * xtal_freq + xtal_freq * num / denom) / div * r
self.nominal_freq_b = f
src = "B"
# print("Mult is", mult, "Num is", num, "Denom is", denom, "Src is", src, "Div is", div)
reset = False
if mult != self.old_mult and num != self.old_num and denom != self.old_denom and src !=
self.old_src:
reset = True
self.clockgen.setupPLL(mult, num, denom, pllsource=src, reset=reset)
self.old_mult = mult
self.old_num = num
self.old_denom = denom
self.old_src = src
self.clockgen.setupMultisynth(output=0, div=div, num=0, denom=1, pllsource=src)
if r > 1:
self.clockgen.setupRdiv(output=0, div=r)
```

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

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

To get the extra resolution that we bought the module for, the code sets up more than the one integer divider we had in the RP2040. There is a numerator integer, a denominator integer, and a multiplier. There is an extra division step if the frequency is under a megahertz. All of this is so we have enough bits of precision to hit any frequency we wish in a large range.

MFSK stands for Multiple Frequency Shift Keying. Instead of just two tones, as we had in RTTY, sets of either 16 tones or 32 tones are used. The spacing between the tones is very narrow (such as 3.9065 hertz apart), so the bandwidth is quite low. Baud rates range from just under 4 to 125.

Unlike RTTY MFSK has a large character set, made up of 255 characters. All of the ASCII characters are there, as well as many characters for international keyboards, symbols for currency, degrees, and many more.

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

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

```
# -*- coding: latin-1 -*-
mfsk_varicode = [
0b11101011100,  # 000 - <NUL>
0b11101100000,  # 001 - <SOH>
0b11101101000,  # 002 - <STX>
0b11101101100,  # 003 - <ETX>
0b11101110000,  # 004 - <EOT>
0b11101110100,  # 005 - <ENQ>
0b11101111000,  # 006 - <ACK>
0b11101111100,  # 007 - <BEL>
0b10101000,     # 008 - <BS>
0b11110000000,  # 009 - <TAB>
0b11110100000,  # 010 - <LF>
0b11110101000,  # 011 - <VT>
0b11110101100,  # 012 - <FF>
0b10101100,     # 013 - <CR>
0b11110110000,  # 014 - <SO>
0b11110110100,  # 015 - <SI>
0b11110111000,  # 016 - <DLE>
0b11110111100,  # 017 - <DC1>
0b11111000000,  # 018 - <DC2>
0b11111010000,  # 019 - <DC3>
0b11111010100,  # 020 - <DC4>
0b11111011000,  # 021 - <NAK>
0b11111011100,  # 022 - <SYN>
0b11111100000,  # 023 - <ETB>
0b11111101000,  # 024 - <CAN>
0b11111101100,  # 025 - <EM>
0b11111110000,  # 026 - <SUB>
0b11111110100,  # 027 - <ESC>
0b11111111000,  # 028 - <FS>
0b11111111100,  # 029 - <GS>
0b100000000000,  # 030 - <RS>
0b101000000000,  # 031 - <US>
0b100,       # 032 - <SPC>
0b111000000,    # 033 - !
0b111111100,    # 034 - ""
0b1011011000,    # 035 -   #
0b1010101000,    # 036 - $
0b1010100000,    # 037 - %
0b1000000000,    # 038 - &
0b110111100,    # 039 - '
0b111110100,    # 040 - (
0b111110000,    # 041 - )
0b1010110100,    # 042 - *
0b111100000,    # 043 - +
0b10100000,     # 044 - ,
0b111011000,    # 045 - -
0b111010100,    # 046 - .
0b111101000,    # 047 - /
```

```
0b11100000,     # 048 - 0
0b11110000,     # 049 - 1
0b101000000,    # 050 - 2
0b101010100,    # 051 - 3
0b101110100,    # 052 - 4
0b101100000,    # 053 - 5
0b101101100,    # 054 - 6
0b110100000,    # 055 - 7
0b110000000,    # 056 - 8
0b110101100,    # 057 - 9
0b111101100,    # 058 - :
0b111111000,    # 059 - ;
0b1011000000,    # 060 - <
0b111011100,    # 061 - =
0b1010111100,    # 062 - >
0b111010000,    # 063 - ?
0b1010000000,    # 064 - @
0b10111100,     # 065 - A
0b100000000,    # 066 - B
0b11010100,     # 067 - C
0b11011100,     # 068 - D
0b10111000,     # 069 - E
0b11111000,     # 070 - F
0b101010000,    # 071 - G
0b101011000,    # 072 - H
0b11000000,     # 073 - I
0b110110100,    # 074 - J
0b101111100,    # 075 - K
0b11110100,     # 076 - L
0b11101000,     # 077 - M
0b11111100,     # 078 - N
0b11010000,     # 079 - O
0b11101100,     # 080 - P
0b110110000,    # 081 - Q
0b11011000,     # 082 - R
0b10110100,     # 083 - S
0b10110000,     # 084 - T
0b101011100,    # 085 - U
0b110101000,    # 086 - V
0b101101000,    # 087 - W
0b101110000,    # 088 - X
0b101111000,    # 089 - Y
0b110111000,    # 090 - Z
0b1011101000,    # 091 - [
0b1011010000,    # 092 - \
0b1011101100,    # 093 - ]
0b1011010100,    # 094 - ^
0b1010110000,    # 095 - _
0b1010101100,    # 096 - `
0b10100,       # 097 - a
```

```
0b1100000,    # 098 - b
0b111000,     # 099 - c
0b110100,     # 100 - d
0b1000,       # 101 - e
0b1010000,    # 102 - f
0b1011000,    # 103 - g
0b110000,     # 104 - h
0b11000,      # 105 - i
0b10000000,   # 106 - j
0b1110000,    # 107 - k
0b101100,     # 108 - l
0b1000000,    # 109 - m
0b11100,      # 110 - n
0b10000,      # 111 - o
0b1010100,    # 112 - p
0b1111000,    # 113 - q
0b100000,     # 114 - r
0b101000,     # 115 - s
0b1100,       # 116 - t
0b111100,     # 117 - u
0b1101100,    # 118 - v
0b1101000,    # 119 - w
0b1110100,    # 120 - x
0b1011100,    # 121 - y
0b1111100,    # 122 - z
0b1011011100,   # 123 - {
0b1010111000,   # 124 - |
0b1011100000,   # 125 - }
0b1011110000,   # 126 - ~
0b101010000000, # 127 - <DEL>
0b101010100000, # 128 -
0b101010101000, # 129 -
0b101010101100, # 130 -
0b101010110000, # 131 -
0b101010110100, # 132 -
0b101010111000, # 133 -
0b101010111100, # 134 -
0b101011000000, # 135 -
0b101011010000, # 136 -
0b101011010100, # 137 -
0b101011011000, # 138 -
0b101011011100, # 139 -
0b101011100000, # 140 -
0b101011101000, # 141 -
0b101011101100, # 142 -
0b101011110000, # 143 -
0b101011110100, # 144 -
0b101011111000, # 145 -
0b101011111100, # 146 -
0b101100000000, # 147 -
```

```
0b101101000000,  # 148 -
0b101101010000,  # 149 -
0b101101010100,  # 150 -
0b101101011000,  # 151 -
0b101101011100,  # 152 -
0b101101100000,  # 153 -
0b101101101000,  # 154 -
0b101101101100,  # 155 -
0b101101110000,  # 156 -
0b101101110100,  # 157 -
0b101101111000,  # 158 -
0b101101111100,  # 159 -
0b1011110100,    # 160 -  
0b1011111000,    # 161 - ¡
0b1011111100,    # 162 - ¢
0b1100000000,    # 163 - £
0b1101000000,    # 164 - ¤
0b1101010000,    # 165 - ¥
0b1101010100,    # 166 - ¦
0b1101011000,    # 167 - §
0b1101011100,    # 168 - ¨
0b1101100000,    # 169 - ©
0b1101101000,    # 170 - ª
0b1101101100,    # 171 - «
0b1101110000,    # 172 - ¬
0b1101110100,    # 173 - 
0b1101111000,    # 174 - ®
0b1101111100,    # 175 - ¯
0b1110000000,    # 176 - °
0b1110100000,    # 177 - ±
0b1110101000,    # 178 - ²
0b1110101100,    # 179 - ³
0b1110110000,    # 180 - ´
0b1110110100,    # 181 - µ
0b1110111000,    # 182 - ¶
0b1110111100,    # 183 - ·
0b1111000000,    # 184 - ¸
0b1111010000,    # 185 - ¹
0b1111010100,    # 186 - º
0b1111011000,    # 187 - »
0b1111011100,    # 188 - ¼
0b1111100000,    # 189 - ½
0b1111101000,    # 190 - ¾
0b1111101100,    # 191 - ¿
0b1111110000,    # 192 - À
0b1111110100,    # 193 - Á
0b1111111000,    # 194 - Â
0b1111111100,    # 195 - Ã
0b10000000000,   # 196 - Ä
0b10100000000,   # 197 - Å
```

```
0b10101000000,  # 198 - Æ
0b10101010000,  # 199 - Ç
0b10101010100,  # 200 - È
0b10101011000,  # 201 - É
0b10101011100,  # 202 - Ê
0b10101100000,  # 203 - Ë
0b10101101000,  # 204 - Ì
0b10101101100,  # 205 - Í
0b10101110000,  # 206 - Î
0b10101110100,  # 207 - Ï
0b10101111000,  # 208 - Ð
0b10101111100,  # 209 - Ñ
0b10110000000,  # 210 - Ò
0b10110100000,  # 211 - Ó
0b10110101000,  # 212 - Ô
0b10110101100,  # 213 - Õ
0b10110110000,  # 214 - Ö
0b10110110100,  # 215 - ×
0b10110111000,  # 216 - Ø
0b10110111100,  # 217 - Ù
0b10111000000,  # 218 - Ú
0b10111010000,  # 219 - Û
0b10111010100,  # 220 - Ü
0b10111011000,  # 221 - Ý
0b10111011100,  # 222 - Þ
0b10111100000,  # 223 - ß
0b10111101000,  # 224 - à
0b10111101100,  # 225 - á
0b10111110000,  # 226 - â
0b10111110100,  # 227 - ã
0b10111111000,  # 228 - ä
0b10111111100,  # 229 - å
0b11000000000,  # 230 - æ
0b11010000000,  # 231 - ç
0b11010100000,  # 232 - è
0b11010101000,  # 233 - é
0b11010101100,  # 234 - ê
0b11010110000,  # 235 - ë
0b11010110100,  # 236 - ì
0b11010111000,  # 237 - í
0b11010111100,  # 238 - î
0b11011000000,  # 239 - ï
0b11011010000,  # 240 - ð
0b11011010100,  # 241 - ñ
0b11011011000,  # 242 - ò
0b11011011100,  # 243 - ó
0b11011100000,  # 244 - ô
0b11011101000,  # 245 - õ
0b11011101100,  # 246 - ö
0b11011110000,  # 247 - ÷
```

```
0b11011110100,  # 248 - ø
0b11011111000,  # 249 - ù
0b11011111100,  # 250 - ú
0b11100000000,  # 251 - û
0b11101000000,  # 252 - n
0b11101010000,  # 253 - ý
0b11101010100,  # 254 - þ
0b11101011000    # 255 - ÿ
]
```

The mfsk_config.py module connects the Radio and MFSK classes and isolates the rest of the program from their details. Most of it is concerned with setting up the nine different modes, each with a different baud rate and number of tones and symbols. Otherwise, it looks much like the config module for RTTY:

```
from mfsk import MFSK
from time import sleep_ms, sleep
from radio import Radio
class MfskProcess:
def __init__(self, pin, frequency, baud, message, call, location):
from machine import Timer
self.osc = Radio()
self.radio_timer = Timer()
self.old_tone = -1
self.baud = baud
self.usb_offset = 1133
self.frequency = frequency
self.message = message
self.r = MFSK(self.radio_timer, self.send_tone)
self.r.stop()
self.r.set_call(call)
self.r.set_location(location)
self.r.set_frequency(frequency)
self.r.set_message(message)
if self.baud == 4:                    # 3.90625 baud
self.r.samplerate = 8000.0
self.r.symlen = 2048.0
self.r.symbits = 5
self.r.depth = 5
self.r.basetone = 256
self.r.numtones = 32
self.r.preamble = 107
elif self.baud == 8:                 # 7.8125 baud
self.r.samplerate = 8000.0
self.r.symlen = 1024.0
self.r.symbits = 5
self.r.depth = 5
self.r.basetone = 128
self.r.numtones = 32
self.r.preamble = 107
elif self.baud == 11:                # 10.7666015625 baud
self.r.samplerate = 11025.0
```

```
self.r.symlen = 1024.0
self.r.symbits = 4
self.r.depth = 10
self.r.basetone = 93
self.r.numtones = 16
self.r.preamble = 107
elif self.baud == 16:                # 15.625 baud
self.r.samplerate = 8000.0
self.r.symlen = 512.0
self.r.symbits = 4
self.r.depth = 10
self.r.basetone = 64
self.r.numtones = 16
self.r.preamble = 107
elif self.baud == 22:                # 21.533203125 baud
self.r.samplerate = 11025.0
self.r.symlen = 512.0
self.r.symbits = 4
self.r.depth = 10
self.r.basetone = 46
self.r.numtones = 16
self.r.preamble = 107
elif self.baud == 31:                # 31.25 baud
self.r.samplerate = 8000.0
self.r.symlen = 256.0
self.r.symbits = 3
self.r.depth = 10
self.r.basetone = 32
self.r.numtones = 8
self.r.preamble = 107
elif self.baud == 32:                # 31.25 baud
self.r.samplerate = 8000.0
self.r.symlen = 256.0
self.r.symbits = 4
self.r.depth = 10
self.r.basetone = 32
self.r.numtones = 16
self.r.preamble = 107
elif self.baud == 64:                # 62.5 baud
self.r.samplerate = 8000.0
self.r.symlen = 128.0
self.r.symbits = 4
self.r.depth = 10
self.r.basetone = 16
self.r.numtones = 16
self.r.preamble = 180
elif self.baud == 128:                # 125 baud
self.r.samplerate = 8000.0
self.r.symlen = 64.0
self.r.symbits = 4
```

```
self.r.depth = 20
self.r.basetone = 8
self.r.numtones = 16
self.r.preamble = 214
self.r.tonespacing = self.r.samplerate / self.r.symlen
print(”Frequency:”, self.frequency)
print(”Message:”, self.message)
print(”Symbits is”, self.r.symbits)
print(”Depth is”, self.r.depth)
print(”Bandwidth is”, (self.r.numtones - 1) * self.r.tonespacing)
print(”Symbol length is”, self.r.symlen)
print(”Baud is”, self.r.samplerate / self.r.symlen)
print(”Tonespacing is”, str(self.r.tonespacing) + “:”)
self.send_code()
def get_radio(self):
return self.osc
def set_message(self, msg):
self.message = msg
def send_code(self):
self.f = float(self.r.basetone + float(self.frequency) + self.usb_offset)
self.osc.set_freq(0, self.f)
start_of_transmission_length = int(8 * (1000 / (self.r.samplerate / self.r.symlen)))
sleep_ms(start_of_transmission_length)
self.r.send_code()
def send_tone(self, tone):
if tone != self.old_tone:
f = float(float(self.frequency) + self.usb_offset + float(tone))
self.osc.set_freq(0, f)
self.old_tone = tone
```

The mfsk.py module handles the actual encoding of the data. This is quite involved, as there is forward error correction to allow it to handle noisy environments. The error correction code was developed by NASA for space probes. The bits are also interleaved (high bits are sent, then next high, etc.) to allow the error correction code to handle bursts of static that would otherwise damage several bits in a row. The bits are also Gray coded, so that adjacent symbols differ by only one bit, which helps to reduce errors in the decoding.

The code looks like this:

```
from mfsk_varicode import mfsk_varicode
from machine import Timer
class MFSK:
NASA_K = 7
POLY1 = 0x6D
POLY2 = 0x4F
def __init__(self, timr, send_tone):
self.timer = timr
self.send_tone = send_tone
#
# Default is MFSK4
#
self.symbits = 5
self.symlen = 2048
```

```
self.samplerate = 8000
self.depth = 5
self.basetone = 256
self.numtones = 32
self.preamble = 107
self.timer_running = False
self.frequency = 7104000.0
self.call = ”
self.location = ”
self.message = “{} {}   “
self.count_tabs = 0
self.has_bits = False
self.sym_queue = []
# Initialization for the forward error correction
self.encoder_output = [0] * (1 << self.NASA_K)
self.mask = (1 << self.NASA_K) - 1
self.encode_state = 0
self.bit_count = 0
self.bit_state = 0
# Code for the forward error correction
def init_encoder(self):
self.interleave_table = [8] * (self.symbits * self.symbits * self.depth)
for x in range(1 << self.NASA_K):
self.encoder_output[x] = (self.parity(self.POLY1 & x) | (self.parity(self.POLY2 &x) << 1))
self.flush_interleave_table()
# Hamming weight (the number of bits that are ones)
def hamming_weight(self, w):
w = (w & 0x55555555) + ((w >>  1) & 0x55555555)
w = (w & 0x33333333) + ((w >>  2) & 0x33333333)
w = (w & 0x0F0F0F0F) + ((w >>  4) & 0x0F0F0F0F)
w = (w & 0x00FF00FF) + ((w >>  8) & 0x00FF00FF)
w = (w & 0x0000FFFF) + ((w >> 16) & 0x0000FFFF)
return w
def parity(self, w):
return self.hamming_weight(w) & 1
def encode(self, bit):
self.encode_state <<= 1
if bit == “1”:
self.encode_state |= 1
return self.encoder_output[self.encode_state & self.mask]
def set_call(self, call):
self.call = call
def set_baud(self, baud):
self.baud = float(baud)
def set_bit_length(self, len):
self.bit_length = 1000000.0 / float(self.baud)
def set_frequency(self, frequency):
self.frequency = float(frequency)
def set_location(self, location):
self.location = location
```

```
def set_message(self, message):
self.message = message.format(self.call, self.location)
self.message = “\r” + chr(2) + “\r” + self.message + “\r” + chr(0) + “\r”
self.has_bits = True
def bit(self):
global mfsk_varicode
for letter in self.message:
code = mfsk_varicode[ord(letter) & 255]
for bit in bin(code)[2:]:
yield bit
def stop(self):
self.timer.deinit()
self.timer_running = False
self.all_done = True
def send_code(self):
self.set_baud(self.samplerate / self.symlen)
self.bit_length = 1000000.0 / float(self.baud)
self.tonespacing = self.samplerate / self.symlen
self.bandwidth = (self.numtones - 1) * self.tonespacing
self.init_encoder()
self.all_done = False
self.clearbits()
self.gen = self.bit()
if self.timer_running == False:
self.timer.init(period=int(self.bit_length/1000), mode=Timer.PERIODIC, callback=self.next_tone)
self.timer_running = True
self.reported_end = False
self.has_bits = True
while self.has_bits:
bit = self.get_bit()
self.send_bit(bit)
self.flush_tx(self.preamble)
self.reported_end = True
def get_bit(self):
try:
bit = next(self.gen)
except StopIteration as e:
self.has_bits = False
return None
return bit
def send_bit(self, bit):
try:
data = self.encode(bit)
for x in range(2):
self.bit_state = (self.bit_state << 1) | ((data >> x) & 1)
self.bit_count += 1
if self.bit_count == self.symbits:
self.interleave()
self.send_symbol()
self.bit_count = 0
```

```
self.bit_state = 0
except Exception as e:
print("Error:", e)
def clearbits(self):
data = self.encode(0)
for x in range(self.preamble):
for y in range(2):
self.bit_state = (self.bit_state << 1) | ((data >> x) & 1)
self.bit_count += 1
if self.bit_count == self.symbits:
self.interleave()
self.bit_count = 0
self.bit_state = 0
def interleave_get(self, x, y, z):
index = self.symbits * self.symbits * x + self.symbits * y + z
return self.interleave_table[index]
def interleave_put(self, x, y, z, val):
index = self.symbits * self.symbits * x + self.symbits * y + z
self.interleave_table[index] = val
def symbols(self):
for x in range(self.depth):
for y in range(self.symbits):
for z in range(self.symbits - 1):
self.interleave_put(x, y, z, self.interleave_get(x, y, z + 1))
for y in range(self.symbits):
self.interleave_put(x, y, self.symbits-1, self.syms[y])
for y in range(self.symbits):
self.syms[y] = self.interleave_get(x, y, self.symbits - y - 1)
def interleave(self):
self.syms = []
for x in range(self.symbits):
self.syms.append(self.bit_state >> ((self.symbits - x - 1)) & 1)
self.symbols()
self.bit_state = 0
for x in range(self.symbits):
self.bit_state = (self.bit_state << 1) | self.syms[x]
def flush_interleave_table(self):
for x in range(len(self.interleave_table)):
self.interleave_table[x] = 0
def flush_tx(self, preamble):
self.send_bit(chr(1));
for x in range(preamble):
self.send_bit(chr(0));
self.bit_state = 0
self.all_done = True
#
# In order to reduce the number of bit errors in a digital modem,
# all symbols are automatically Gray encoded such that adjacent
# symbols in a constellation differ by only one bit.
#
```

```
def gray_encode(self, data):
bits = data;
bits ^= data >> 1;
bits ^= data >> 2;
bits ^= data >> 3;
bits ^= data >> 4;
bits ^= data >> 5;
bits ^= data >> 6;
bits ^= data >> 7;
return bits;
def send_symbol(self):
from uasyncio import sleep_ms
import utime
sym = self.bit_state & (self.numtones - 1)
sym = self.gray_encode(sym)
while len(self.sym_queue) > 10:
# 256000 / 500 is 512 milliseconds (2 symbols at 3.90625 baud)
sleep_ms(int(self.bit_length / 500))           # Needed so the web server gets some time
utime.sleep_ms(int(self.bit_length / 500))     # Needed so ^C works
self.sym_queue.append(sym)
def sendchar(self, ch):
code = mfsk_varicode[ord(ch) & 255]
for bit in bin(code)[2:]:
self.send_bit(bit);
def sendidle(self):
self.sendchar(chr(0));
def next_tone(self, unused):
if self.sym_queue:
sym = self.sym_queue.pop(0)
self.send_tone(self.basetone + sym * self.tonespacing)
```

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

```
from mfsk_config import MfskProcess
from machine import Pin
from time import sleep
def main():
mp = MfskProcess(15, 7040000, 4, ”, “AB6NY”, “CM87xe”)
mp.set_message(”{} Testing from {} using a Raspberry Pi Pico RP2040”)
while True:
mp.send_code()
while mp.r.all_done == False:
sleep(5)
main()
```

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

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

Mfsk

Python Programming

Python Radio 43: Super High Frequency Radar

Simon Quellen Field
Simon Quellen Field
Follow
5 min read
·
Jun 16, 2025
Listen
Share
More

24 Gigahertz radar in the 1.25 centimeter band

Press enter or click to view image in full size
24 GHz Radar
All photos by the author.

In the previous chapter on radar, we used a device that operated at 3 GHz, resulting in a 10-centimeter wavelength. But we can now buy radar devices for less than $10 that deliver 24 gigahertz. That’s a wavelength of 1.25 centimeters, placing it in the middle of the K band (Super High Frequency, or SHF).

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

I chose the ESP32-C3 Super Mini with the tiny OLED display for this part of the project. Here is the micropython code:

```
From machine import UART, I2C, Pin
From SH1106 import SH1106_I2C
From sys import print_exception
From time import sleep_ms
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
# self.display.rotate(1)
Def oled(self, s, line, column):
Self.display.fill_rect(self.xOffset, self.yOffset + 2 + line * 9, 128-self.xOffset, 9, 0)
Self.display.text(s, self.xOffset + 2 + column * 8, self.yOffset + 2 + line * 9, 1)
Self.display.show()
D = None
Try:
D = Display()
Except OSError as e:
If e == 19: # ENODEV
D = None
# d.display.invert(0)
Uart = UART(1, baudrate=115200, tx=1, rx=2)
Def main():
If d:
d.oled(“Range:”, 0, 0)
d.oled(“cm”, 2, 0)
```

```
cm = “”
while True:
sleep_ms(1)
if uart.any():
line = uart.readline()
if line:
try:
cm = line.decode(“utf-8”)
except Exception as e:
print_exception(e)
print(“Line:”, line)
if “Range” in cm:
cm = cm[6:].strip()
print(cm)
if d:
d.oled(cm, 1, 0)
main()
```

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

The rest of the code loops, getting a line from the UART and parsing it to get the range to the target (a person in the typical use case for an automatic door opener).

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

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

Press enter or click to view image in full size

Radar with two receiving antennas.

Instead of using micropython, we will use regular Python on a Windows computer for a change. Linux or Macintosh will also work, using a different UART port name.

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

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

```
From time import sleep
From serial import Serial
Import matplotlib.pyplot as plt
Uart = Serial(“com7”, 256000)
Def convert_sign(x):
If x & 0x8000:
X = x – 0x8000
Else:
X = -x
Return x
Def main():
Fig = plt.figure(figsize=(7, 7))
Plt.ion()
Plt.show()
While True:
Sleep(0.001)
```

```
Line = uart.read(30)
If line and len(line) >= 29:
# print(f”{hex(int.from_bytes(line, ‘little’))}”)
Header = int.from_bytes(line[:4], ‘little’)
A_x = convert_sign(int.from_bytes(line[4:6], ‘little’))
A_y = convert_sign(int.from_bytes(line[6:8], ‘little’))
A_speed = convert_sign(int.from_bytes(line[8:10], ‘little’))
A_resolution = int.from_bytes(line[10:12], ‘little’)
B_x = convert_sign(int.from_bytes(line[12:14], ‘little’))
B_y = convert_sign(int.from_bytes(line[14:16], ‘little’))
B_speed = convert_sign(int.from_bytes(line[16:18], ‘little’))
B_resolution = int.from_bytes(line[18:20], ‘little’)
C_x = convert_sign(int.from_bytes(line[20:22], ‘little’))
C_y = convert_sign(int.from_bytes(line[22:24], ‘little’))
C_speed = convert_sign(int.from_bytes(line[24:26], ‘little’))
C_resolution = int.from_bytes(line[26:28], ‘little’)
End_frame = int.from_bytes(line[28:], ‘little’)
Plt.clf()
Fig.text((500+a_x)/1000, a_y/5000, f”o”)
Fig.text(0, 0, f”{a_x} mm, {a_y} mm, {a_speed} cm/sec”)
Fig.canvas.draw()
Plt.pause(.02)
Plt.show()
# print(f”{hex(header)}”)
Print(f”X: {a_x} mm”, end=” “)
Print(f”Y: {a_y} mm”, end=” “)
Print(f”Speed: {a_speed} cm/sec”, end=” “)
Print(“\r”, end=””)
# print(f”Resolution: {hex(a_resolution)}, {a_resolution}”)
# print(f”X: {hex(b_x)}, {b_x} mm”)
# print(f”Y: {hex(b_y)}, {b_y} mm”)
# print(f”Speed: {hex(b_speed)}, {b_speed} cm/sec”)
# print(f”Resolution: {hex(b_resolution)}, {b_resolution} mm”)
# print(f”X: {hex(c_x)}, {c_x} mm”)
# print(f”Y: {hex(c_y)}, {c_y} mm”)
# print(f”Speed: {hex(c_speed)}, {c_speed} cm/sec”)
# print(f”Resolution: {hex(c_resolution)}, {c_resolution} mm”)
# print(f”{hex(end_frame)}”)
Main()
```

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

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

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

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

In between are three groups of eight bytes (0E 03 B1 86 10 00 40 01, 00 00 00 00 00 00 00 00, and 00 00 00 00 00 00 00 00). Each one is data for a target. The device can track three targets at once. In the example, there is only one target. The other two groups are all zeroes.

The data for a target has four 16-bit numbers, in “little-endian” format, that describe the X coordinate in millimeters, the Y coordinate in millimeters, the speed in centimeters per second, and the resolution (always 320 millimeters).

The sign bit is backwards from what we see in normal two’s-complement arithmetic, with a 0 meaning

negative and a 1 meaning positive. We handle this unusual format with a convert_sign() subroutine. The resolution (weirdly) does not use the sign bit convention.
Because we are running on a computer with a screen, we can plot the result using Matplotlib in Python. We set interactive mode (plt.ion()), adjust the coordinates to screen coordinates, and draw a little “o” to represent the person being tracked.
Since I am working alone here in the lab, I commented out the code for the other two targets.
Here is what it looks like:
The 24 GHz Radar screen in action.
The distance resolution claims about one foot (32 centimeters), but the data from the device shows single-centimeter resolution, which I take with about 320 grains of salt. But averaging a few hundred readings might get us a pretty accurate distance, perhaps with centimeter resolution.
The range of the device is about 8 meters (about 25 feet). It is most sensitive to human bodies, perhaps by ignoring things that don’t move. It might even be possible to use it to detect a heartbeat with a little digital signal processing.
Python Radio 19: Thor Mode
Follow
7 min read
·
Sep 6, 2024
Listen
Share
More
The best of both worlds…
Press enter or click to view image in full size
Image by the author
What if we combined the forward error correction of MFSK with the robustness and ease of tuning of FSQ and DominoEX?
We would get Thor.
The screenshot above shows Thor11, the 11 baud mode that gave us trouble in DominoEX.
The random characters in between the lines of perfect copy are noise from the radio during the periods when we weren’t sending.
Thor uses the same varicode as MFSK. It also has a shorter Thor varicode that is used to send callsigns when the system is idling. That looks like this:
```
thor_varicode = [
0b101110000000,  # 032 - <SPC>
0b101110100000,  # 033 - !
0b101110101000,  # 034 - ’
0b101110101100,  # 035 - #
0b101110110000,  # 036 - $
0b101110110100,  # 037 - %
0b101110111000,  # 038 - &
0b101110111100,  # 039 - ‘
0b101111000000,  # 040 - (
0b101111010000,  # 041 - )
0b101111010100,  # 042 - *
0b101111011000,  # 043 - +
0b101111011100,  # 044 - ,
0b101111100000,  # 045 - -
0b101111101000,  # 046 - .
0b101111101100,  # 047 - /
```

```
0b101111110000,  # 048 - 0
0b101111110100,  # 049 - 1
0b101111111000,  # 050 - 2
0b101111111100,  # 051 - 3
0b110000000000,  # 052 - 4
0b110100000000,  # 053 - 5
0b110101000000,  # 054 - 6
0b110101010100,  # 055 - 7
0b110101011000,  # 056 - 8
0b110101011100,  # 057 - 9
0b110101100000,  # 058 - :
0b110101101000,  # 059 - ;
0b110101101100,  # 060 - <
0b110101110000,  # 061 - =
0b110101110100,  # 062 - >
0b110101111000,  # 063 - ?
0b110101111100,  # 064 - @
0b110110000000,  # 065 - A
0b110110100000,  # 066 - B
0b110110101000,  # 067 - C
0b110110101100,  # 068 - D
0b110110110000,  # 069 - E
0b110110110100,  # 070 - F
0b110110111000,  # 071 - G
0b110110111100,  # 072 - H
0b110111000000,  # 073 - I
0b110111010000,  # 074 - J
0b110111010100,  # 075 - K
0b110111011000,  # 076 - L
0b110111011100,  # 077 - M
0b110111100000,  # 078 - N
0b110111101000,  # 079 - O
0b110111101100,  # 080 - P
0b110111110000,  # 081 - Q
0b110111110100,  # 082 - R
0b110111111000,  # 083 - S
0b110111111100,  # 084 - T
0b111000000000,  # 085 - U
0b111010000000,  # 086 - V
0b111010100000,  # 087 - W
0b111010101100,  # 088 - X
0b111010110000,  # 089 - Y
0b111010110100,  # 090 - Z
0b111010111000,  # 091 - [
0b111010111100,  # 092 - \
0b111011000000,  # 093 - ]
0b111011010000,  # 094 - ^
0b111011010100,  # 095 - _
0b111011011000,  # 096 - `
0b111011011100,  # 097 - a
```

```
0b111011100000,  # 098 - b
0b111011101000,  # 099 - c
0b111011101100,  # 100 - d
0b111011110000,  # 101 - e
0b111011110100,  # 102 - f
0b111011111000,  # 103 - g
0b111011111100,  # 104 - h
0b111100000000,  # 105 - i
0b111101000000,  # 106 - j
0b111101010000,  # 107 - k
0b111101010100,  # 108 - l
0b111101011000,  # 109 - m
0b111101011100,  # 110 - n
0b111101100000,  # 111 - o
0b111101101000,  # 112 - p
0b111101101100,  # 113 - q
0b111101110000,  # 114 - r
0b111101110100,  # 115 - s
0b111101111000,  # 116 - t
0b111101111100,  # 117 - u
0b111110000000,  # 118 - v
0b111110100000,  # 119 - w
0b111110101000,  # 120 - x
0b111110101100,  # 121 - y
0b111110110000  # 122 - z
]
```

The thor_config.py module looks familiar by now:

```
from thor import THOR
from time import sleep_ms, sleep
from machine import Timer
from radio import Radio
class ThorConfig:
def __init__(self, baud, frq, call, location):
self.dds = Radio()
self.dds.on()
self.dds.send()
self.radio_timer = Timer()
self.baud = baud
self.message = ''
self.frequency = frq
self.usb_offset = 1133
self.all_done = False
self.call = call
self.location = location
self.r = THOR(self.radio_timer, self.send_tone, self.report_all_done)
self.r.set_call(call)
self.r.set_location(location)
self.r.set_frequency(float(frq))
self.r.symbits = 4
self.r.depth = 10
```

```
self.r.numtones = 18
self.r.preamble = 4
if baud == 2:                    # 2.0 baud
self.r.samplerate = 8000.0
self.r.symlen = 4000.0
self.r.doublespaced = 1
self.r.depth = 4
elif baud == 4:                  # 3.90625 baud
self.r.samplerate = 8000.0
self.r.symlen = 2048.0
self.r.doublespaced = 2
elif baud == 5:                  # 5.38330078125 baud
self.r.samplerate = 11025.0
self.r.symlen = 2048.0
self.r.doublespaced = 2
elif baud == 8:                  # 7.8125 baud
self.r.samplerate = 8000.0
self.r.symlen = 1024.0
self.r.doublespaced = 2
elif baud == 11:                 # 10.7666015625 baud
self.r.samplerate = 11025.0
self.r.symlen = 1024.0
self.r.doublespaced = 1
elif baud == 16:                 # 15.625 baud
self.r.samplerate = 8000.0
self.r.symlen = 512.0
self.r.doublespaced = 1
elif baud == 22:                 # 21.533203125 baud
self.r.samplerate = 11025.0
self.r.symlen = 512.0
self.r.doublespaced = 1
self.r.tonespacing = self.r.samplerate / self.r.symlen
self.r.init_params()
print(”Frequency:”, self.frequency)
print(”Symbits is”, self.r.symbits)
print(”Depth is”, self.r.depth)
print(”Bandwidth is”, (self.r.numtones - 1) * self.r.tonespacing)
print(”Symbol length is”, self.r.symlen)
print(”Baud is”, self.r.samplerate / self.r.symlen)
print(”Bit length is”, self.r.bit_length)
print(”Tonespacing is”, str(self.r.tonespacing))
def get_radio(self):
return self.dss
def set_message(self, msg):
self.r.set_message(msg.format(self.call, self.location))
self.dds.on()
self.dds.send()
self.r.send_code()
def send_code(self):
self.r.send_code()
```

```
def send_tone(self, tone):
self.f = float(float(self.frequency) + self.usb_offset + float(tone))
self.dds.set_freq(0, self.f)
self.dds.on()
self.dds.send()
def report_all_done(self):
self.all_done = True
print(”All done!”)
```
As with DominoEX, most of the code is just setting up the different baud rates.
The real work is done in the thor.py module:
```
from thor_varicode import thor_varicode
from mfsk_varicode import mfsk_varicode
from machine import Timer
class THOR:
NASA_K = 7
POLY1 = 0x6D
POLY2 = 0x4F
def __init__(self, timr, send_tone, report_message_end=None):
self.timer = timr
self.send_tone = send_tone
self.report_message_end = report_message_end
self.QUEUE_LENGTH = 80
#
# Default is THOR MICRO
#
self.symbits = 4
self.symlen = 4000
self.samplerate = 8000
self.depth = 4
self.doublespaced = 1
self.basetone = 256
self.numtones = 18
self.preamble = 10
self.timer_running = False
self.bandwidth = 0
self.tonespacing = 0
self.secondary = False
self.previous_tone = 0
self.frequency = 7104000.0
self.call = ''
self.location = ''
self.message = '{} {}    '
self.count_tabs = 0
self.has_bits = False
self.sym_queue = []
# Initialization for the forward error correction
self.encoder_output = [0] * (1 << self.NASA_K)
self.mask = (1 << self.NASA_K) - 1
self.encode_state = 0
self.bit_count = 0
```

```
self.bit_state = 0
# Code for the forward error correction
def init_encoder(self):
self.interleave_table = [8] * (self.symbits * self.symbits * self.depth)
for x in range(1 << self.NASA_K):
self.encoder_output[x] = (self.parity(self.POLY1 & x) | (self.parity(self.POLY2 &x) << 1))
self.flush_interleave_table()
# Hamming weight (the number of bits that are ones)
def hamming_weight(self, w):
w = (w & 0x55555555) + ((w >>  1) & 0x55555555)
w = (w & 0x33333333) + ((w >>  2) & 0x33333333)
w = (w & 0x0F0F0F0F) + ((w >>  4) & 0x0F0F0F0F)
w = (w & 0x00FF00FF) + ((w >>  8) & 0x00FF00FF)
w = (w & 0x0000FFFF) + ((w >> 16) & 0x0000FFFF)
return w
def parity(self, w):
return self.hamming_weight(w) & 1
def encode(self, bit):
self.encode_state <<= 1
if bit == “1”:
self.encode_state |= 1
return self.encoder_output[self.encode_state & self.mask]
def set_call(self, call):
self.call = call
def set_baud(self, baud):
self.baud = float(baud)
def set_bit_length(self, len):
self.bit_length = 1000000.0 / float(self.baud)
def set_frequency(self, frequency):
self.frequency = float(frequency)
def set_location(self, location):
self.location = location
def set_message(self, message):
self.message = “\r” + chr(2) + “\r” + message + “\r” + chr(0) + “\r”
self.message += chr(0) + chr(0) + chr(0) + chr(0) + chr(0) + chr(0) + chr(0)
self.message += chr(0) + chr(0) + chr(0) + chr(0) + chr(0) + chr(0) + chr(0)
self.has_bits = True
def init_params(self):
self.set_baud(self.samplerate / self.symlen)
self.tonespacing = self.samplerate * self.doublespaced / self.symlen
self.bandwidth = (self.numtones - 1) * self.tonespacing
self.basetone = int(1500.0 * self.symlen / self.samplerate + 0.5)
self.bit_length = 1000000.0 / float(self.baud)
def bit(self):
for letter in self.message:
code = mfsk_varicode[0]
if self.secondary:
if ord(letter) >= 0 and ord(letter) < 256:
code = thor_varicode[ord(letter) & 255]
else:
```

```
code = mfsk_varicode[ord(letter) & 255]
for bit in bin(code)[2:]:
yield bit
def stop(self):
self.timer.deinit()
self.timer_running = False
def send_code(self):
self.init_params()
self.init_encoder()
self.is_done = False
self.clearbits()
self.gen = self.bit()
if self.timer_running == False:
self.timer.init(period=int(self.bit_length/1000), mode=Timer.PERIODIC, callback=self.next_tone)
self.timer_running = True
self.reported_end = False
self.has_bits = True
# Send 64 zero bits to flush the receive decoder
for x in range(16):
self.bit_state = 0
self.send_symbol()
while self.has_bits:
bit = self.get_bit()
self.send_bit(bit)
self.flush_tx(self.preamble)
def get_bit(self):
try:
bit = next(self.gen)
except StopIteration as e:
self.has_bits = False
return None
return bit
def send_bit(self, bit):
try:
data = self.encode(bit)
for x in range(2):
self.bit_state = (self.bit_state << 1) | ((data >> x) & 1)
self.bit_count += 1
if self.bit_count == self.symbits:
self.interleave()
self.send_symbol()
self.bit_count = 0
self.bit_state = 0
except Exception as e:
print(”Error:”, e)
def clearbits(self):
data = self.encode(0)
for x in range(1400):
for y in range(2):
self.bit_state = (self.bit_state << 1) | ((data >> x) & 1)
```

```
self.bit_count += 1
if self.bit_count == self.symbits:
self.interleave()
self.bit_count = 0
self.bit_state = 0
def interleave_get(self, x, y, z):
index = self.symbits * self.symbits * x + self.symbits * y + z
return self.interleave_table[index]
def interleave_put(self, x, y, z, val):
index = self.symbits * self.symbits * x + self.symbits * y + z
self.interleave_table[index] = val
def symbols(self):
for x in range(self.depth):
for y in range(self.symbits):
for z in range(self.symbits - 1):
self.interleave_put(x, y, z, self.interleave_get(x, y, z + 1))
for y in range(self.symbits):
self.interleave_put(x, y, self.symbits-1, self.syms[y])
for y in range(self.symbits):
self.syms[y] = self.interleave_get(x, y, self.symbits - y - 1)
def interleave(self):
self.syms = []
for x in range(self.symbits):
self.syms.append(self.bit_state >> ((self.symbits - x - 1)) & 1)
self.symbols()
self.bit_state = 0
for x in range(self.symbits):
self.bit_state = (self.bit_state << 1) | self.syms[x]
def flush_interleave_table(self):
for x in range(len(self.interleave_table)):
self.interleave_table[x] = 0
def flush_tx(self, preamble):
for x in range(preamble):
self.sendidle();
self.bit_state = 0
self.is_done = True
self.report_message_end()
def send_symbol(self):
from time import sleep_ms
sym = self.bit_state
while len(self.sym_queue) > self.QUEUE_LENGTH:
sleep_ms(int(self.bit_length / 50000))          # Needed so ^C works
self.sym_queue.append(sym)
def sendchar(self, letter, secondary):
code = mfsk_varicode[0]
if self.secondary:
if ord(letter) >= 0 and ord(letter) < 256:
code = thor_varicode[ord(letter) & 255]
else:
code = mfsk_varicode[ord(letter) & 255]
```

```
for bit in bin(code)[2:]:
self.send_bit(bit);
def sendidle(self):
for x in range(8):
self.sendchar(chr(0), 0);
def next_tone(self, unused):
if self.sym_queue:
sym = self.sym_queue.pop(0)
tone = (self.previous_tone + 2 + sym) % self.numtones
self.previous_tone = tone
self.send_tone(self.basetone + tone * self.tonespacing)
```

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

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

Because of the forward error correction, the symbols are placed in a queue for sending, which is what send_symbol() manages. The send_char() method sends from either the Thor or the MFSK varico tables, depending on its argument.

Amateur Radio

Python Programming

Python Radio 17: FSQ Mode

Simon Quellen Field
Simon Quellen Field

Follow

7 min read

·

Sep 4, 2024

Listen

Share

More

Fast Simple QSO

Press enter or click to view image in full size

Image by author

FSQ stands for Fast Simple QSO (QSO means conversation). It is a chat mode in amateur radio that h several advantages over MFSK and RTTY.

Like MFSK, it uses a varicode, so that frequently used characters are sent faster. It uses 33 tones so that all of the lower-case characters (and a few more, such as space, period, and carriage return) can be sent with a single tone. FSQ6 (the 6 is the baud rate) can send 60 words per minute if lowercase letters are used. The baud rate (number of symbols per second) can be anything from 2 to 6. A symbol is a single letter if in lowercase, and two symbols are needed for other characters.

The receiver does not need to change anything as the baud rate changes, so the sender can adjust the baud rate to the propagation conditions, slowing down when there is a lot of noise. The slower speeds are more robust to interference and to weak signals.

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

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

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

Very low-power FSQ is also a good mode for exploring radio propagation. If your 50-milliwatt signal can be heard, then it is likely that any mode will work at that frequency and distance. This is known as MEPT. That stands for Manned Experimental Propagation Transmitter. Even though the com is doing the sending, it is assumed that there is a person controlling and monitoring the transmissions, hence the “manned” part of the acronym.

It is also good for telemetry. You can add a sensor such as the DHT22 temperature-humidity sensor or the BME280 temperature-pressure-humidity sensor and transmit the readings to your whole neighborhood.

In FLDIGI, the reception looks like this:

Press enter or click to view image in full size

Image by author

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

We will describe here the modulator (as we did with other modes), but FSQ is usually used with the FSQCall program, which adds several automated functions making it especially useful for networks anc emergency communications. We have hooks in the modulator for those features, such as directed messaging (call-sign to call-sign, as opposed to a message for everyone). FSQCall also provides error-corrected file transfers.

The varicode for FSQ has four tables. There is one for the 26 lowercase letters, space, period, and carriage return, making 28 tones. The remaining tones 29, 30, and 31 are used to select the other three tables, respectively. Since FSQ uses the difference between two tones as the symbol, the 33 tones allow for 32 symbols.

Here is the fsq_varicode.py module:

```
Fsq_varicode = {
‘ ‘:    ( 0,  0),
‘a’:    ( 1,  0),
‘b’:    ( 2,  0),
‘c’:    ( 3,  0),
‘d’:    ( 4,  0),
‘e’:    ( 5,  0),
‘f’:    ( 6,  0),
‘g’:    ( 7,  0),
‘h’:    ( 8,  0),
‘i’:    ( 9,  0),
‘j’:    (10,  0),
‘k’:    (11,  0),
‘l’:    (12,  0),
‘m’:    (13,  0),
‘n’:    (14,  0),
‘o’:    (15,  0),
‘p’:    (16,  0),
‘q’:    (17,  0),
‘r’:    (18,  0),
‘s’:    (19,  0),
‘t’:    (20,  0),
‘u’:    (21,  0),
‘v’:    (22,  0),
‘w’:    (23,  0),
```

```
‘x’:    (24,  0),
‘y’:    (25,  0),
‘z’:    (26,  0),
‘.’:    (27,  0),
‘\r’:   (28,  0), # Carriage return and line feed (newline)
‘@’:    ( 0, 29),
‘A’:    ( 1, 29),
‘B’:    ( 2, 29),
‘C’:    ( 3, 29),
‘D’:    ( 4, 29),
‘E’:    ( 5, 29),
‘F’:    ( 6, 29),
‘G’:    ( 7, 29),
‘H’:    ( 8, 29),
‘I’:    ( 9, 29),
‘J’:    (10, 29),
‘K’:    (11, 29),
‘L’:    (12, 29),
‘M’:    (13, 29),
‘N’:    (14, 29),
‘O’:    (15, 29),
‘P’:    (16, 29),
‘Q’:    (17, 29),
‘R’:    (18, 29),
‘S’:    (19, 29),
‘T’:    (20, 29),
‘U’:    (21, 29),
‘V’:    (22, 29),
‘W’:    (23, 29),
‘X’:    (24, 29),
‘Y’:    (25, 29),
‘Z’:    (26, 29),
‘,’:    (27, 29),
‘?’:    (28, 29),
‘~’:    ( 0, 30),
‘1’:    ( 1, 30),
‘2’:    ( 2, 30),
‘3’:    ( 3, 30),
‘4’:    ( 4, 30),
‘5’:    ( 5, 30),
‘6’:    ( 6, 30),
‘7’:    ( 7, 30),
‘8’:    ( 8, 30),
‘9’:    ( 9, 30),
‘0’:    (10, 30),
‘!’:    (11, 30),
‘”’:    (12, 30),
‘#’:    (13, 30),
‘$’:    (14, 30),
‘%’:    (15, 30),
```

```
‘&’:    (16, 30),
‘\”:    (17, 30),
‘(‘:    (18, 30),
‘)’:    (19, 30),
‘*’:    (20, 30),
‘+’:    (21, 30),
‘-‘:    (22, 30),
‘/’:    (23, 30),
‘:’:    (24, 30),
‘;’:    (25, 30),
‘<’:    (26, 30),
‘>’:    (27, 30),
0:      (28, 30), # IDLE
‘=’:    ( 0, 31),
‘[‘:    ( 1, 31),
‘\\’:   ( 2, 31),
‘]’:    ( 3, 31),
‘^’:    ( 4, 31),
‘_’:    ( 5, 31),
‘`’:    ( 9, 31),
‘{‘:    ( 6, 31),
‘|’:    ( 7, 31),
‘}’:    ( 8, 31),
‘`’:    ( 9, 31),
‘\u00B1’: (10, 31), # plus/minus
‘\u00F7’: (11, 31), # division sign
‘\u00B0’: (12, 31), # degrees sign
‘\u00D7’: (13, 31), # multiply sign
‘\u00A3’: (14, 31), # pound sterling sign
‘\b’:   (27, 31), # BS
‘\u007F’: (28, 31), # DEL
}
```

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

```
From fsq import FSQ
From time import sleep
From radio import Radio
Class FSQConfig:
Def __init__(self, frq, baud, call, location):
Self.frequency = frq
Self.baud = baud
Self.mycall = call
Self.location = location
Self.r = FSQ(self.send_tone, self.baud, self.all_done)
Self.dds = Radio()
Self.dds.send()
Self.is_beacon = False
Self.message = ”
Self.spacing = 8.7890625            # 8.7890625 Hz
Self.usb_offset = 1350.0
```

```
Self.is_directed = False
Self.tocall = "N0CALL"
Self.beacon_interval = 60.0
Self.incremental_tone = 0.0
Self.r.set_frequency(self.frequency)
Self.r.set_call(self.mycall)
Self.r.set_location(self.location)
Self.r.set_call(self.mycall)
Self.r.set_location(self.location)
Print("Frequency:", self.frequency)
Print("Baud:", self.baud)
Print("Beacon?:", self.is_beacon)
Print("Directed?:", self.is_directed)
Print("To callsign:", self.tocall)
Def get_radio():
Return dss
Def set_message(self, msg):
Self.message = msg.format(self.mycall, self.location)
Self.all_done = False
If self.is_beacon:
Self.r.set_message("\r\n\r\n{}:{}{}\r  \b  ".format(self.mycall, self.crc(self.mycall),
self.message))
Else:
Self.r.set_message("{}:{}{}\r  \b  ".format(self.mycall, self.crc(self.mycall), self.message))
Def send_code(self):
Self.dds.on()
Sleep(.1)
Self.r.send_code()
Def send_tone(self, tone):
Self.incremental_tone = (self.incremental_tone + float(tone) + 1.0) % 33
Self.f = int(int(self.frequency) + self.usb_offset + self.incremental_tone * self.spacing)
Self.dds.set_freq(0, self.f)
Self.dds.send()
Def all_done(self):
If self.is_beacon:
Self.r.stop()            # stop sending bits
Self.dds.off()
Sleep(float(self.beacon_interval))
Self.dds.on()
Self.r.send_code()        # Repeat for a beacon
Else:
Self.r.stop()            # stop sending bits
Self.dds.off()
Self.all_done = True
Def crc(self, text):
Self.table = []
For x in range(256):
Byte_val = x
For y in range(8):
If byte_val & 0x80:
```

```
Byte_val = (byte_val * 2) ^ 7
Else:
Byte_val = (byte_val * 2) ^ 0
Self.table.append(byte_val & 0xFF)
Val = 0
For ch in text:
Val = self.table[val ^ ord(ch)] & 0xFF
Return “%0.2X” % (val & 0xFF)
```

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

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

```
From machine import Timer
From fsq_varicode import fsq_varicode
Class FSQ:
Def __init__(self, baud, send_tone, report_message_end=None):
Self.send_tone = send_tone
Self.report_message_end = report_message_end
Self.set_baud(baud)
Self.frequency = “7104000”
Self.call = “N0CALL”
Self.location = “CM87xe”
Self.message = “{} {}   “
Self.baud = baud
Self.bit_length = int(1000 / float(baud))
Self.timer = Timer()
Self.all_done = False
Def set_call(self, call):
Self.call = call
Def set_baud(self, baud):
Self.baud = baud
Self.bit_length = int(1000 / float(self.baud))
Def set_frequency(self, frequency):
Self.frequency = frequency
Def set_location(self, location):
Self.location = location
Def set_message(self, message):
Self.message = message.format(self.call, self.location)
Def bit(self):
For letter in self.message:
Code = fsq_varicode.get(letter)
If not code:
Code = fsq_varicode.get(” “)                # Make illegal characters send as spaces
Count = 0
For tone in code:
If tone > 0 or count == 0:
Yield tone
Count += 1
Self.all_done = True
Def stop(self):
```

```
Self.timer.deinit()
Self.all_done = True
Def send_code(self):
Self.all_done = False
Self.gen = self.bit()
Self.timer.init(period=self.bit_length, mode=Timer.PERIODIC, callback=self.bit_finished)
Def send_bit(self, unused):
Try:
Tone = next(self.gen)
Except StopIteration as tone:
Self.all_done = True
Self.stop()
Self.report_message_end()
Return
Self.send_tone(tone)
Def bit_finished(self, unused):
Self.send_bit(True)
```

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

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

The main.py module is even simpler:

```
From fsq_config import FSQConfig
From time import sleep
# FLDIGI knows these baud rates: 1.5, 2, 3, 4.5, 6
Def main():
Fsq = FSQConfig(7040000, 12, “AB6NY”, “CM87xe”)
While True:
Fsq.set_message(”{} Testing from {} using a Raspberry Pi Pico RP2040”)
Fsq.send_code()
While fsq.all_done == False:
Sleep(5)
Main()
```

It should be self-explanatory. The radio.py and SI5351.py modules are the same as the ones we used for MFSK.

Fsq Mode

Python Programming