Python Radio 22: Wi-Fi

Simon Quellen Field

Simon Quellen Field

Follow

8 min read

·

Sep 9, 2024

Listen

Share

More

Run a web server on a tiny computer.

Press enter or click to view image in full size

MidJourney

Our ESP32 already has a radio in it. It runs at 2.4 GHz, and is used for Wifi and Bluetooth Low Energy (BLE for short).

We can get to the REPL over Wifi, which can be quite convenient if the ESP32 is at the top of a pole controlling a weather station, or atop a tower holding radio antennas.

To do this is fairly simple. We put these two lines in boot.py:

Import webrepl

Webrepl.start()

And then create a file called webrepl\_cfg.py that has a password in it:

PASS = ‘simon’

The ESP32 will then advertise itself so we can connect to it over Wifi, and enter <http://192.168.4.1> in a web browser.

We can make it even easier by having the ESP32 connect to our local Wifi:

From network import WLAN

Wlan = WLAN(mode=WLAN.STA)

Nets = wlan.scan()

For net in nets:

If net.ssid == ‘mywifi’:

Print(’Network found!’)

Wlan.connect(net.ssid, auth=(net.sec, ‘mywifikey’), timeout=5000)

While not wlan.isconnected():

Machine.idle() # save power while waiting

Print(’WLAN connection succeeded!’)

Break

Now we can simply enter its IP address on the local network into our browser, and save having to connect via Wifi to the ESP32 directly.

If you want the device to always have the same IP address (which makes it much easier to connect) you can put this code in boot.py:

Import machine

From network import WLAN

Wlan = WLAN() # get current object, without changing the mode

If machine.reset\_cause() != machine.SOFT\_RESET:

Wlan.init(WLAN.STA)

# configuration below MUST match your home router settings!!

Wlan.ifconfig(config=(’192.168.178.107’, ‘255.255.255.0’, ‘192.168.178.1’, ‘8.8.8.8’))

If not wlan.isconnected():

# change the line below to match your network ssid, security and password

Wlan.connect(’mywifi’, auth=(WLAN.WPA2, ‘mywifikey’), timeout=5000)

While not wlan.isconnected():

Machine.idle() # save power while waiting

Of course, the IP address can be read from a file on the ESP32, so that each device can share the same code, and only the configuration file has to be unique to each device.

Making the ESP32 present a web page with buttons that control things is a little more involved. But once we have a few helper classes on the machine, the actual part left to do becomes simple.

Here is the HTML of the web page we want to serve:

<html>

<head>

<title>

Simon’s LED

</title>

<script>

Function on()

{

Let xhttp = new XMLHttpRequest();

Xhttp.onreadystatechange = function(){ handle\_response(this, list) };

Try

{

Xhttp.open( “GET”, “/?on”, true );

Xhttp.send();

}

Catch( err )

{

Console.log( “Caught error executing /?on” );

}

}

Function off()

{

Let xhttp = new XMLHttpRequest();

Xhttp.onreadystatechange = function(){ handle\_response(this, list) };

Try

{

Xhttp.open( “GET”, “/?off”, true );

Xhttp.send();

}

Catch( err )

{

Console.log( “Caught error executing /?off” );

}

}

</script>

</head>

<body>

<h1>

Simon’s LED

</h1>

<p/>

<br/><button type=”button” onclick=”on();”>On

<br/><button type=”button” onclick=”off();”>Off

</body>

</html>

All it is going to do is present two buttons that will turn on or off the backlight on the TTGO Display. Or any LED connected to pin 4 on any ESP32.

When we connect to the ESP32’s IP address in our browser, we get this page:

Press enter or click to view image in full size

Image by the author

In the upper left corner is an icon. That is being served by the ESP32 because we put a file there called favicon.ico. Browsers look for this file to put in that corner of the page.

Our main.py module controls getting connected to the Wifi and setting up the web server, but critically, it also handles what the buttons do:

From uasyncio import get\_event\_loop

From async\_webserver import WebServer

From connect import Connect

From machine import Pin

Led = Pin(4, Pin.OUT)

Def callback(q):

Print(”Query is”, q)

If q == “on”:

Print(”Turning backlight On”)

Led(1)

Elif q == “off”:

Print(”Turning backlight Off”)

Led(0)

Return None

Def main():

Con = Connect(”MyLED”)

Con.verbose = True

Con.reconnect()

Con.verbose = True

Web = WebServer(callback, False)

Web.verbose = True

Def web\_task():

While True:

Await web.serve()

Loop = get\_event\_loop()

Loop.create\_task(web\_task())

Loop.run\_forever()

Loop.close()

Main()

The Webserver class takes a callback method as an argument. That method handles what to do with query strings (the part of the URL after a question mark).

Now we just need to look at the helper classes Connect and WebServer.

The Connect class sets up two IP addresses. The first one is 192.168.4.1, the address of the ESP32 when we connect to its Wifi SSID. When we look for a Wifi server to connect to, we will see one called “MyLED 10.90.20.173” (the IP address will be different on different local networks). We could connect to that, and then tell the browser to go to 192.168.4.1 to get the web page.

But, since we were kind enough to put the local network address in the name, we could instead avoid connecting to the ESP32’s Wifi, and instead just send our browser to the IP address in the name, using our normal local network. If your computer only has one Wifi port, this makes it much easier to go back and forth between the Internet and the ESP. You don’t have to keep switching Wifi servers.

The connect.py module looks like this:

#

# We have a list of SSID,password pairs in the file network.cfg.

# We add any open SSIDs we can see.

# Connect to the first one that works.

# If we have moved to a new location, the caller will call reconnect

# We also become a WiFi access point without a password (default IP 192,168.4.1)

#

Class Connect:

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

From network import WLAN, STA\_IF, AP\_IF, AUTH\_OPEN

Self.ssid = ”

Self.who = who

Self.who\_am\_i = who

Self.verbose = True

Self.network\_list = []

Self.sta = WLAN( STA\_IF )

Self.sta.active( True )

Self.ap = WLAN( AP\_IF )

Self.ap.active( True )

Self.ap.config( essid=self.who\_am\_i, authmode=AUTH\_OPEN )

# Read the config file and scan for SSIDs

Def read\_config\_file( self ):

From network import WLAN, STA\_IF

Self.known\_networks = []

Try:

F = open( “network.cfg” )

If f:

Text = f.readline()

While text:

Self.known\_networks.append( text.rstrip( “\r\n” ) )

Text = f.readline()

f.close()

except Exception as e:

print( “Read\_config\_file():”, e )

try:

self.network\_list = self.sta.scan()

except Exception as e:

print( “read\_config\_file():”, e )

self.sta.disconnect()

self.sta.active( False )

self.sta = WLAN( STA\_IF )

self.sta.active( True )

# Check to see if a line from our config file matches an SSID from our scan

Def is\_available( self, name\_comma\_password ):

Name, pasw = name\_comma\_password.split( “,” )

For net in self.network\_list:

Ssid = net[0]

Target = ssid.decode( “utf-8” )

If name == target:

Return True, name, pasw

Return False, ”, ”

# Try to connect to any of the networks we know about

Def do\_connect( self ):

From network import WLAN, STA\_IF, AUTH\_OPEN

From time import sleep

If self.verbose:

Print( “Connect” )

Try:

Self.sta = WLAN( STA\_IF )

Self.sta.active( True )

If not self.sta.isconnected():

Self.read\_config\_file()

For id in self.known\_networks:

Known, self.ssid, paswd = self.is\_available( id )

If known:

If self.verbose:

Print( “We know”, str( self.ssid ) )

Try:

Self.sta.connect( self.ssid, paswd )

Count = 0

While not self.sta.isconnected():

If count > 10:

Return False

Sleep( 1 )

Count += 1

# If we can’t reach the Internet, try the next SSID

Ip = self.sta.ifconfig()[0]

Self.who\_am\_i = self.who + “ “ + ip

Self.sta.config(dhcp\_hostname=self.who\_am\_i)

Self.ap.config( essid=self.who\_am\_i, authmode=AUTH\_OPEN )

Return True

Except Exception as e:

Print( “Error in do\_connect():”, e )

Pass

Else:

If self.verbose:

Print( “Already connected” )

Return True

Except Exception as e:

Print( “do\_connect():”, e )

Return False

Def keep\_trying( self ):

While not self.sta.isconnected():

Try:

Self.do\_connect()

Except Exception as e:

Print( “Error in keep\_trying():”, e )

Pass

If self.verbose:

Ip = self.sta.ifconfig()[0]

Print(”Connected to”, ip)

Def reconnect( self ):

If self.verbose:

Print( “Reconnecting” )

Self.sta.disconnect()

Self.sta.active( False )

Self.keep\_trying()

We can see the two WLAN ports, named AP\_IF for “Access Point Interface” and STA\_IF, for “Station Interface”. The access point is the wifi server, and the station is the connection to our local network.

We have a file called network.cfg that contains SSIDs and passwords for the Wifi access points we expect to be near:

BirdfarmOffice2,12345678

BirdfarmBalcony,

Birdfarm2TVRoom,

BirdfarmGym,

BirdfarmBarn,

BirdfarmGuest,

DLINK,12345678

Simon’s pocket,

NETGEAR28,12345678

Netgear28,12345678

Netgear28,12345678

PubNetPatio,6503632620

PubNetBar,6503632620

The Connect class goes through the list, checking if any of those SSIDs can connect. If they can, then it connects and returns.

The WebServer class has a lot to do:

From errno import EAGAIN, ETIMEDOUT

From machine import reset

From select import poll, POLLIN

From time import sleep, sleep\_ms

Import uasyncio as asyncio

Head = ”HTTP/1.1 200 OK

Content-Type: text/html

Connection: Closed

”

Class WebServer(object):

Def \_\_init\_\_(self, callback, captive\_portal=False):

Import usocket as socket

Self.call = callback

Self.conn = None

Self.s = None

Self.addr = None

Self.path = ”

Self.query = ”

Self.request = ”

Self.method = ”

Self.verbose = False

Self.timeout = 3

Self.captive\_portal = captive\_portal

Try:

If self.captive\_portal:

From dnsquery import CaptivePortal

Self.cp = CaptivePortal(self.verbose)

Self.s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

Self.s.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

Self.addr = socket.getaddrinfo(’0.0.0.0’, 80)[0][-1]

Self.s.bind(self.addr)

Self.s.listen(5) # Allow 5 connections before refusing any new connections

Self.poller = poll()

Self.poller.register(self.s, POLLIN)

Except OSError as e:

Print(”Can’t create socket”, e)

Sleep(20)

Reset()

Def decode\_path(self, req):

If not req:

Return “/”

Cmd, headers = req.split(”\r\n”, 1)

Self.method, self.path, protocol = cmd.split(” “)

Self.query = ”

R = self.path.find(’?’)

If r > 0:

Self.query = self.path[r+1:]

Self.path = self.path[:r]

If self.path == ‘/’:

Self.path = “index.html”

Else:

Self.path = self.path[1:]

Self.verbose and print(”Query:”, self.query, “Path:”, self.path)

Return ‘/’ + self.path

Async def send\_file(self, filename):

Self.verbose and print(”Send file:”, filename)

If filename == “connecttest.txt”:

Return True

If filename == “wpad.dat”:

Return True

Try:

F = open(filename, “rb”)

Except OSError as e:

Print(”Can’t open”, filename, e)

Return False

While True:

Try:

Block = f.read(100)

If len(block) > 0:

Await self.writer.awrite(block)

Else:

Break

Except OSError as e:

Print(”Can’t send”, filename, e)

f.close()

return False

f.close()

return True

async def send\_some\_html(self, html):

await self.writer.awrite(head)

await self.writer.awrite(html)

def parse\_request(self):

self.decode\_path(self.request)

async def serve(self):

self.verbose and print(”serve”)

self.request = ”

html = None

if self.captive\_portal:

self.cp.handle\_dns()

self.conn = None

while True:

# self.verbose and print(”Waiting to accept”)

Got\_something = self.poller.poll(1) # 1 ms timeout

If got\_something:

Try:

Self.conn, address = self.s.accept()

Self.conn.setblocking(False)

Self.call(”accepted”)

Break

Except OSError as e:

Self.conn.close()

If e.args[0] != ETIMEDOUT:

Print(”Exception in accept:”, e)

Sleep(10)

Reset()

Else:

Return False

Else:

Await asyncio.sleep(0.2)

If self.conn:

Self.verbose and print(’\nConnect from’, self.addr[0])

Self.reader = asyncio.StreamReader(self.conn)

Self.writer = asyncio.StreamWriter(self.conn, {})

Data = await self.reader.read(-1)

Self.request = data.decode(”utf-8”)

Data = None

Self.parse\_request()

Self.request = ”

Answer = None

If self.query:

Answer = self.call(self.query)

If answer:

Try:

Await self.send\_some\_html(answer)

Except Exception as e:

Print(”Can’t send html”, e)

Else:

If not await self.send\_file(self.path):

Self.send\_file(”404.html”)

Await self.reader.aclose()

Await self.writer.aclose()

Self.path = ”

Self.query = ”

Self.method = ”

Self.call(”closing”)

Self.conn.close()

Await asyncio.sleep(0.01) # Give time for conn.close to work before closing socket

Self.call(”done”)

Return True

It opens a network socket and polls for connections. When one comes in, it sets up two streams, a reader and a writer. It parses the URL, and if there is a query (a question mark), it calls our callback that we built in main.py. The callback can return some HTML code that will be served.

If there was no query, it reads a file from the flash directory and sends that. This is how index.html and favicon.ico are served.

That’s quite a bit of work just to turn on an LED, but to make the ESP32 do something else we only need to make a few changes to index.html and main.py.

Wifi

Python Programming