Python Radio 34: Immaculate Reception

Simon Quellen Field

Simon Quellen Field

Follow

19 min read

·

Feb 5, 2025

Listen

Share

More

FM Radio with Digital Data

Press enter or click to view image in full size

Immaculate Reception

MidJourney

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

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

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

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

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

Press enter or click to view image in full size

The hardware for the FM receiver

Photo by the author.

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

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

D1 Mini Ground → Ground

D1 Mini pin D2 → SDIO

D1 Mini pin D1 → SCLK

D1 Mini pin D3 → RST

The breakout board has a 1/8-inch (3.5 mm) phone jack for earphones. You can use the earphones as the antenna, or (as I did) you can attach a wire to the earphone sleeve and another to the D1 Mini’s ground, and attach those to an external FM antenna. This helps for weak stations.

The hardware goes together quickly. The software took a lot longer to create. But all you need to do is copy and paste.

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

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

The main.py file is still the largest and is mostly the driver for the Si4703:

From time import sleep

Class xmit:

I2C\_ADDRESS = 0x10

# The Si4703 reads the registers starting at 0xA and wraps around to 0x0 at 0xF

# This enables code to save time by reading only a few status registers and ignore the control registers.

DEVICEID = 0x00 # Never changes

CHIPID = 0x01 # Never changes

POWERCFG = 0x02 # Control registers start here

CHANNEL = 0x03

SYSCONFIG1 = 0x04

SYSCONFIG2 = 0x05

SYSCONFIG3 = 0x06

OSCILLATOR = 0x07

# 8 and 9 are reserved

STATUSRSSI = 0x0A # Status registers start here

READCHAN = 0x0B

RDSA = 0x0C

RDSB = 0x0D

RDSC = 0x0E

RDSD = 0x0F

START\_OSC = 0x8100

VOLUME\_LOW = 0

EXTEND\_RANGE = 0x100

ENABLE\_UNMUTE = 0x4001

FREQ\_DEFAULT = 88\_500\_000

# Register 0x02 – POWERCFG

SMUTE = (1 << 15)

DMUTE = (1 << 14)

MONO = (1 << 13)

# bit 12 is zero

RDSM = (1 << 11)

SEEK\_MODE = (1 << 10)

SEEKUP = (1 << 9)

SEEK = (1 << 8)

# bit 7 is zero

DISABLE = (1 << 6)

SEEKDN = (1 << 1)

ENABLE = (1 << 0)

# Register 0x03 – CHANNEL

TUNE = (1 << 15)

# Low 9 bits are the channel

# Register 0x04 – SYSCONFIG1

RDSIEN = (1 << 15)

STCIEN = (1 << 14)

# Bit 13 is zero

ENABLE\_RDS = (1 << 12)

DE = (1 << 11)

AGCD = (1 << 10)

# Bits 9 and 8 are zero

# The four below are all 2 bits wide

BLNDADJ = (1 << 6)

GPIO3 = (1 << 4)

GPIO2 = (1 << 2)

GPIO1 = (1 << 0)

# Register 0x05 – SYSCONFIG2

SEEKTH = 8 # The top 8 bits

BAND = 6 # The next 2 bits

SPACE1 = 5

SPACE0 = 4

VOLUME\_MASK = 0x000F # The bottom 4 bits

# Register 0x06 – SYSCONFIG3

SMUTER = 14 # 2 bits

SMUTEA = 12 # 2 bits

# Bits 11 and 10 are unassigned

RDSPRF = 9 # RDS performance bit

VOLEXT = 8 # Extended volume range

SEEK\_SNR = 4 # 4 bits wide

SEEK\_CNT = 0 # 4 bits wide

# Register 0x07 – TEST1

AHIZEN = (1 << 14)

XOSCEN = (1 << 15)

# Register 0x0A – STATUSRSSI

RDSR = (1 << 15)

STC = (1 << 14)

TUNING\_READY = (1 << 14)

SFBL = (1 << 13)

AFCRL = (1 << 12)

RDSS = (1 << 11)

BLERA = (1 << 9) # 2 bits wide

STEREO = (1 << 8)

RSSI = (1 << 0) # Bottom 8 bits

# Register 0x0B – READCHAN

BLERB = (1 << 14) # 2 bits wide

BLERC = (1 << 12) # 2 bits wide

BLERD = (1 << 10) # 2 bits wide

READCHANBITS = (1 << 9) # Low 9 bits

# RDS Variables

# Register RDSB

GROUPTYPE = 11

GT\_MASK = 0x1F

TP = 10

TA = 4

MS = 3

TYPE0\_MASK = 0x0003

TYPE2\_MASK = 0x000F

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

From machine import RTC, Pin

Self.LED = Pin(2)

Self.LED.off()

Self.\_bus = bus

Self.\_i2c\_address = i2c\_address

Self.\_pin\_reset = pin\_reset

Self.regs = [0] \* 16

Self.chan = 5

Self.threshold = 25

Self.rtc = RTC()

Self.rds\_list = []

Self.flush\_RDS()

Self.\_assert\_reset()

Self.read\_all\_registers()

Self.regs[self.OSCILLATOR] = self.START\_OSC

Self.write\_all\_registers()

Sleep(1)

Self.read\_all\_registers()

Self.regs[self.POWERCFG] = self.ENABLE\_UNMUTE

Self.regs[self.SYSCONFIG1] |= self.ENABLE\_RDS

Self.regs[self.SYSCONFIG2] |= self.VOLUME\_LOW

Self.regs[self.SYSCONFIG2] |= (0x19 << self.SEEKTH)

Self.regs[self.SYSCONFIG2] &= 0xFFF0 # Clear volume bits

Self.regs[self.SYSCONFIG2] |= 1 # Lowest volume

# self.regs[self.SYSCONFIG3] |= self.EXTEND\_RANGE

# self.regs[self.SYSCONFIG3] |= (0x4 << self.SEEK\_SNR)

# self.regs[self.SYSCONFIG3] |= (0x8 << self.SEEK\_CNT)

Self.write\_all\_registers()

Sleep(0.2)

Def flush\_RDS(self):

Self.pi = 0

Self.ps = bytearray(‘ ‘ \* 8, ‘utf-8’)

Self.rt = bytearray(‘ ‘ \* 64, ‘utf-8’)

Self.rt\_index = 0

Self.ps\_buf = [0] \* 4

Self.ps\_cnt = 0

Self.rt\_buf = [0] \* 32

Self.rt\_cnt = 0

Self.old\_rt\_index = 0

Self.debug\_str = “”

Self.program\_service = “”

Self.RDS\_text = “”

Self.other = “”

Self.time\_str = “”

Self.has\_time = False

Def \_assert\_reset(self):

Self.\_pin\_reset.on()

Sleep(0.01)

Self.\_pin\_reset.off()

Sleep(0.01)

Self.\_pin\_reset.on()

Def has\_RDS(self):

Self.read\_all\_registers()

If self.regs[self.STATUSRSSI] & self.RDSR: # We received some RDS data

Return True

Return False

Def get\_group\_type(self, gt):

From group\_types import group\_types

Type\_str = group\_types[gt]

Return type\_str

Def get\_program\_type(self, pt):

From type\_strings import rbds\_types # Change this to rds\_types if not in North America

Type\_str = rbds\_types[pt]

Return type\_str

Def read\_RDS(self):

Self.read\_all\_registers()

Self.new\_data = False

Self.old\_ps\_AB = “”

Self.old\_rt\_AB = “”

If self.regs[self.STATUSRSSI] & self.RDSR: # We received some RDS data

From read\_RDS import read

Read(self)

Return self.new\_data

Def clear\_RDS(self):

Self.ps = bytearray(‘ ‘ \* 10, ‘utf-8’)

Self.rt = bytearray(‘ ‘ \* 65, ‘utf-8’)

Def RDS\_performance(self, yes\_no):

If yes\_no:

Self.bit\_set(self.SYSCONFIG3, self.RDSPRF)

Else:

Self.bit\_clr(self.SYSCONFIG3, self.RDSPRF)

Def frequency(self):

Self.chan = (self.read\_register(self.READCHAN) & 0x3FF)

Freq = self.chan \* 200\_000 + 87\_500\_000

Return freq

Def get\_volume(self):

Return self.read\_register(self.SYSCONFIG2)

Def set\_volume(self, v):

Self.read\_all\_registers()

Self.regs[self.SYSCONFIG2] &= 0xFFF0

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

Self.write\_all\_registers()

Sleep(0.2)

Def set\_frequency(self, freq):

# Channels in the U.S. start at 87.5 MHz and have 0.2 MHz spacing

Self.flush\_RDS()

Chan = int((freq – 87\_500\_000) / 200\_000)

Self.read\_all\_registers()

Self.regs[self.CHANNEL] &= 0xFE00 # Clear channel bits

Self.regs[self.CHANNEL] |= chan | self.TUNE

Self.write\_all\_registers()

Sleep(0.2)

While self.read\_register(self.STATUSRSSI) & self.TUNING\_READY == 0:

Pass

Self.stereo = self.regs[self.STATUSRSSI] & self.STEREO != 0

Self.regs[self.CHANNEL] &= ~self.TUNE # Clear the tune bit

Self.write\_all\_registers()

Sleep(0.2)

Def seek(self, direction, first\_time = True):

Self.flush\_RDS()

Self.read\_all\_registers()

Self.regs[self.POWERCFG] |= self.SEEK\_MODE

If direction == True:

Self.regs[self.POWERCFG] &= ~self.SEEKDN

Self.regs[self.POWERCFG] |= self.SEEKUP

Else:

Self.regs[self.POWERCFG] &= ~self.SEEKUP

Self.regs[self.POWERCFG] |= self.SEEKDN

Self.regs[self.POWERCFG] |= self.SEEK

Self.write\_all\_registers()

While self.read\_register(self.STATUSRSSI) & self.TUNING\_READY == 0:

Pass

Self.stereo = self.regs[self.STATUSRSSI] & self.STEREO != 0

Sfbl = self.regs[self.STATUSRSSI] & self.SFBL

If sfbl: # We have reached the band limit – double check RSSI

Rssi = self.regs[self.STATUSRSSI] & 0xFF

If rssi < self.threshold and first\_time == True:

If direction == True:

Self.set\_frequency(self.frequency() + 200\_000)

Else:

Self.set\_frequency(self.frequency() – 200\_000)

Return self.seek(direction, False)

Self.regs[self.POWERCFG] &= ~self.SEEK

Self.regs[self.POWERCFG] &= ~self.SEEK\_MODE

Self.write\_all\_registers()

Def show\_seek\_config(self):

Self.read\_all\_registers()

Th = (self.regs[self.SYSCONFIG2] >> 8) & 0xFF

Snr = (self.regs[self.SYSCONFIG3] >> 4) & 0xF

Cnt = self.regs[self.SYSCONFIG3] & 0xF

Snrdb = self.regs[self.SYSCONFIG3] & 0xF

Print(th, snr, cnt)

Def seek\_config(self, threshold, snr, cnt):

Self.threshold = threshold

Self.read\_all\_registers()

Self.regs[self.SYSCONFIG2] &= ~0xFF00

Self.regs[self.SYSCONFIG2] |= (threshold & 0xFF) << 8

Self.regs[self.SYSCONFIG3] &= ~0xFF

Self.regs[self.SYSCONFIG3] |= (snr & 0xF) << 4

Self.regs[self.SYSCONFIG3] |= cnt & 0xF

Self.write\_all\_registers()

Def rssi(self):

Self.stereo = self.regs[self.STATUSRSSI] & self.STEREO != 0

Return self.read\_register(self.STATUSRSSI) & 0xFF

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

Self.read\_all\_registers()

Return self.regs[reg\_address]

Def bit\_clr(self, reg\_address, bits):

Self.read\_all\_registers()

Self.regs[reg\_address] &= ~bits

Self.write\_all\_registers()

Def bit\_set(self, reg\_address, bits):

Self.read\_all\_registers()

Self.regs[reg\_address] |= bits

Self.write\_all\_registers()

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

Self.regs[reg\_address] = value

Self.write\_all\_registers()

Def write\_all\_registers(self):

# Writing to the Si4703 starts at register 2 (since the other registers are read-only?)

# So we only need a list that holds 2 through 7: 6 words or 12 bytes

Buf = [0] \* 12

For i in range(0, 6):

Buf[i\*2], buf[(i\*2)+1] = divmod(self.regs[i+2], 0x100)

Self.\_bus.writeto(self.\_i2c\_address, bytearray(buf, ‘utf-8’))

Def read\_all\_registers(self):

R = self.\_bus.readfrom(self.\_i2c\_address, 32)

I = 0

While i < len(r):

Ind = int(((i/2) + 10) % 16)

Self.regs[ind] = (r[i] << 8) | r[i+1]

I += 2

Def dump\_registers(self):

Print(“DVID CHIP PWRC CHAN SYS1 SYS2 SYS3 OSC RSSI RCHN RDSA RDSB RDSC RDSD”)

For x in range(16):

Print(“{:04X}”.format(self.regs[x]), end=” “)

Print()

Def histogram(self):

From histogram import histogram

Histogram(self)

From machine import Pin, I2C

D1 = Pin(5)

D2 = Pin(4)

D3 = Pin(0)

Def show\_RDS(x, how\_many):

x.RDS\_performance(True)

old\_s = “”

for cnt in range(how\_many):

sleep(0.04) # Records come in 11.4 times per second

if x.read\_RDS():

r = x.rssi()

s = x.program\_service

s += “ {:3.1f} – {:04x} – {:s} {:s} {:3d} “.format(x.frequency() / 1\_000\_000, x.pi, x.ms, x.program\_type, r)

s += x.RDS\_text

s += “ [“ + x.other + “]”

# s += “ [“ + x.time\_str + “]”

If x.has\_time:

T = x.rtc.datetime()

Yr = t[0]

Mo = t[1]

Dy = t[2]

Wd = t[3]

Hr = t[4]

Mn = t[5]

Sc = t[6]

S += “ [{:2d}/{:02d}/{:02d} {:02d}:{:02d}:{:02d}]”.format(mo, dy, yr, hr, mn, sc)

# s += x.debug\_str # Remove this if you don’t want debug info

S += ‘ \r’ # The <CR> prevents scrolling. Lines write over old lines.

If s != old\_s:

Print(s, end=””)

# print() # Remove this if you don’t want scrolling

Old\_s = s

x.RDS\_performance(False)

def demo(x, times, slp):

from tune\_demo import tune\_demo

tune\_demo(x, times, slp)

def chip\_dump(x):

from chip\_info import show\_data

show\_data(x)

def main():

i2c = I2C(scl=D1, sda=D2, freq=100\_000)

x = xmit(i2c, D3)

sleep(1)

chip\_dump(x)

x.set\_frequency(98\_500\_000)

x.show\_seek\_config()

x.dump\_registers()

x.set\_volume(7)

sleep(5)

# x.histogram()

Print(“{:d} stations have RDS data”.format(len(x.rds\_list)))

For f in x.rds\_list:

x.set\_frequency(f)

show\_RDS(x, 10\_000)

x.set\_frequency(98\_500\_000)

show\_RDS(x, 10\_000)

x.set\_frequency(104\_900\_000)

show\_RDS(x, 10\_000)

x.set\_frequency(105\_700\_000)

show\_RDS(x, 10\_000)

x.set\_frequency(99\_100\_000)

show\_RDS(x, 10\_000)

x.set\_frequency(100\_300\_000)

show\_RDS(x, 10\_000)

demo(x, 10, 10)

main()

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

The \_\_init\_\_() method starts the receiver chip receiving using some default settings.

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

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

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

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

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

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

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

That’s it for the driver.

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

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

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

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

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

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

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

The chip\_info.py module looks like this:

Def show\_data(x):

x.read\_all\_registers()

print(“\033[2J”) # Clear the screen in case a reset printed garbage

dev\_id = x.regs[x.DEVICEID]

if dev\_id & 0xFFF == 0x242:

print(“Manufacturer: Skyworks Solutions”)

part\_family = (dev\_id >> 12) & 0xF

if part\_family == 1:

print(“Part family Si4702/3”)

chip\_id = int(x.regs[x.CHIPID])

rev = (chip\_id >> 10) & 0x3F

print(“Silicon revision:”, rev)

dev = (chip\_id >> 6) & 0xF

if dev == 0:

print(“Device: Si4700”)

if dev == 1:

print(“Device: Si4702”)

if dev == 8:

print(“Device: Si4701”)

if dev == 9:

print(“Device: Si4703”)

firmware = chip\_id & 0x3F

print(“Firmware revision:”, firmware)

# dev\_id was 0x1242

# chip\_id was 0x1253

# RRRR RRcc ccFF FFFF

# 0001 0010 0101 0011

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