Python Radio 29: Far Away BBS

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

Simon Quellen Field

Simon Quellen Field

Follow

19 min read

·

Oct 13, 2024

Listen

Share

More

Press enter or click to view image in full size

Long-distance communication using tiny radios.

MidJourney

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

The total cost to win the 1,000 mile-per-watt award is less than $10: $5 for a Raspberry Pi Pico, $1 for the radio, and a few inches of copper tape on a piece of cardboard.

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

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

Press enter or click to view image in full size

The Ra-01 transceiver module.

The Ra-01 transceiver module (photo by author)

Press enter or click to view image in full size

The RF96 transceiver module.

The RF96 transceiver module (photo by author)

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

Press enter or click to view image in full size

My hand holding two RF96 transceiver modules.

Photo by author

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

Press enter or click to view image in full size

Ra-01 connected to RP2040 on a breadboard.

Photo by author

The RF96 looks like this:

Press enter or click to view image in full size

RF96 and RP2040 on a breadboard.

Photo by author

The SX127x chip talks to the RP2040 using the SPI interface and 8 pins. Two are power and ground, leaving six for communication.

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

The other pins can be anything. I chose pin 15 for NSS (chip select), pin 14 for RESET, and pin 20 for DIO0 (Data Input/Output Zero).

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

Press enter or click to view image in full size

RP2040 Purple Board Pinout.

<https://tamanegi.digick.jp/computer-embedded/mcuboa/rp2040purple/#show>

The MicroPython SX127x driver is courtesy of Wei Lin. It did not support the RP2040, so I added that support. The driver files are config\_lora.py, controller.py, and sx127x.py, along with files defining board-specific details for several microprocessors: controller\_rpi.py, controller\_esp8266.py, controller\_esp32.py, and controller\_pc.py. I added controller\_rp2040.py, and a bit of code to config\_lora.py. Here are the files I added or modified:

The file config\_lora.py:

Import sys

Import os

Import time

IS\_PC = False

IS\_MICROPYTHON = (sys.implementation.name == ‘micropython’)

IS\_ESP8266 = (os.uname().sysname == ‘esp8266’)

IS\_ESP32 = (os.uname().sysname == ‘esp32’)

IS\_RP2040 = (os.uname().sysname == ‘rp2’)

IS\_TTGO\_LORA\_OLED = None

IS\_RPi = not (IS\_MICROPYTHON or IS\_PC)

Def mac2eui(mac):

Mac = mac[0:6] + ‘fffe’ + mac[6:]

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

If IS\_MICROPYTHON:

# Node Name

Import machine

Import ubinascii

Uuid = ubinascii.hexlify(machine.unique\_id()).decode()

If IS\_RP2040:

NODE\_NAME = ‘RP2040’

If IS\_ESP8266:

NODE\_NAME = ‘ESP8266\_’

If IS\_ESP32:

NODE\_NAME = ‘ESP32\_’

Import esp

IS\_TTGO\_LORA\_OLED = (esp.flash\_size() > 5000000)

NODE\_EUI = mac2eui(uuid)

NODE\_NAME = NODE\_NAME + uuid

# millisecond

Millisecond = time.ticks\_ms

# Controller

SOFT\_SPI = None

If IS\_TTGO\_LORA\_OLED:

From controller\_esp\_ttgo\_lora\_oled import Controller

SOFT\_SPI = True

Elif IS\_RP2040:

From controller\_rp2040 import Controller

Else:

From controller\_esp import Controller

If IS\_RPi:

# Node Name

Import socket

NODE\_NAME = ‘RPi\_’ + socket.gethostname()

# millisecond

Millisecond = lambda : time.time() \* 1000

# Controller

From controller\_rpi import Controller

If IS\_PC:

# Node Name

Import socket

NODE\_NAME = ‘PC\_’ + socket.gethostname()

# millisecond

Millisecond = lambda : time.time() \* 1000

# Controller

From controller\_pc import Controller

I only had to add a few lines here and there for the RP2040.

The controller\_rp2040.py is a simple port, copying most of the file from the other controller\_xxx.py files and changing a few lines to fit the RP2040:

From machine import Pin, SPI, reset

Import config\_lora

Import controller

Class Controller(controller.Controller):

# LoRa config

PIN\_ID\_FOR\_LORA\_RESET = 14

PIN\_ID\_FOR\_LORA\_SS = 15

PIN\_ID\_SCK = 18

PIN\_ID\_MOSI = 19

PIN\_ID\_MISO = 16

PIN\_ID\_FOR\_LORA\_DIO0 = 20

PIN\_ID\_FOR\_LORA\_DIO1 = None

PIN\_ID\_FOR\_LORA\_DIO2 = None

PIN\_ID\_FOR\_LORA\_DIO3 = None

PIN\_ID\_FOR\_LORA\_DIO4 = None

PIN\_ID\_FOR\_LORA\_DIO5 = None

If config\_lora.IS\_RP2040:

ON\_BOARD\_LED\_PIN\_NO = 25

ON\_BOARD\_LED\_HIGH\_IS\_ON = True

GPIO\_PINS = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,

10, 11, 12, 13, 14, 15, 16, 17, 18, 19,

20, 21, 22, 23, 24, 25)

Def \_\_init\_\_(self,

Pin\_id\_led = ON\_BOARD\_LED\_PIN\_NO,

On\_board\_led\_high\_is\_on = ON\_BOARD\_LED\_HIGH\_IS\_ON,

Pin\_id\_reset = PIN\_ID\_FOR\_LORA\_RESET,

Blink\_on\_start = (2, 0.5, 0.5)):

Super().\_\_init\_\_(pin\_id\_led,

On\_board\_led\_high\_is\_on,

Pin\_id\_reset,

Blink\_on\_start)

Def prepare\_pin(self, pin\_id, in\_out = Pin.OUT):

If pin\_id is not None:

Pin = Pin(pin\_id, in\_out)

New\_pin = Controller.Mock()

New\_pin.pin\_id = pin\_id

New\_pin.value = pin.value

If in\_out == Pin.OUT:

New\_pin.low = lambda : pin.value(0)

New\_pin.high = lambda : pin.value(1)

Else:

New\_pin.irq = pin.irq

Return new\_pin

Def prepare\_irq\_pin(self, pin\_id):

Pin = self.prepare\_pin(pin\_id, Pin.IN)

If pin:

Pin.set\_handler\_for\_irq\_on\_rising\_edge = lambda handler: pin.irq(handler = handler, trigger = Pin.IRQ\_RISING)

Pin.detach\_irq = lambda : pin.irq(handler = None, trigger = 0)

Return pin

Def get\_spi(self):

Spi = None

Id = 0

If config\_lora.IS\_RP2040:

Try:

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

Sck = Pin(self.PIN\_ID\_SCK, Pin.OUT, Pin.PULL\_DOWN),

Mosi = Pin(self.PIN\_ID\_MOSI, Pin.OUT, Pin.PULL\_UP),

Miso = Pin(self.PIN\_ID\_MISO, Pin.IN, Pin.PULL\_UP))

Spi.init()

Except Exception as e:

Print(e)

If spi:

Spi.deinit()

Spi = None

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

Return spi

Def prepare\_spi(self, spi):

If spi:

New\_spi = Controller.Mock()

Def transfer(pin\_ss, address, value = 0x00):

Response = bytearray(1)

Pin\_ss.low()

Spi.write(bytes([address]))

Spi.write\_readinto(bytes([value]), response)

Pin\_ss.high()

Return response

New\_spi.transfer = transfer

New\_spi.close = spi.deinit

Return new\_spi

Def \_\_exit\_\_(self):

Self.spi.close()

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

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

The controller.py file:

From time import sleep

Class Controller:

Class Mock:

Pass

ON\_BOARD\_LED\_PIN\_NO = None

ON\_BOARD\_LED\_HIGH\_IS\_ON = True

GPIO\_PINS = []

PIN\_ID\_FOR\_LORA\_RESET = None

PIN\_ID\_FOR\_LORA\_SS = None

PIN\_ID\_SCK = None

PIN\_ID\_MOSI = None

PIN\_ID\_MISO = None

PIN\_ID\_FOR\_LORA\_DIO0 = None

PIN\_ID\_FOR\_LORA\_DIO1 = None

PIN\_ID\_FOR\_LORA\_DIO2 = None

PIN\_ID\_FOR\_LORA\_DIO3 = None

PIN\_ID\_FOR\_LORA\_DIO4 = None

PIN\_ID\_FOR\_LORA\_DIO5 = None

Def \_\_init\_\_(self,

Pin\_id\_led = ON\_BOARD\_LED\_PIN\_NO,

On\_board\_led\_high\_is\_on = ON\_BOARD\_LED\_HIGH\_IS\_ON,

Pin\_id\_reset = PIN\_ID\_FOR\_LORA\_RESET,

Blink\_on\_start = (2, 0.5, 0.5)):

Self.pin\_led = self.prepare\_pin(pin\_id\_led)

Self.on\_board\_led\_high\_is\_on = on\_board\_led\_high\_is\_on

Self.pin\_reset = self.prepare\_pin(pin\_id\_reset)

Self.reset\_pin(self.pin\_reset)

Self.spi = self.prepare\_spi(self.get\_spi())

Self.transceivers = {}

Self.blink\_led(\*blink\_on\_start)

Def add\_transceiver(self,

Transceiver,

Pin\_id\_ss = PIN\_ID\_FOR\_LORA\_SS,

Pin\_id\_RxDone = PIN\_ID\_FOR\_LORA\_DIO0,

Pin\_id\_RxTimeout = PIN\_ID\_FOR\_LORA\_DIO1,

Pin\_id\_ValidHeader = PIN\_ID\_FOR\_LORA\_DIO2,

Pin\_id\_CadDone = PIN\_ID\_FOR\_LORA\_DIO3,

Pin\_id\_CadDetected = PIN\_ID\_FOR\_LORA\_DIO4,

Pin\_id\_PayloadCrcError = PIN\_ID\_FOR\_LORA\_DIO5):

Transceiver.transfer = self.spi.transfer

Transceiver.blink\_led = self.blink\_led

Transceiver.pin\_ss = self.prepare\_pin(pin\_id\_ss)

Transceiver.pin\_RxDone = self.prepare\_irq\_pin(pin\_id\_RxDone)

Transceiver.pin\_RxTimeout = self.prepare\_irq\_pin(pin\_id\_RxTimeout)

Transceiver.pin\_ValidHeader = self.prepare\_irq\_pin(pin\_id\_ValidHeader)

Transceiver.pin\_CadDone = self.prepare\_irq\_pin(pin\_id\_CadDone)

Transceiver.pin\_CadDetected = self.prepare\_irq\_pin(pin\_id\_CadDetected)

Transceiver.pin\_PayloadCrcError = self.prepare\_irq\_pin(pin\_id\_PayloadCrcError)

Transceiver.init()

Self.transceivers[transceiver.name] = transceiver

Return transceiver

Def prepare\_pin(self, pin\_id, in\_out = None):

Reason = ‘’’

# a pin should provide:

# .pin\_id

# .low()

# .high()

# .value() # read input.

# .irq() # (ESP8266/ESP32 only) ref to the irq function of real pin object.

‘’’

Raise NotImplementedError(reason)

Def prepare\_irq\_pin(self, pin\_id):

Reason = ‘’’

# a irq\_pin should provide:

# .set\_handler\_for\_irq\_on\_rising\_edge() # to set trigger and handler.

# .detach\_irq()

‘’’

Raise NotImplementedError(reason)

Def get\_spi(self):

Reason = ‘’’

# initialize SPI interface

‘’’

Raise NotImplementedError(reason)

Def prepare\_spi(self, spi):

Reason = ‘’’

# a spi should provide:

# .close()

# .transfer(pin\_ss, address, value = 0x00)

‘’’

Raise NotImplementedError(reason)

Def led\_on(self, on = True):

Self.pin\_led.high() if self.on\_board\_led\_high\_is\_on == on else self.pin\_led.low()

Def blink\_led(self, times = 1, on\_seconds = 0.1, off\_seconds = 0.1):

For i in range(times):

Self.led\_on(True)

Sleep(on\_seconds)

Self.led\_on(False)

Sleep(off\_seconds)

Def reset\_pin(self, pin, duration\_low = 0.05, duration\_high = 0.05):

Pin.low()

Sleep(duration\_low)

Pin.high()

Sleep(duration\_high)

Def \_\_exit\_\_(self):

Self.spi.close()

Finally, the sx127x.py file:

From time import sleep

Import gc

Import config\_lora

PA\_OUTPUT\_RFO\_PIN = 0

PA\_OUTPUT\_PA\_BOOST\_PIN = 1

# registers

REG\_FIFO = 0x00

REG\_OP\_MODE = 0x01

REG\_FRF\_MSB = 0x06

REG\_FRF\_MID = 0x07

REG\_FRF\_LSB = 0x08

REG\_PA\_CONFIG = 0x09

REG\_LNA = 0x0c

REG\_FIFO\_ADDR\_PTR = 0x0d

REG\_FIFO\_TX\_BASE\_ADDR = 0x0e

FifoTxBaseAddr = 0x00

# FifoTxBaseAddr = 0x80

REG\_FIFO\_RX\_BASE\_ADDR = 0x0f

FifoRxBaseAddr = 0x00

REG\_FIFO\_RX\_CURRENT\_ADDR = 0x10

REG\_IRQ\_FLAGS\_MASK = 0x11

REG\_IRQ\_FLAGS = 0x12

REG\_RX\_NB\_BYTES = 0x13

REG\_PKT\_RSSI\_VALUE = 0x1a

REG\_PKT\_SNR\_VALUE = 0x1b

REG\_MODEM\_CONFIG\_1 = 0x1d

REG\_MODEM\_CONFIG\_2 = 0x1e

REG\_PREAMBLE\_MSB = 0x20

REG\_PREAMBLE\_LSB = 0x21

REG\_PAYLOAD\_LENGTH = 0x22

REG\_FIFO\_RX\_BYTE\_ADDR = 0x25

REG\_MODEM\_CONFIG\_3 = 0x26

REG\_RSSI\_WIDEBAND = 0x2c

REG\_DETECTION\_OPTIMIZE = 0x31

REG\_DETECTION\_THRESHOLD = 0x37

REG\_SYNC\_WORD = 0x39

REG\_DIO\_MAPPING\_1 = 0x40

REG\_VERSION = 0x42

# modes

MODE\_LONG\_RANGE\_MODE = 0x80 # bit 7: 1 => LoRa mode

MODE\_SLEEP = 0x00

MODE\_STDBY = 0x01

MODE\_TX = 0x03

MODE\_RX\_CONTINUOUS = 0x05

MODE\_RX\_SINGLE = 0x06

# PA config

PA\_BOOST = 0x80

# IRQ masks

IRQ\_TX\_DONE\_MASK = 0x08

IRQ\_PAYLOAD\_CRC\_ERROR\_MASK = 0x20

IRQ\_RX\_DONE\_MASK = 0x40

IRQ\_RX\_TIME\_OUT\_MASK = 0x80

# Buffer size

MAX\_PKT\_LENGTH = 255

Class SX127x:

# The controller can be ESP8266, ESP32, Raspberry Pi, or a PC.

# The controller needs to provide an interface consisted of:

# 1. A SPI, with transfer function.

# 2. A reset pin, with low(), high() functions.

# 3. IRQ pinS , to be triggered by RFM96W’s DIO0~5 pins. These pins each has two functions:

# 3.1 set\_handler\_for\_irq\_on\_rising\_edge()

# 3.2 detach\_irq()

# 4. A function to blink on-board LED.

Def \_\_init\_\_(self,

Name = ‘SX127x’,

Parameters = {‘frequency’: 433E6, ‘tx\_power\_level’: 2, ‘signal\_bandwidth’: 125E3,

‘spreading\_factor’: 8, ‘coding\_rate’: 5, ‘preamble\_length’: 8,

‘implicitHeader’: False, ‘sync\_word’: 0x12, ‘enable\_CRC’: False},

# parameters = {‘frequency’: 433E6, ‘tx\_power\_level’: 2, ‘signal\_bandwidth’: 125E3,

# ‘spreading\_factor’: 8, ‘coding\_rate’: 5, ‘preamble\_length’: 8,

# ‘implicitHeader’: False, ‘sync\_word’: 0x12, ‘enable\_CRC’: False},

onReceive = None):

self.name = name

self.parameters = parameters

self.\_onReceive = onReceive

self.\_lock = False

def init(self, parameters = None):

if parameters: self.parameters = parameters

# check version

Version = self.readRegister(REG\_VERSION)

If version != 0x12:

Raise Exception(‘Invalid version.’)

# put in LoRa and sleep mode

Self.sleep()

# config

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

Self.setSignalBandwidth(self.parameters[‘signal\_bandwidth’])

# set LNA boost

Self.writeRegister(REG\_LNA, self.readRegister(REG\_LNA) | 0x03)

# set auto AGC

Self.writeRegister(REG\_MODEM\_CONFIG\_3, 0x04)

Self.setTxPower(self.parameters[‘tx\_power\_level’])

Self.\_implicitHeaderMode = None

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

Self.setSpreadingFactor(self.parameters[‘spreading\_factor’])

Self.setCodingRate(self.parameters[‘coding\_rate’])

Self.setPreambleLength(self.parameters[‘preamble\_length’])

Self.setSyncWord(self.parameters[‘sync\_word’])

Self.enableCRC(self.parameters[‘enable\_CRC’])

# set LowDataRateOptimize flag if symbol time > 16ms (default disable on reset)

# self.writeRegister(REG\_MODEM\_CONFIG\_3, self.readRegister(REG\_MODEM\_CONFIG\_3) & 0xF7) # default disable on reset

If 1000 / (self.parameters[‘signal\_bandwidth’] / 2\*\*self.parameters[‘spreading\_factor’]) > 16:

Self.writeRegister(REG\_MODEM\_CONFIG\_3, self.readRegister(REG\_MODEM\_CONFIG\_3) | 0x08)

# set base addresses

Self.writeRegister(REG\_FIFO\_TX\_BASE\_ADDR, FifoTxBaseAddr)

Self.writeRegister(REG\_FIFO\_RX\_BASE\_ADDR, FifoRxBaseAddr)

Self.standby()

Def beginPacket(self, implicitHeaderMode = False):

Self.standby()

Self.implicitHeaderMode(implicitHeaderMode)

# reset FIFO address and paload length

Self.writeRegister(REG\_FIFO\_ADDR\_PTR, FifoTxBaseAddr)

Self.writeRegister(REG\_PAYLOAD\_LENGTH, 0)

Def endPacket(self):

# put in TX mode

Self.writeRegister(REG\_OP\_MODE, MODE\_LONG\_RANGE\_MODE | MODE\_TX)

# wait for TX done, standby automatically on TX\_DONE

While (self.readRegister(REG\_IRQ\_FLAGS) & IRQ\_TX\_DONE\_MASK) == 0:

Pass

# clear IRQ’s

Self.writeRegister(REG\_IRQ\_FLAGS, IRQ\_TX\_DONE\_MASK)

Self.collect\_garbage()

Def write(self, buffer):

currentLength = self.readRegister(REG\_PAYLOAD\_LENGTH)

size = len(buffer)

# check size

Size = min(size, (MAX\_PKT\_LENGTH – FifoTxBaseAddr – currentLength))

# write data

For i in range(size):

Self.writeRegister(REG\_FIFO, buffer[i])

# update length

Self.writeRegister(REG\_PAYLOAD\_LENGTH, currentLength + size)

Return size

Def aquire\_lock(self, lock = False):

If not config\_lora.IS\_MICROPYTHON: # MicroPython is single threaded, doesn’t need lock.

If lock:

While self.\_lock: pass

Self.\_lock = True

Else:

Self.\_lock = False

Def println(self, string, implicitHeader = False):

Self.aquire\_lock(True) # wait until RX\_Done, lock and begin writing.

Self.beginPacket(implicitHeader)

Self.write(string.encode())

Self.endPacket()

Self.aquire\_lock(False) # unlock when done writing

Def getIrqFlags(self):

irqFlags = self.readRegister(REG\_IRQ\_FLAGS)

self.writeRegister(REG\_IRQ\_FLAGS, irqFlags)

return irqFlags

def packetRssi(self):

return (self.readRegister(REG\_PKT\_RSSI\_VALUE) – (164 if self.\_frequency < 868E6 else 157))

def packetSnr(self):

return (self.readRegister(REG\_PKT\_SNR\_VALUE)) \* 0.25

def standby(self):

self.writeRegister(REG\_OP\_MODE, MODE\_LONG\_RANGE\_MODE | MODE\_STDBY)

def sleep(self):

self.writeRegister(REG\_OP\_MODE, MODE\_LONG\_RANGE\_MODE | MODE\_SLEEP)

def setTxPower(self, level, outputPin = PA\_OUTPUT\_PA\_BOOST\_PIN):

if (outputPin == PA\_OUTPUT\_RFO\_PIN):

# RFO

Level = min(max(level, 0), 14)

Self.writeRegister(REG\_PA\_CONFIG, 0x70 | level)

Else:

# PA BOOST

Level = min(max(level, 2), 17)

Self.writeRegister(REG\_PA\_CONFIG, PA\_BOOST | (level – 2))

Def setFrequency(self, frequency):

Self.\_frequency = frequency

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

433E6: (108, 64, 0),

434E6: (108, 128, 0),

866E6: (216, 128, 0),

868E6: (217, 0, 0),

915E6: (228, 192, 0)}

Self.writeRegister(REG\_FRF\_MSB, frfs[frequency][0])

Self.writeRegister(REG\_FRF\_MID, frfs[frequency][1])

Self.writeRegister(REG\_FRF\_LSB, frfs[frequency][2])

Def setSpreadingFactor(self, sf):

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

Self.writeRegister(REG\_DETECTION\_OPTIMIZE, 0xc5 if sf == 6 else 0xc3)

Self.writeRegister(REG\_DETECTION\_THRESHOLD, 0x0c if sf == 6 else 0x0a)

Self.writeRegister(REG\_MODEM\_CONFIG\_2, (self.readRegister(REG\_MODEM\_CONFIG\_2) & 0x0f) | ((sf << 4) & 0xf0))

Def setSignalBandwidth(self, sbw):

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

Bw = 9

For i in range(len(bins)):

If sbw