Python Radio 33: FM — No Static At All

Simon Quellen Field

Simon Quellen Field

Follow

9 min read

·

Jan 26, 2025

Listen

Share

More

Building a wide-band radio transmitter

Press enter or click to view image in full size

Screenshot of FM transmitter signal

Screenshot by the author

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

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

The man was Edwin Howard Armstrong.

Removing Static

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

When an AM signal is over-amplified, the sound in the receiver suffers distortion. The tops and bottoms of the sine wave are “clipped” and look flattened. Amplified enough, all that remains is a square wave.

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

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

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

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

Wide Band FM

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

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

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

Building Our Transmitter

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

But with a $2 Wemos D1 Mini and a $3 CJMCU-4713 FM transmitter breakout board, we can have an FM radio station for five bucks. And it transmits farther than the Raspberry Pi software transmitter. Ours will send high-fidelity signals about 300 feet, more than enough to cover the house and driveway, and possibly a neighbor or three.

Press enter or click to view image in full size

Our FM Transmitter

Photo by author

Six wires is all it takes.

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

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

FM GND → D1 G

FM 3vo → D1 3v3

FM RST → D1 D4

FM SCL → D1 D1

FM SDA → D1 D2

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

The Software

As usual here, the hardware is the easy part.

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

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

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

# See <https://cdn-shop.adafruit.com/datasheets/SiLabs%20Programming%20guide%20AN332.pdf>

From time import sleep

From array import array

Class xmit:

I2C\_ADDRESS = 0x63

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

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

11270, 11271)

DEFAULT\_REGISTERS\_VALUES = ((1, 199), (257, 0), (259, 0), (513, 32768), (514, 1), (8448, 3), (8449, 6625),

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

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

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

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

FREQ\_UNIT = int(10e3)

FREQ\_MIN = int(76e6)

FREQ\_MAX = int(108e6)

FREQ\_DEFAULT = 88.8e6

POWER\_DEFAULT = 115

MAX\_LINE\_INPUT\_LEVELS\_mV\_pk = {0: 190, 1: 301, 2: 416, 3: 636}

Def \_\_init\_\_(self, bus, pin\_reset, i2c\_address = I2C\_ADDRESS,

Freq = FREQ\_DEFAULT, tx\_power = POWER\_DEFAULT, stereo = True):

Self.\_bus = bus

Self.\_i2c\_address = i2c\_address

Self.\_pin\_reset = pin\_reset

Self.init()

Self.set\_frequency(freq)

Self.set\_power(tx\_power)

Self.stereo = stereo

Def init(self):

Self.power\_up()

Self.write\_all\_registers(self.DEFAULT\_REGISTERS\_VALUES)

Self.set\_frequency(self.FREQ\_DEFAULT)

Self.set\_power(self.POWER\_DEFAULT)

Def reset(self):

Self.init()

Def power\_up(self, analog\_audio\_inputs = True):

Self.\_assert\_reset()

Self.\_write\_bytes(array(‘B’, [0x01, 0x12, 0x50 if analog\_audio\_inputs else 0x0F]))

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

Def power\_down(self):

Self.\_write\_bytes(array(‘B’, [0x11]))

Def \_assert\_reset(self):

Self.\_pin\_reset.on()

Sleep(0.01)

Self.\_pin\_reset.off()

Sleep(0.01)

Self.\_pin\_reset.on()

@property

Def frequency(self):

Self.\_write\_bytes(array(‘B’, [0x33, 1]))

Bytes\_array = self.\_read\_bytes(8)

Freq = bytes\_array[2] << 8 | bytes\_array[3]

Self.\_frequency = freq \* 10e3

Return self.\_frequency

Def set\_frequency(self, freq):

Assert self.FREQ\_MIN <= freq <= self.FREQ\_MAX

Assert (freq // 1e3) % 50 == 0

Self.\_frequency = freq

Freq = round(freq // self.FREQ\_UNIT)

Self.\_write\_bytes(array(‘B’, [0x30, 0x00, freq >> 8 & 0xFF, freq & 0xFF]))

Sleep(0.2) # need 100ms

@property

Def tx\_power(self):

Self.\_write\_bytes(array(‘B’, [0x33, 1]))

Bytes\_array = self.\_read\_bytes(8)

Self.\_tx\_power = bytes\_array[5]

Return self.\_tx\_power

Def set\_power(self, power):

Assert power == 0 or (88 <= power <= 115)

Self.\_tx\_power = round(power)

Self.\_write\_bytes(array(‘B’, [0x31, 0, 0, self.\_tx\_power, 0]))

Sleep(0.2) # need 100ms

Def mute(self, value = True):

If value:

Self.\_write\_bytes(array(‘B’, [0x31, 0, 0, 0, 0]))

Else:

Self.set\_power(self.\_tx\_power)

Def mute\_line\_input(self, value = True):

Self.write\_register(0x2105, 0x03 if value else 0x00)

Def set\_line\_input\_level(self, attenuation\_level = 3, line\_level = None):

Line\_level = self.MAX\_LINE\_INPUT\_LEVELS\_mV\_pk[attenuation\_level] if line\_level is None else line\_level

Self.write\_register(0x2104, ((attenuation\_level & 0x03) << 12) | (line\_level & 0x3FF))

@property

Def stereo(self):

Return (self.read\_register(0x2100) & 0x03) == 0x03

@stereo.setter

Def stereo(self, value = True):

Current\_value = self.read\_register(0x2100)

Self.write\_register(0x2100, current\_value & ~3 | (3 if value else 0))

Sleep(0.01) # status: 0x84

Def enable(self, value = True):

Self.mute(not value)

# =================== data access ======================

Def \_get\_element\_value(self, reg\_address, idx\_lowest\_bit, n\_bits):

Reg\_value = self.read\_register(reg\_address)

Mask = 2 \*\* n\_bits – 1

Return reg\_value >> idx\_lowest\_bit & mask

Def \_set\_element\_value(self, reg\_address, idx\_lowest\_bit, n\_bits, element\_value, ):

Reg\_value = self.read\_register(reg\_address)

Mask = 2 \*\* n\_bits – 1

Return reg\_value & ~(mask << idx\_lowest\_bit) | ((element\_value & mask) << idx\_lowest\_bit)

# =================== data access ======================

Def \_read\_bytes(self, n\_bytes):

Return self.\_bus.readfrom(self.\_i2c\_address, n\_bytes)

Def \_write\_bytes(self, bytes\_array):

Self.\_bus.writeto(self.\_i2c\_address, bytes\_array)

Sleep(0.01) # wait for CTS

Return self.\_read\_bytes(1)[0]

Def \_set\_property(self, address, value):

Self.\_write\_bytes(array(‘B’, [0x12, 0,

Address >> 8 & 0xFF, address & 0xFF,

Value >> 8 & 0xFF, value & 0xFF]))

Sleep(0.01) # set\_property takes 10ms

Def \_get\_property(self, address):

Self.\_write\_bytes(array(‘B’, [0x13, 0, address >> 8 & 0xFF, address & 0xFF]))

Bytes\_array = self.\_read\_bytes(4)

Return bytes\_array[2] << 8 | bytes\_array[3]

Def read\_register(self, reg\_address):

Return self.\_get\_property(reg\_address)

Def write\_register(self, reg\_address, value):

Return self.\_set\_property(reg\_address, value)

Def read\_all\_registers(self):

Addressed\_values = []

For address in self.REGISTERS\_ADDRESSES:

Try:

Value = self.read\_register(address)

Addressed\_values.append((address, value))

Except:

Pass

Return addressed\_values

Def write\_all\_registers(self, addressed\_values):

For (address, value) in addressed\_values:

Try:

Self.write\_register(address, value)

Except:

Pass

Def status(self, status):

S = “”

If status & 0x80:

S += “CTS “

If status & 0x40:

S += “ERR “

If status & 4:

S += “RDSINT “

If status & 2:

S += “ASQINT “

If status & 1:

S += “STCINT “

Return s

Def set\_ps(self, ps\_id, a1, a2, a3, a4):

Status = self.\_write\_bytes(array(‘B’, [0x36, ps\_id, a1, a2, a3, a4]))

Print(“Status is”, hex(status), self.status(status))

Def set\_ps\_string(self, s):

For x in range(4-len(s)%4):

S = s + “ “

For x in range(len(s) / 4):

Self.set\_ps(x, s[0], s[1], s[2], s[3])

S = s[4:]

Def set\_rds\_buff(self, location, a1, a2, a3, a4):

If location == 0:

First = 6

Else:

First = 4

Status = self.\_write\_bytes(array(‘B’, [0x35, first, 0x20, location, a1, a2, a3, a4]))

Print(“Status is”, hex(status), self.status(status))

Def set\_rds\_buff\_string(self, s):

For x in range(4-len(s)%4):

S = s + “ “

For x in range(len(s) / 4):

Self.set\_rds\_buff(x, s[0], s[1], s[2], s[3])

S = s[4:]

Def enable\_rds(self):

Self.write\_register(0x2100, 7)

Self.write\_register(0x2103, 200)

Self.write\_register(0x2C01, 0x40A7)

Self.write\_register(0x2C02, 3)

Self.write\_register(0x2C03, 0x1008)

Self.write\_register(0x2C04, 3)

Self.write\_register(0x2C05, 1)

Self.write\_register(0x2C07, 8)

From machine import Pin, I2C

D1 = Pin(5)

D2 = Pin(4)

D4 = Pin(2)

Def main():

I2c = I2C( scl=D1, sda=D2, freq=100\_000)

X = xmit(i2c, D4)

x.enable()

x.set\_frequency(90\_100\_000)

x.set\_power(115)

x.set\_line\_input\_level(3, None)

x.enable\_rds()

x.set\_ps\_string(b”Birdfarm”)

x.set\_rds\_buff\_string(b”This is a test of the RDS system.”)

main()

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

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

The Bad News

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

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

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

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

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

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

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

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

Receiving RDS data

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

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

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

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

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

Python

Radio

Radio Hackers

Fm Transmitter

Micropython