Python Radio 26: Double Your Power

…and also build a full duplex repeater

Simon Quellen Field

Simon Quellen Field

Follow

8 min read

·

Sep 19, 2024

Listen

Share

More

Press enter or click to view image in full size

Two powerful robot twins

MidJourney

With this project, we will double the range of our communications and double the processing power of our computer at the same time.

Our HC-12 radio was half duplex. It could either transmit or receive, but not both at the same time. This project will fix that (by using two HC-12 modules. Hey, they’re cheap.)

The radios already have a range of over a kilometer. But we always want more. A repeater is a radio that can listen to our HC-12 on one channel, and repeat that data on another channel. We can place it between two locations that want to communicate but can’t, either because they are too far away, or because there is an obstruction in the way, such as a building, or in my case, a mountain.

To enable us to listen to one radio and transmit on the other at the same time, we will be using both processors on either the ESP32 or the RP2040. They both have two main processors, in addition to some lesser processors that we won’t be using here.

When using two processors at the same time, there are occasions when they need to communicate with one another. They can do this by sharing a bit of memory. To make the sharing happen properly, they must agree to take turns, so that one of them isn’t reading the memory while the other is not finished writing to it, or (worse) they aren’t both writing at the same time.

We do this by using a lock. MicroPython makes using both processors easy, and it makes locking memory very easy as well, as we will see.

To illustrate how to use both processors and a lock to protect their communication, we will start with a simple example. We will have one processor blink the onboard LED while the second processor reads the count of how many times the LED blinked. Here is the code:

From \_thread import start\_new\_thread, allocate\_lock

From time import sleep

From machine import Pin

LED = Pin(2, Pin.OUT)

Count = 0

Def blink(cps, lock):

Global count

Print(“Blink is running”)

While True:

LED.value(LED.value() ^ 1)

Sleep(1/cps)

With lock:

Count += 1

Lock = allocate\_lock()

Def main():

Global count, lock

While True:

With lock:

C = count

Print(“Main says blink count is”, c)

Sleep(3)

Start\_new\_thread(blink, (5,lock))

Main()

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

We import two methods from the \_thread module: start\_new\_thread(), and allocate\_lock().

The blink() routine will run on the second processor. It announces itself, and then loops, flashing the LED at the rate cps (cycles per second), and keeping a count of how many times it blinked.

We allocate a lock, and pass it as an argument to blink() when we start the thread on the second processor using the start\_new\_thread() method. The arguments to blink() are sent in a tuple which is the second argument to start\_new\_thread().

Then we call main(), which loops, getting the lock before reading the blink count and printing it.

Notice how easy it is to synchronize the two processors using the with lock: statement.

We are now ready to build our repeater.

Press enter or click to view image in full size

433 MHz repeater using the RP2040

Photo by the author

The photo shows the repeater using the RP2040. The radio in front is using UART 1 on pins 4 and 5. The radio in the back is using UART 0 on pins 16 and 17. I put two electrolytic capacitors from power to ground to reduce noise, but they may not be necessary (the user guide suggests using one).

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

From machine import UART, Pin

From time import sleep\_ms

Class HC12:

Def \_\_init\_\_(self, unum, tx, rx, setpin, baud):

Self.unum = unum

Self.tx = tx

Self.rx = rx

Self.setpin = setpin

Self.baud = baud

Self.response = ‘’

Self.set = Pin(self.setpin, Pin.OUT)

Self.uart = UART(self.unum, baudrate=9600, tx=Pin(self.tx), rx=Pin(self.rx), timeout=100, timeout\_char=20)

Self.long\_distance()

Def cleanup(self, data):

If data:

Return bytes(b for b in data if b < 128)

Return data

Def command(self, cmd):

Self.set(0)

Sleep\_ms(40)

If cmd == ‘’:

Self.uart.write(“AT”)

Else:

Self.uart.write(“AT+” + cmd)

Sleep\_ms(40)

Self.set(1)

Sleep\_ms(1000) # Give the module time to process the command and reply

Self.response = self.read()

Self.response = self.cleanup(self.response)

Try:

If self.response:

Try:

Decoded = self.response.decode(‘utf-8’)

Self.response = decoded

Except Exception as e:

Print(“Can’t decode “, self.response, “into utf-8”, e)

Except Exception as e:

Print(“Exception in HC12.command:”, e)

Print(“Response was:”, self.response)

Def long\_distance(self):

Self.command(“FU4”)

Sleep\_ms(100)

Self.uart.deinit()

Self.uart = UART(self.unum, baudrate=1200, tx=Pin(self.tx), rx=Pin(self.rx), timeout=100, timeout\_char=20)

Sleep\_ms(1000)

Def write(self, msg):

Cnt = self.uart.write(msg)

Self.uart.flush() # Wait for all writing to finish

Return cnt

Def read(self):

Return self.uart.read()

Def read\_str(self):

R = self.cleanup(self.uart.read())

If r:

Return r.decode(‘utf-8’)

Return “”

Def any(self):

Return self.uart.any()

Def status(self):

Self.command(“RX”)

Print(self.response)

The constructor for the HC12 class creates the UART object using arguments passed in for the pins and baud rate. It also sets the radio into long-distance mode (1200 baud).

The command() method handles setting and clearing the SET bit to issue the AT commands to the radio. The radio has a habit of sending characters like 0x80 and 0xFF which can’t be decoded into utf-8 for printing, so the cleanup() method is used to remove anything with the high bit set.

The rest of the code follows the previous HC-12 project and should be familiar or self-explanatory.

Our main.py module looks like this:

From \_thread import start\_new\_thread, allocate\_lock

From machine import Pin

From hc12 import HC12

LED = Pin(25, Pin.OUT)

From whoami import WhoAmI

W = WhoAmI()

Send = None

Run\_thread = True

Def transmit(lock):

Global send, run\_thread

Radio = HC12(1, 4, 5, 3, 1200)

Radio.long\_distance()

Radio.command(w.tx\_channel())

Radio.status()

Print(“Transmit is running”)

While run\_thread:

With lock:

If send:

Print(“Transmit sending: “ + send)

LED(0)

Radio.write(send)

LED(1)

Send = None

Class Ping:

Def \_\_init\_\_(self):

Self.radio = HC12(1, 4, 5, 3, 1200)

Self.radio.long\_distance()

Self.radio.command(w.tx\_channel())

Self.radio.status()

Self.count = 0

Def timeout(self, timer):

Self.radio.write(w.name() + “ sending “ + str(self.count))

Self.count += 1

Def main():

Global send, run\_thread, lock

Print(“I am:”, w.name())

Lock = allocate\_lock()

Radio = HC12(0, 16, 17, 15, 1200)

Radio.long\_distance()

Radio.command(w.rx\_channel()) # Not actually needed: channel 1 is the default

Radio.status()

If w.name() == “repeater”:

Start\_new\_thread(transmit, (lock,))

Tim = None

If w.name() == “rover1”:

From machine import Timer

Ping = Ping()

Tim = Timer(period=5000, mode=Timer.PERIODIC, callback=ping.timeout)

Try:

While True:

If radio.any():

Input\_str = radio.read\_str()

Print(“Main received: “ + input\_str)

With lock:

Send = input\_str

Except KeyboardInterrupt as e:

Print(“KeyboardInterrupt in main:”, e)

If tim:

Print(“Shutting down timer”)

Tim.deinit()

Run\_thread = False

Else:

Print(“No timer to shut down”)

# Give the board 5 seconds after a reset to allow us to replace main.py during debugging

From time import sleep

Sleep(5)

Main()

This code will be run on two identical boards. One board will be the repeater. The other board will be called “rover1” and it will act as the client radio sending and receiving from the repeater. The two roles are established by the whoami.py and whoami.cfg files which will be shown shortly.

Starting with main(), we see that it prints out its name, allocates a lock, and sets UART 0 to be on channel 1.

If its name is “repeater”, it starts the transmit() thread on the second processor.

If it is “rover1” it starts a timer to call a timeout every five seconds. This will send a message to the repeater, acting as the client radio.

The main() routine then loops, reading from UART 0, and putting anything it reads into the shared variable send so that the transmit() thread can rebroadcast it.

There are a couple of housekeeping details we need to worry about. If we want to shut things down by hitting control-C, we want to turn off the timer and stop the thread running on the other processor.

The second bit of housekeeping is only used for the debugging phase. If two processors are both running when we try to send new files to the RP2040, the upload hangs. So we delay by five seconds after a reset, so we can quickly start loading files.

The transmit() thread sets up UART 1 on the transmit channel (which we set in the whoami.cfg file). I use channel 4 to transmit and channel 1 to receive. The first four channels are legal to use without a license. The higher 96 channels require an amateur radio license but have the advantage that you can use an amplifier to get as much as 50 watts of output power.

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

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

You can power the repeater with a solar panel and a battery and place it in a waterproof box on a roof or the top of a mountain.

Instead of the Ping class sending automatically, you can have the radio read from the USB port and send the text it reads. This lets you use your laptop keyboard to send and receive.

Any number of rovers can use the repeater. It is up to them to play nice and take turns so the text is not intermingled. The standard way to do this is to send the word “over” when you are finished with a thought.

The radios have a connector for an external antenna if you want even more range. The repeater will want an omnidirectional antenna (the ground plane antenna we built earlier is a good choice).

The rovers can use directional antennas, such as a dish or a Yagi-Uda. You can build these yourself or purchase them (commercial antennas for the 70-centimeter band are easy to find). Of course, they can also use the ground plane antenna or just the tiny spring antenna.

As promised, the whoami.cfg file for rover1 looks like this:

{“name”:”rover1”,”tx\_channel”:”C001”,”rx\_channel”:”C004”}

And for the repeater it looks like this:

{“name”:”repeater”,”tx\_channel”:”C004”,”rx\_channel”:”C001”}

Note that the receive channel and the transmit channel are swapped between the two.

The whoami.py file looks like this:

Class WhoAmI:

Def \_\_init\_\_(self):

Self.me = {}

Try:

With open(“whoami.cfg”,”rb”) as f:

Line = f.read(1024)

From json import loads

Self.me = loads(line)

Except OSError as e:

Print(“Error reading whoami.cfg:”, e )

Def name(self):

If “name” in self.me:

Return self.me[“name”]

Return “Unknown”

Def tx\_channel(self):

If “tx\_channel” in self.me:

Return self.me[“tx\_channel”]

Return None

Def rx\_channel(self):

If “rx\_channel” in self.me:

Return self.me[“rx\_channel”]

Return None

Radio Hackers

Python

Radio