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 pi 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.
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)
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) |
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
```

Open Source RF: Exploring The ISM Bands With RTL_433

Follow

5 min read

.

Jul 7, 2025

Listen

Share

More

The RTL-SDR dongle can be a cheap entry into the world of radio.

If you aren’t a medium member, you can read with no paywall via

We’ve spoken before in previous articles about the utility that you can get by acquiring one of the RTL-SDR dongles. These cheap USB devices are a great introduction to the world of radio and can be picked up extremely cheaply, sometimes running at less than $20 delivered to your door.

But what do you do when you get one? And, if you've never had a radio before, how do you turn this little USB device into ears for the world? Read on, because today we'll be showing you how to use your SDR and a simple open-source software package to pick up a slew of unlicensed devices.
For the purposes of today's article, we'll assume you already have an RTL-SDR as well as a computer with either Linux or Windows with WSL. Let's go!
Press enter or click to view image in full size
The ISM Bands
Short for industrial, scientific and medical, it's no exaggeration to say that the ISM bands make the world go round. Here, you'll find weather stations, pacemakers, tyre pressure monitoring systems, as well as a whole bunch of other interesting devices.
Giving unlicensed users the ability to access the radio spectrum with no license required, the ISM bands are a small chunk of spectrum that (within limitations) is available to all users.
Ranging from the High-Frequency bands at around 6MHz through to the microwave bands at 24GHz, w the most common allocations being at 900 + 400MHz. It's the 400 MHz or 70cm band that we're interested in today. It gives acceptable performance, without the need for crazy antennas, and if you're near a big city, you should at least see a few signals of interest to get you started.
Press enter or click to view image in full size
RTL_433
The first thing to do is check out , as you'll get the latest info on the package as well as all the details you need to get it up and running. If you're on Linux or WSL, though, it's as easy as getting it installed. Simply follow the prompts after issuing the command
apt install rtl_433
Once it's installed, we can use the usual flags to check for prompts and other useful commands. So, to bring up the help menu, you can use
rtl_433 -h
While you can run a broad array of flags with your initial command, to start detecting signals, you'll simply need to plug in your RTL device and run
rtl_433
Providing you've set your device up correctly, you should see the following messages in your terminal as the program starts to run.
Press enter or click to view image in full size
If you've got said messages, then that's it! You're ready to go, and any ISM signals within range should be displayed in your terminal thanks to your RTL-SDR.
Press enter or click to view image in full size
What Can I Find?
Well, as it happens, quite a lot actually. A quick look at the notes to find supported protocols shows more than a few interesting vendors there. Needless to say, weather stations, temperature sensors and rain gauges are just some of the devices that you'll find as supported protocols. Look a little harder, and you'll also find LORA, Home Automation devices and possibly even some remote controls of different types. There are, in fact, over 200 protocols listed as compatible in the RTL-433 documents, so there's plenty there to keep you amused.
It's worth mentioning, though, that despite the name, 400MHz isn't the only game in town supported by the RTL hardware. You'll also get coverage at both 300 and 900 MHz, as well, meaning that you shouldn't be surprised if you observe any systems operating within that frequency.
Also, changing out your antenna will give you the best bang for your buck should you wish to take things further, but consider it a "nice to have" problem rather than something essential for beginners.
If you take a look at the attached screenshot, you'll see a bunch of packets that we managed to pick up while compiling the article. While some are dupes, there are still plenty of interesting things to find if you're patient.

Your Gateway Drug
Before you jump too far into this, though, here’s a word of warning.
A cheap receiver paired with a large quantity of curiosity and time was responsible for many people entering the world of radio inadvertently. While it’s decidedly less glamorous in a world with the internet and mobile phones, it’s still an interesting and essential part of our modern world.
So while it starts with a $20 USB dongle picking up ISM signals in your backyard, it’s only a small step away from listening to faraway numbers stations in a radio shack crammed full of test gear.
To be fair, though, this is the magic of science and physics in a nutshell. While the world of radio isn’t without its problems, for many amateurs, hackers and makers, the magic of radio wasn’t ever something that disappeared.
Medium has recently made some algorithm changes to improve the discoverability of articles like this one. These changes are designed to ensure that high-quality content reaches a wider audience, and your engagement plays a crucial role in making that happen.
If you found this article insightful, informative, or entertaining, we kindly encourage you to show your support. Clapping for this article not only lets the author know that their work is appreciated but also helps boost its visibility to others who might benefit from it.
■ Enjoyed this article? Join the community! ■
■ Join our OSINT channel for exclusive updates or
■ Follow our crypto for the latest giveaways
■ Follow us on and
■ We’re now on !
■ Articles we think you’ll like:
What The Tech?!
Shodan:
✉■ Want more content like this?
Rtlsdr
Radio
Radio Hackers
Python Radio 36: Mesh Networking
Follow
16 min read
.
Apr 15, 2025
Listen
Share
More
Get the word out to nodes you can’t see.
Press enter or click to view image in full size
MidJourney
Here in California’s San Francisco Bay Area, we have a very active group using the Meshtastic software to form a mesh network of LoRa nodes.
I can send messages to San Francisco 54 miles away using only a few milliwatts of power and a tiny solar-powered computer that cost me about $30 to put together.
The Meshtastic software is very robust and capable, with lots of bells and whistles. I thought it would be good fun to build a much simpler, smaller system using nodes that cost $5 (although I have found two sites on Aliexpress.com that offer them for 99 cents, but only one to a customer, presumably as a marketing program).
By using the built-in WiFi instead of a separate LoRa radio chip, we save cost at the expense of long distance but gain a lot of extra bandwidth.
The board I chose is one of my new favorites, the ESP32S3 Supermini. It has 4 megabytes of flash

memory, 2.5 megabytes of RAM, runs at 240 megahertz, has two 32-bit processors (and a third 32-bit low-power processor), WiFi, Bluetooth, 24 GPIO pins, an RGB LED, and a Type-C connector for power and programming.
And the little thing is the size of my thumbnail:
Press enter or click to view image in full size
Photo by author
That tiny little surface-mounted red device labeled C3 is the antenna. For most uses, you would probably want to solder a 6-inch wire to that to get better range, but it reaches all over my house through walls, so it is fine for most things.
We use the tiny antenna to make testing our mesh network easier. We want to have some nodes out of reach of at least one node to make sure we are actually networking and not talking to them directly.
The Design
The goal of the project is to make it simple.
So we will not do any fancy routing. We will broadcast each message to all the neighbors we can reach. They will receive it and broadcast it to everyone they can reach.
This could result in a catastrophic explosion of packets, saturating the network quickly, except for two things:
Each packet has a time-to-live number that is decremented before retransmitting the packet. If the number gets to zero, the node drops the packet.
Each packet has a sequence number, and if a node sees a sequence number it has already processed drops the packet. We keep the last 20 sequence numbers in a circular list.
That’s the whole design.
Much of the code is just a simple web server.
The Code
We have two support modules, connect.py and uping.py. We don’t actually need uping.py, but it made debugging the code easier at one point. It ‘ping’s an IP address to verify the connection.
Here is connect.py, the module that set up the WiFi access point and connects to the other nodes’ access points:

```
class Connect:
def __init__( self ):
from network import WLAN, STA_IF, AP_IF, AUTH_OPEN
from random import randint
self.ssid = ""
self.verbose = False
self.network_list = []
self.sta = WLAN( STA_IF )
self.sta.active( False )
self.sta.active( True )
self.ap = WLAN( AP_IF )
self.ap.active( False )
self.ap.active( True )
self.mac = self.ap.config('mac')
self.mac_print = ""
for m in self.mac:
self.mac_print += hex(m)[2:] + ":"
self.mac_print = self.mac_print[:-1]
self.mac_str = ""
for m in self.mac[3:]:
self.mac_str += hex(m)[2:]
ip1 = 10
```

```
ip2 = self.mac[-2]
ip3 = self.mac[-1]
ip4 = 1
ip_str = str(ip1) + "." + str(ip2) + "." + str(ip3) + "." + str(ip4)
ip_mask = "255.255.255.0"
self.ap.ifconfig((ip_str, ip_mask, ip_str, "8.8.8.8"))
self.ap_ip = self.ap.ifconfig()[2]
self.sta_host_ip = False
self.sta_client_ip = False
self.who_am_i = "mesh_" + self.mac_str
self.ap.config( essid=self.who_am_i, authmode=AUTH_OPEN )
self.verbose and print(f"Serving as {self.who_am_i} on subnet {self.ap_ip}")
def connect( self ):
from network import WLAN, STA_IF, AUTH_OPEN
from uping import ping
from time import sleep
self.verbose and print( "Connect" )
for count in range(5):
self.sta.active( True )
try:
self.network_list = self.sta.scan()
# Sort by RSSI
self.network_list = sorted(self.network_list, key=lambda x: x[3], reverse=True)
break
except Exception as e:
print( "scan failed:", e )
self.sta.disconnect()
self.sta.active( False )
self.sta = WLAN( STA_IF )
self.sta.active( True )
try:
if not self.sta.isconnected():
for net in self.network_list:
self.ssid = net[0].decode("utf-8")
if "mesh_" in self.ssid:
try:
self.sta.connect( self.ssid, "" )
count = 0
while not self.sta.isconnected() and count < 10:
sleep( 1 )
count += 1
if not self.sta.isconnected():
continue
ip = self.sta.ifconfig()[0]
self.sta.config(dhcp_hostname=self.who_am_i)
self.sta_host_ip = self.sta.ifconfig()[2]
self.sta_client_ip = self.sta.ifconfig()[0]
p = ping(str(self.sta_host_ip), quiet=True)
if p[1] == 0:
self.verbose and print(f"Can't ping {self.sta_host_ip}")
```

```
else:
# self.verbose and print(f"Pinged: {p}")
# self.verbose and print(f"Ifconfig: {self.sta.ifconfig()}")
# self.verbose and print(f"Connected to host at subnet {self.sta_host_ip}")
pass
return True
except Exception as e:
print( "Error in connect():", e )
pass
else:
if self.verbose:
print( "Already connected" )
return True
except Exception as e:
print( "connect():", e )
self.ssid = ""
self.verbose and print(f"No mesh nodes found")
return False
def reconnect( self ):
# self.verbose and print( "Reconnecting" )
if self.sta.isconnected():
self.sta.disconnect()
self.sta.active( False )
return self.connect()
```

It sets up an access point, advertising itself as mesh_XXXXXX, where the Xs are the last 3 bytes of the MAC address, in hexadecimal.

It then searches for other SSIDs that start with “mesh_” and tries to connect to them.

The first node to power up will not find any. But the second one to power up within reach of the signal will find this node and connect.

The uping.py module implements the ping command:

```
# µPing (MicroPing) for MicroPython
# copyright (c) 2018 Shawwwn <shawwwn1@gmail.com>
# License: MIT
# Internet Checksum Algorithm
# Author: Olav Morken
# https://github.com/olavmrk/python-ping/blob/master/ping.py
# @data: bytes
def checksum(data):
if len(data) & 0x1: # Odd number of bytes
data += b'\0'
cs = 0
for pos in range(0, len(data), 2):
b1 = data[pos]
b2 = data[pos + 1]
cs += (b1 << 8) + b2
while cs >= 0x10000:
cs = (cs & 0xffff) + (cs >> 16)
cs = ~cs & 0xffff
return cs
def ping(host, count=4, timeout=5000, interval=10, quiet=False, size=64):
```

```
import utime
import uselect
import uctypes
import usocket
import ustruct
import urandom
# prepare packet
assert size >= 16, "pkt size too small"
pkt = b'Q'*size
pkt_desc = {
"type": uctypes.UINT8 | 0,
"code": uctypes.UINT8 | 1,
"checksum": uctypes.UINT16 | 2,
"id": uctypes.UINT16 | 4,
"seq": uctypes.INT16 | 6,
"timestamp": uctypes.UINT64 | 8,
} # packet header descriptor
h = uctypes.struct(uctypes.addressof(pkt), pkt_desc, uctypes.BIG_ENDIAN)
h.type = 8 # ICMP_ECHO_REQUEST
h.code = 0
h.checksum = 0
h.id = urandom.getrandbits(16)
h.seq = 1
# init socket
sock = usocket.socket(usocket.AF_INET, usocket.SOCK_RAW, 1)
sock.setblocking(0)
sock.settimeout(timeout/1000)
addr = usocket.getaddrinfo(host, 1)[0][-1][0] # ip address
sock.connect((addr, 1))
not quiet and print("PING %s (%s): %u data bytes" % (host, addr, len(pkt)))
seqs = list(range(1, count+1)) # [1,2,...,count]
c = 1
t = 0
n_trans = 0
n_recv = 0
finish = False
while t < timeout:
if t==interval and c<=count:
# send packet
h.checksum = 0
h.seq = c
h.timestamp = utime.ticks_us()
h.checksum = checksum(pkt)
if sock.send(pkt) == size:
n_trans += 1
t = 0 # reset timeout
else:
seqs.remove(c)
c += 1
# recv packet
```

```
while 1:
socks, _, _ = uselect.select([sock], [], [], 0)
if socks:
resp = socks[0].recv(4096)
resp_mv = memoryview(resp)
h2 = uctypes.struct(uctypes.addressof(resp_mv[20:]), pkt_desc, uctypes.BIG_ENDIAN)
# TODO: validate checksum (optional)
seq = h2.seq
if h2.type==0 and h2.id==h.id and (seq in seqs): # 0: ICMP_ECHO_REPLY
t_elasped = (utime.ticks_us()-h2.timestamp) / 1000
ttl = ustruct.unpack('!B', resp_mv[8:9])[0] # time-to-live
n_recv += 1
not quiet and print("%u bytes from %s: icmp_seq=%u, ttl=%u, time=%f ms" % (len(resp), addr, seq,
ttl, t_elasped))
seqs.remove(seq)
if len(seqs) == 0:
finish = True
break
else:
break
if finish:
break
utime.sleep_ms(1)
t += 1
# close
sock.close()
ret = (n_trans, n_recv)
not quiet and print("%u packets transmitted, %u packets received" % (n_trans, n_recv))
return (n_trans, n_recv)
```

Unlike the other modules, I didn’t write this one. But it came in quite handy, so I share it here.

The Main Program

As you might expect, most of the work is done in the main program. Some of it is just a simple web server, something we’ve covered earlier.

The unquote(), handle_query(), decode_path(), req_handler(), and client_handler() process packets that come in and parse the few HTTP protocols we accept. There is nothing “mesh networky” about them.

The dup_sequence() method handles the circular list of sequence numbers we’ve seen. When it finds a duplicate, it returns True to drop the packet.

The show_neighbors() method is for debugging. It prints the neighbor list on the console.

The whodat dictionary helps keep track of the names of the nodes. It is also just for debugging, and it occasionally gets it wrong in non-critical ways.

When a node connects to our node, add_neighbor() puts it into a list of neighbors. It ignores neighbors we already have in the list. Importantly, it calls greet() to send a message back to the connecting node, telling it who we are, and collecting similar information in the reply.

Before I go any further, here is the code:

```
from uasyncio import get_event_loop
from connect import Connect
from time import sleep
from machine import Pin
from neopixel import NeoPixel
```

```
from socket import getaddrinfo, socket, AF_INET, SOCK_STREAM, SOL_SOCKET, SO_REUSEADDR
from whoami import WhoAmI
from sys import print_exception
from random import randint
w = WhoAmI()
rgb = NeoPixel(Pin(w.neo_pin(), Pin.OUT), 1)
def unquote(string):
if not string:
return b""
if isinstance(string, str):
string = string.encode("utf-8")
bits = string.split(b"%")
if len(bits) == 1:
return string
res = bytearray(bits[0])
append = res.append
extend = res.extend
for item in bits[1:]:
try:
append(int(item[:2], 16))
extend(item[2:])
except KeyError:
append(b"%")
extend(item)
return bytes(res).decode("utf-8")
class Mesh:
def __init__(self):
self.verbose = False
self.neighbors = []
self.query = ""
self.sequence_number = randint(1, 100_000_000)
self.sequences = [0] * 20
self.seq_count = 0
self.path = ""
self.port = 80
self.whodat = {}
self.con = None
self.name = w.name()
self.addr = getaddrinfo("0.0.0.0", self.port)[0][-1]
self.srv = socket(AF_INET, SOCK_STREAM)
self.srv.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
self.srv.bind(self.addr)
self.srv.listen(10) # at most 10 clients
self.srv.setblocking(False)
self.srv.setsockopt(SOL_SOCKET, 20, self.client_handler)
def dup_sequence(self, seq):
for s in self.sequences:
if seq == s:
return True
self.sequences[self.seq_count] = seq
```

```
self.seq_count += 1
self.seq_count %= 20
return False
def show_neighbors(self):
print("Neighbors:")
print("----------")
for n in self.neighbors:
ssid, ip, name = n
if ip == self.con.sta_host_ip:
print(f"Neighbor '{ssid}' '{ip}' '{name}' [Our host]")
else:
print(f"Neighbor '{ssid}' '{ip}' '{name}'")
for n in self.whodat.values():
his_name, his_ip = n
print(f"Client '{his_name}' '{his_ip}'")
print("----------")
def who_name(self, ap_ip, sta_ip, name):
if name != "unknown":
for n in self.whodat.values():
his_name, his_ip = n
if ap_ip[:-2] == his_ip[:-2]:
self.whodat[his_ip] = (name, his_ip)
if sta_ip and sta_ip[:-2] == his_ip[:-2]:
self.whodat[his_ip] = (name, his_ip)
def add_neighbor(self, ip, ssid="no ssid", name="unknown"):
self.who_name(ip, None, name)
# self.verbose and print(f"Adding neighbor: {ip} {ssid} {name}")
for i, n in enumerate(self.neighbors):
n_ssid, n_ip, n_name = n
# self.verbose and print(f"Checking neighbor: {n_ip} {n_ssid} {n_name}")
if n_ip == ip:                  # We've seen this before
# self.verbose and print(f"We already have this neighbor")
if n_name == "unknown":
self.neighbors[i][2] = name
if n_ssid == "no ssid":
self.neighbors[i][0] = ssid
return
if ip == self.con.ap_ip:            # We are not our own neighbor
# self.verbose and print(f"We are not our own neighbor")
return
try:
self.greet(ip, name)
except Exception as e:
self.verbose and print(f"Exception in connecting: {e}")
print_exception(e)
pass
for n in self.neighbors:
n_ssid, n_ip, n_name = n
if n_ssid == ssid and n_ip = ip and n_name == name:
return
```

```
self.verbose and print(f"{w.name()}: Added neighbor {ssid} {ip} {name}")
self.neighbors.append([ssid, ip, name])
def execute(self, payload):
if payload.get("rgb", False):
i = payload["rgb"]
r = (i >> 16) & 0xFF
g = (i >>  8) & 0xFF
b = (i      ) & 0xFF
rgb[0] = (r, g, b)
rgb.write()
name = payload["name"]
ap_ip, sta_ip = payload["origin"]
self.who_name(ap_ip, sta_ip, name)
elif payload.get("text", False):
print()
msg = payload["message"]
who = payload["to"]
name = payload["name"]
print(f"Message from {name} to {who}:")
print(msg)
print()
pass
elif payload.get("greet", False):
name = payload["name"]
ap_ip, sta_ip = payload["origin"]
self.who_name(ap_ip, sta_ip, name)
else:
self.verbose and print(f"Execute({payload})")
def colors(self):
import json
from random import randint
r = randint(0, 255)
g = randint(0, 255)
b = randint(0, 255)
for n in self.neighbors:
ssid, ip, name = n
payload = {
"name": w.name(),
"to": name,
"ssid": self.con.who_am_i,
"rgb": (r << 16) + (g << 8) + b,
"time_to_live": 1,
"direct": self.con.ap_ip,
"origin": (self.con.ap_ip, self.con.sta_client_ip)
}
self.direct_msg(ip, "none", json.dumps(payload))
def greet(self, ip, name):
import json
# self.verbose and print(f"Greeting {name} at {ip}")
payload = {
```

```
"name": w.name(),
"to": name,
"ssid": self.con.who_am_i,
"greet": True,
"time_to_live": 0,
"direct": self.con.ap_ip,
"origin": (self.con.ap_ip, self.con.sta_client_ip)}
self.direct_msg(ip, "none", json.dumps(payload))
def direct(self, ip):
if ip == self.con.sta_host_ip:
# self.verbose and print(f"Sending to our host")
return True
for n in self.whodat.values():
his_name, his_ip = n
if ip == his_ip:
# self.verbose and print(f"Sending to our client {his_name} at {his_ip}")
return True
subnet_ip = ip[:-2]
subnet_ap = self.con.ap_ip[:-2]
if subnet_ip == subnet_ap:
# self.verbose and print(f"Sending to same subnet as our ap: {subnet_ip}")
return True
return False
def direct_msg(self, ip, file, payload_string):
from urequests import get
from random import randint
import json
self.sequence_number += randint(1, 1_000)
payload = json.loads(payload_string)
payload["direct"] = self.con.ap_ip
payload["sequence"] = self.sequence_number
payload_string = json.dumps(payload)
# self.verbose and print(f"My ip addresses: {self.con.ap_ip} {self.con.sta_client_ip}")
if not self.direct(ip):
# self.verbose and print(f"Can't reach {ip} directly")
return
for count in range(5):
try:
get_str = f"http://{ip}/{file}?packet={payload_string}"
response = get(get_str, timeout=randint(1, 10))
if response.status_code != 200:
response.close()
continue
if response.text[0] == "{":
p = json.loads(response.text)
self.add_neighbor(p["origin"][0], p["ssid"], p["name"])
response.close()
return
except OSError as e:
if e.args[0] == 104: # ECONNRESET
```

```
# self.verbose and print(f"Connection to {ip} reset")
pass
elif e.args[0] == 113: # ECONNABORTED
# self.verbose and print(f"Connection to {ip} aborted")
pass
elif e.args[0] == 116: # ETIMEDOUT
# self.verbose and print(f"Connection to {ip} timed out")
pass
else:
print(f"Exception in get: {e}")
print_exception(e)
except Exception as e:
print(f"Exception in get: {e}")
print_exception(e)
def broadcast(self, payload_string):
from urequests import get
self.show_neighbors()
print()
print("**************************************************")
print(f"Broadcast:")
for n in self.neighbors:
ssid, ip, name = n
# self.verbose and print(f"Send to {name} at {ip} ", end="")
# self.verbose and print(payload_string)
self.direct_msg(ip, "none", payload_string)
# for n in self.whodat.values():
#     his_name, his_ip = n
#     self.verbose and print(f"Send to {his_name} at {his_ip} ")
#     self.direct_msg(his_ip, "none", payload_string)
print("**************************************************")
print()
def handle_query(self):
import json
command, payload_string = self.query.split("=")
payload_string = unquote(payload_string)
if command == "packet":
payload = json.loads(payload_string)
name = payload.get("name", "unknown")
ssid = payload.get("ssid", "no ssid")
to = payload.get("to", "unknown")
seq = payload.get("sequence", False)
if self.dup_sequence(seq):
# self.verbose and print("We already saw this message:", payload_string)
return
direct = payload.get("direct", "unknown")
its_from_ap, its_from_sta = payload.get("origin", ("unknown", "unknown"))
ttl = payload.get("time_to_live", 0)
# self.verbose and print("Query:", name, ssid, to, its_from_ap, its_from_sta)
self.add_neighbor(its_from_ap, ssid, name)
self.add_neighbor(its_from_sta, ssid, name)
```

```
if to == w.name():                  # It's for me
self.execute(payload)
elif ttl > 0:
ttl -= 1
payload["time_to_live"] = ttl
payload_string = json.dumps(payload)
if to == "all":
self.broadcast(payload_string)
else:
# Here is where we would put the routing code
self.broadcast(payload_string)
elif command == "text":
payload = json.loads(payload_string)
text = payload.get("text", "")
who = payload.get("to", "")
its_from = payload.get("from", "")
message = payload.get("message", "")
seq = payload.get("sequence", False)
if self.dup_sequence(seq):
# self.verbose and print("We already saw this message:", payload_string)
return
print()
print("Message:")
print(payload_string)
payload = {
"text": True,
"message": message,
"name": its_from,
"to": who,
"time_to_live": 3,
"direct": self.con.ap_ip,
"origin": (self.con.ap_ip, self.con.sta_client_ip)}
self.broadcast(json.dumps(payload))
print()
else:
self.verbose and print("Unexpected query:", self.query)
return None
def send_file(self, client_socket):
from ubinascii import hexlify
# self.verbose and print("Send file:", self.path)
if self.path:
if self.path == "/index.html":
style = "<style>\r\n      th, td\r\n      {\r\n        padding: 5px;\r\n        spacing: 5px;\r\n
}\r\n    </style>"
my_info = f"<h2>{w.name()}: ap={self.con.ap_ip} sta={self.con.sta_client_ip} on
{self.con.who_am_i}</h2><p/>"
neighbors = "<table border=1>"
for n in self.neighbors:
ssid, ip, name = n
neighbors += f"\r\n      <tr>\r\n        <td>{name}</td>\r\n        <td>{ip}</td>\r\n
```

```
<td>{ssid}</td>\r\n      </tr>"
for n in self.whodat.values():
his_name, his_ip = n
neighbors += f"\r\n      <tr>\r\n        <td>Child</td>\r\n        <td>{his_ip}</td>\r\n
<td>{his_name}</td>\r\n      </tr>"
neighbors += "\r\n    </table>"
station_str = "<p/>Stations:\r\n"
stations = self.con.ap.status("stations")
for s in stations:
mac = hexlify(s[0]).decode()[4:]
station_str += f"    <br/>{mac}\r\n"
r, g, b = rgb[0]
rgb_str = f"(r={r}, g={g}, b={b})"
script = """
<script>
sequence_number = Math.round(Math.random() * 10000);
function sendTextToServer() {
sequence_number++;
const who = document.getElementById("who").value;
const from = document.getElementById("from").value;
const textToSend = document.getElementById("textInput").value;
packet = { "message": encodeURIComponent(textToSend), "to": who, "from": from, "sequence":
sequence_number }
const url = "/text?text=" + JSON.stringify(packet);
fetch(url, { method: 'GET', })
.then(response => {
if (!response.ok) { throw new Error("HTTP error! status: " + response.status); }
return response.text();
})
.then(data => {
document.getElementById("responseFromServer").textContent = "Server Response: " + data;
document.getElementById("textInput").value = "";
})
.catch(error => {
console.error("Error sending text:", error);
document.getElementById("responseFromServer").textContent = "Error: " + error.message;
});
}
</script>
"""
file = f"""
<html>
<head>
<title>Mesh</title>
{style}
{script}
</head>
<body>
{my_info}
<h3>Mesh: {w.name()}'s neighbors:</h3>
```

```
{neighbors}
{station_str}
<h3>LED:</h3>
RGB is {rgb_str}
<p/>
<form id="textForm">
<div>
<label for="who">Send to:</label>
<input type="text" id="who" name="text" placeholder="all">
<br/>
<label for="from">Send from:</label>
<input type="text" id="from" name="text" placeholder="Joe">
<br/>
<label for="textInput">Enter text to send:</label>
<input type="text" id="textInput" name="text" required>
</div>
<button type="button" onclick="sendTextToServer()">Send Text</button>
</form>
<div id="responseFromServer"></div>
</body>
</html>
"""
client_socket.write(f"HTTP/1.1 200 OK\r\nContent-Length: {len(file)}\r\nContent-Type:
text/html\r\n\r\n")
client_socket.write(file)
elif self.path == "/text":
file = "Received text message"
client_socket.write(f"HTTP/1.1 200 OK\r\nContent-Length: {len(file)}\r\nContent-Type:
text/html\r\n\r\n")
client_socket.write(file)
elif self.path == "/none":
import json
payload = {
"name": w.name(),
"ssid": self.con.who_am_i,
"direct": self.con.ap_ip,
"origin": (self.con.ap_ip, self.con.sta_client_ip)}
file = json.dumps(payload)
client_socket.write(f"HTTP/1.1 200 OK\r\nContent-Length: {len(file)}\r\nContent-Type:
text/html\r\n\r\n")
client_socket.write(file)
else:
with open(self.path, "rb") as f:
file = f.read()
client_socket.write(f"HTTP/1.1 200 OK\r\nContent-Length: {len(file)}\r\nContent-Type:
text/html\r\n\r\n")
client_socket.write(file)
def decode_path(self, req):
if not req:
return "/"
```

```
self.path = None
cmd, headers = req.split("\r\n", 1)
if cmd:
s = cmd.split(" HTTP/")
cmd = s[0]
self.query = ""
r = cmd.find("?")
if r > 0:
self.query = cmd[r+1:]
cmd = cmd[:r]
self.method, self.path = cmd.split(" ")
if self.path == "/":
self.path = "/index.html"
return self.path
def req_handler(self, client_socket):
try:
req = client_socket.read()
if req:
self.decode_path(req.decode("utf-8"))
# self.verbose and print("Decoded path:", self.path)
self.send_file(client_socket)
client_socket.close()
if self.query != "":
self.handle_query()
return
except OSError as e:
if e.args[0] == 128:        # ENOTCONN
pass
except Exception as e:
self.verbose and print("Err:", e)
print_exception(e)
client_socket.close()
def client_handler(self, srv):
try:
client_socket, addr = srv.accept()
except OSError as e:
if e.args[0] == 23:         # Too many open files: we forgot to close a socket somewhere
pass
# self.verbose and print("Serving:", addr[0])
self.whodat[addr[0]] = ("unknown", addr[0])
client_socket.setblocking(False)
client_socket.setsockopt(SOL_SOCKET, 20, self.req_handler)
def main():
rgb[0] = (64, 0, 0)
rgb.write()
sleep(10)               # During development, so the chip allows file transfers
con = Connect()
con.verbose = True
con.reconnect()
rgb[0] = (64, 64, 0)
```

```
rgb.write()
mesh = Mesh()
mesh.con = con
mesh.verbose = True
if con.sta_host_ip:
# mesh.verbose and print(f"{w.name()}: Connected to {con.ssid} at {con.sta_host_ip}")
mesh.add_neighbor(con.sta_host_ip, con.ssid, "unknown")
rgb[0] = (0, 64, 0)
rgb.write()
while True:
sleep(3 * 60)
print("============================================================")
mesh.show_neighbors()
mesh.colors()
if con.sta and con.sta.isconnected() == False:
con.reconnect()
if con.sta_host_ip:
# mesh.verbose and print(f"{w.name()}: Connected to {con.ssid} at {con.sta_host_ip}")
mesh.add_neighbor(con.sta_host_ip, con.ssid, "unknown")
main()
```

The execute() method handles special message types the node can receive. One is a message that se the colors and brightness of the RGB LED. This is handy for debugging, as not all nodes will have a computer attached for serial output.

It also handles printing out messages, and the reply to the greet() packet. The latter calls who_name() to associate the name with the IP addresses we got from the connecting node.

The whole point of a mesh network is to communicate with nodes we can't connect to directly. That's where the direct() method comes in. It looks at all the IP addresses we know about and returns False if the address given is unreachable.

We finally get to direct_msg(), the heart of the code. This assigns sequence numbers and inserts them into the payload, and then tries 5 times to send the packet, stopping at the first success. If it gets a reply, it adds the node as a neighbor (it lets add_neighbor() handle duplicates).

The broadcast() method is simple. It goes through the neighbor list and sends the packet to each one.

We use the HTTP GET method for packets, which means some packets might have a question mark (a 'query') with commands in it. If the command is "packet", we collect the neighbor information, store it, and then check to see if the packet is for us. If it was, we call execute(). Otherwise we decrement the time_to_live number and broadcast the packet to everyone we know.

If the command was "text", then we got it from a web browser (more on that in a bit). We repackage the data (adding our IP addresses, time_to_live, etc.) and broadcast it.

The part of the HTTP packet ahead of the query is the path. This is normally the file the server will read and send to the browser, and indeed we do that if asked. But we also handle "fake" files /text and /none. The /text file just acknowledges that we got a text message, so the web browser that sent it knows we got it. The /none file answers the greet message by returning our name, ssid, and IP addresses.

The bulk of the code in send_file() creates the /index.html file to send to any web browser that connects with us. This file prints out node information and includes a form for sending messages. The form looks like this:

I named my nodes Alice, Bob, and Carol (the next one will be Ted).

We can see that Carol has Alice as a neighbor, but Carol can't see Bob. I sent a message to Bob, telling Bob my browser was Joe, and Carol sent it to Alice who then sent it to Bob. The meshing

worked (finally, after three weeks of coding and debugging).
The main() method sets the RGB LED to red, then connects to anyone it can, sets the RGB LED to yellow while it initializes the Mesh class, and then sets it to green before looping to do something every 3 minutes just to show that the mesh is meshing.
The something that it does is just to print out the neighbor list on the console (if there is one) and then broadcast a message to set everyone's LED to a random color. Again, just to show the syster is up and running.
When setting up a node out somewhere here on the farm, it is nice not to need a laptop to see what's going on. Watching the LED change color tells me everything is working.
It is now easy to add features. You can collect temperature, humidity, and other weather data and send it out to the mesh. You can turn lights on and off, or tell when the well pump is running (you all have those right? Or is it just me?)
Each node has 150 megabits of bandwidth to play with (15 megabytes per second). Our typical packet length is about 100 bytes, so we could theoretically send 150,000 packets per second. We send each of our packets 3 times per node on the assumption all nodes are within 3 hops of one another, but we could easily change this, or make it an option in the web form. But the network would have to get pretty big before we ran out of bandwidth, even though we waste it by using the simple design.
Programming
Education
Software Development
Python
Python Radio 39: Secret Unbreakable Code
Simon Quellen Field
Simon Quellen Field
Follow
10 min read
.
May 19, 2025
Listen
Share
More
A One-Time Pad is Cryptographically Secure.
Press enter or click to view image in full size

Secret agent reading coded message.
MidJourney
How sure are you that governments can't break the encryptions commonly in use? If you were a journalist in a repressive dictatorship, would you trust your freedom or your life to them?
There is a well-known method that is proven to be safe against cryptographic attacks. It is called a one-time pad.
The idea is simple. Each person has a pad of paper that has random letters on it. They code the message by adding the next random letter to each letter of the original message. Then they burn that page or the pad.
In the 1940s, the information theorist Claude Shannon proved this was unbreakable by cryptanalytic methods.
Four things have to be true for it to be secure. The pad must be truly random (no pseudo-random sequences such as those computers generate). You can never reuse a page of the pad. Both people m keep the pad secret. And the pad must be at least as long as the message to be encoded.
The last two things are the concern of the two people, and the code and hardware I present here don't address them.
Never reusing a page sounds simple. We just delete the page from memory. But is it truly deleted?

Can the NSA or the FSB read the deleted text and then be able to decode old messages?
We handle that by writing random text over the old page instead of deleting it. The truly paranoid can do this in a loop, so that the page gets overwritten by new random data many times. My code does it just once.
Enter the Hardware
That leaves the first requirement: that we make the pad from truly random letters. The computer can’t do that by itself. It needs hardware help.
Here is my design for a hardware random bit generator:
Press enter or click to view image in full size

Schematic for hardware random bit generator.
Schematic by the author
This design uses the unpredictable breakdown of a Zener diode that is due to the quantum mechanics of electron tunneling.
I ordered the parts and started writing the code.
But as I was writing, it occurred to me that I have another handy source of random entropy.
The Benefits of Radio Static
One of the problems we encountered when dealing with the data link using the 433 MHz transmitters and receivers was the problem of noise. When the signal was strong, the receiver performed well. The signal-to-noise ratio was excellent, even at half a kilometer.
But when we weren’t transmitting, we got flooded with noise. The receiver would increase its sensitivity until it heard something, even when there was nothing to hear.
And I already had the receiver. All I needed to do was listen to the output when the transmitter was off. That radio noise has all the entropy we need. We don’t even need an antenna.
I connected the power, ground, and output of the receiver to an ESP32-S3 and collected some data using the analog-to-digital port.
Some Software Assistance
Either hardware source has plenty of entropy (randomness). But we won’t assume that there isn’t some bias in there somewhere. Many of the samples will be zero, since I didn’t bother to add a DC bias to bring negative samples into the range of the ADC. We can handle all of that in software if we collect more entropy than we need.
There is a Python library full of hash routines that have been exhaustively tested for their cryptographic properties. We can collect 800 bits of entropy and hand that to the SHA256 hashing algorithm to get 256 bits of well-characterized random numbers.
We are now ready to walk through the code.
```
From machine import Pin, ADC, freq
From os import statvfs, remove, stat
From hashlib import sha256
From sys import print_exception
Freq(240_000_000)
Green_led = Pin(34, Pin.OUT)
Hex_digit = {
“0”: 0,
“1”: 1,
“2”: 2,
“3”: 3,
“4”: 4,
“5”: 5,
“6”: 6,
“7”: 7,
“8”: 8,
```

“9”: 9,
“a”: 10,
“b”: 11,
“c”: 12,
“d”: 13,
“e”: 14,
“f”: 15,
}
Def df():
S = statvfs(‘//’)   # Note: ‘//’ represents the root filesystem
Block_size = s[0]
Free_blocks = s[3]
Print(f”Block size: {block_size}”)
Print(f”Free blocks: {free_blocks}”)
Return int(free_blocks * block_size)
Class OTP:
Def __init__(self):
Self.code = “”
Self.unused = 0
B = None
Try:
With open(“one_time_pad.nxt”, “rb”) as f:
B = f.read()
Except OSError as e:
Pass
If b:
Self.unused = int.from_bytes(b, ‘little’)
Else:
Print(f”Next address is starting at {self.unused}”)
Self.last_unused = self.unused
Def in_hex(self, b):
S = “”
For x in b:
S += f”{x:02x}”
Return s
Def update(self):
From time import ticks_cpu
B = self.unused.to_bytes(4, ‘little’)
With open(“one_time_pad.nxt”, “wb”) as f:
f.write(b)
if self.unused == 0:
return
size = self.unused – self.last_unused
random = (int.from_bytes(self.code[0:4], ‘little’) + ticks_cpu()) % 1_000_000
with open(“one_time_pad.otp”, “r+b”) as f:
b = b’’
while len(b) == 0:
f.seek(random, 0)
b = f.read(4)
random = (int.from_bytes(b, ‘little’) + ticks_cpu()) % 1_000_000

```
f.seek(random, 0)
b = f.read(size)
f.seek(self.last_unused, 0)
backward = bytes(reversed(b))
f.write(backward)                    # Clobber used-up pad bytes with random data
def delete_one_time_pad(self):
try:
remove( “one_time_pad.nxt”)
except OSError as e:
pass
try:
remove( “one_time_pad.otp”)
except OSError as e:
pass
# @micropython.native
Def create_one_time_pad(self):
Self.unused = 0
Leave_free = 512 * 1_024
Free = df() / 2                      # Don’t use more than half the remaining flash
Print(f”Filesystem free space: {free}”)
If free < leave_free:
Return
Adc = ADC(Pin(10))
Try:
Stat(“one_time_pad.otp”)
Print(“One time pad already exists”)
Return
Except OSError:
Pass
With open(“one_time_pad.otp”, “wb”) as f:
From time import ticks_ms, ticks_diff
Start = ticks_ms()
Count = 0
While free > leave_free:
# Collect 100 bytes of random noise (800 bits of entropy)
Noise_string = b’’
For characters in range(25):
Noise = 0
For foo in range(20):
Noise += int(adc.read())
Noise_string += noise.to_bytes(4, ‘little’)
Sha = sha256(noise_string).digest()
f.write(sha)
free -= len(sha)
left = free – leave_free
count += len(sha)
elapsed_seconds = ticks_diff(ticks_ms(), start) / 1_000
rate = count / elapsed_seconds
mins = int((left / rate) // 60)
hrs = int(mins // 60)
```

```
mins -= hrs * 60
# print(f”{left:6d}, [time left:{hrs:2d}:{mins:02d}]  {self.in_hex(sha)}”)
Print(f”time left:{hrs:2d}:{mins:02d}”, end=”\r”)
Self.unused = 0
Self.last_unused = 0
Self.update()
Def decrypt(self, pwd, hex_msg):
Pl = len(pwd)
Msg = hex_msg[0:8]
For x in range(8, len(hex_msg), 2):
Msg += chr((hex_digit[hex_msg[x]] << 4) | hex_digit[hex_msg[x+1]])
Unused = int(msg[0:8], 16)
L = len(msg) – 8
Out = “”
With open(“one_time_pad.otp”, “rb”) as f:
f.seek(unused, 0)
self.code = f.read(l)
for x in range(l):
p = ord(pwd[x % pl])
out += chr(p ^ self.code[x] ^ ord(msg[x+8]))
try:
msg_length = int(out[0:8], 16) + 8
except ValueError as e:
print(“Decode failed. Don’t decode on the same device you encode on.”)
return “”
if unused > self.unused:
self.unused = unused                    # Stay matched with the other device so we can reply
return(out[8:msg_length])
def encrypt(self, pwd, msg):
pl = len(pwd)
l = ((len(msg) // 32) + 1) * 32
out = bytearray(b’\x00’ * (l+8))
out[:8] = bytearray(f”{self.unused:08.8x}”, “utf-8”)
with open(“one_time_pad.otp”, “rb”) as f:
f.seek(self.unused, 0)
try:
self.code = f.read(l)
except Exception as e:
print_exception(e)
for x in range(len(msg)):
p = ord(pwd[x % pl])
out[x+8] = int(p ^ self.code[x] ^ ord(msg[x]))
start = len(msg) + 8
stop = l
for cnt in range(start, stop, 1):
print(cnt)
out[cnt] = int(self.code[cnt])
print(“Test decode: “, end=””)
test = bytearray(b’\x00’ * len(msg))
for x in range(len(msg)):
```

```
p = ord(pwd[x % pl])
test[x] = int(p ^ self.code[x] ^ out[x+8])
for x in test:
if x >= 32 and x < 127:
print(chr(x), end="")
else:
print(".", end="")
print()
self.unused += l
self.update()
return out[:l]
def ui(self):
Green_led.on()
While True:
Test = input("Create, Remove, Encrypt, or Decrypt? ")
If len(test) == 0:
Print("Type c, r, e, or d")
Elif test[0].lower() == 'c':
Self.create_one_time_pad()
Elif test[0].lower() == 'r':
Confirm = input("Really delete the one time pad that takes a long time to build? ")
If confirm == "yes":
Self.delete_one_time_pad()
Elif test[0].lower() == 'e':
Pwd = input("Enter password: ")
Msg = input("Enter message to encrypt: ")
Coded = self.encrypt(pwd, f"{len(msg):08.8x}" + msg)
Print(f"Coded: {coded[0:8].decode()}{self.in_hex(coded[8:])}")
Elif test[0].lower() == 'd':
Pwd = input("Enter password: ")
Msg = input("Enter message to decrypt: ")
If msg != "":
Decoded = self.decrypt(pwd, msg)
Print(f"Decoded: {decoded}")
Else:
Print("Type c, r, e, or d")
Def main():
Otp = OTP()
Otp.ui()
Main()
```

We'll start with the create_one_time_pad() method.

We want a large pad, but we don't want to completely fill up our flash memory. We find out how much space we have, use only half, and leave at least half a megabyte free.

Now we collect some random bits. We add up 20 bytes from the ADC to get a 4-byte integer. We conve that into a 4-byte string and append it to our string of random characters. We do this 25 times to get 100 bytes of random noise.

We hand that string to the SHA256() method, and write the resulting 32 bytes to our one-time pad. We continue doing this until the pad is full.

This can take about 15 minutes, so we want to give the user some indication of progress. We calculate the rate at which the file grows and give an estimate of the completion time. On smaller

devices with only 4 megabytes of flash, it may take five minutes or less.
Finally, we call the update() method.
Update() has two jobs. It keeps track of which part of the pad we have used in a file called one_time_pad.nxt. Its second job is to clobber the part we have used. We want to put random data there. But where can we find random data? Oh, yes! We have a whole file full of it!
We take 4 bytes of our last random patch and add the clock to it. Then we use that to seek into the file to get another one. We keep doing that until we get something that isn't zero. I did this because it was convenient during debugging to kill the program before the whole 15 minutes were up, and the file was shorter than a megabyte. Randomly seeking would often lead off the end of the file, returning zeros.
Once we had enough random bits to write over the part of the pad we had consumed, we reverse the b string for luck and write over the used part of the pad.
Now we are ready to encrypt a message.
You have probably used two-factor authentication before as a security method. Three-factor authentication is a little better. The three parts are:
Something you know (such as a password).
Something you have (in our case, that is our one-time pad).
Something you are (this is where fingerprint readers or iris scans come in).
You can buy fingerprint readers for about $10 that are easy to connect to our ESP32 using a UART port. Mine is on order, so this code doesn't use a fingerprint scanner yet, although that is an easy software adjustment to make.
The code does use a password, however. So we have two factors, which is good enough for my bank, maybe it's good enough for now.
The password and the message to encode get passed to the encrypt() method.
Encrypt() adds the address into the pad as the first 8 bytes. This is not encrypted.
To keep an adversary from knowing how long our message is, we bump the length up to the next 32 bytes. Our encrypted coded message will always be a multiple of 32 bytes long.
Then, for each letter in the message, we XOR it with a letter from the password and a letter from the one-time pad.
The nice feature of the XOR function is that you can run it twice and get the original message back. That is how we decrypt. To make sure we did it right, we decrypt it right away and present it to the user. She already knows the message, so we haven't given anything away.
Lastly, we call update to save the address and clobber the code.
Decrypt() reads the address (remember, we did not encrypt that) in the first 8 bytes of the coded message. It reads the pad at that address and does the XORs. The first 8 bytes of the message are the message length. We need that so we can ignore the random noise that fills out the length to 32 bytes. Then we return the decoded message.
The only part left is the main() function, which is mostly self-explanatory.
It collects the password and message and adds the message length before calling encrypt(). At the other end, the message recipient will paste the message into decrypt() to read it.
The Results
Let's look at the code in action. Here is the encoding:
Create, Remove, Encrypt, or Decrypt? E
Enter password: this is a longish password with lots of entropy and yet still something I can remember.
Enter message to encrypt: This is a test of our one-time pad encryption mechanism.
Test decode: 00000038This is a test of our one-time pad encryption mechanism.
Coded:
5d8adb1b51b0081b606d8fe8ff404589622fdd606997e09829b88c019469e8c260942583acc8fc8a37aab
Create, Remove, Encrypt, or Decrypt?

Next, we copy the coded message into the decoding machine:
Create, Remove, Encrypt, or Decrypt? D
Enter password: this is a longish password with lots of entropy and yet still something I can remember.
Enter message to decrypt:
5d8adb1b51b0081b606d8fe8ff404589622fdd606997e09829b88c019469e8c260942583acc8fc8a37aab
Decoded: This is a test of our one-time pad encryption mechanism.
Create, Remove, Encrypt, or Decrypt?
Make sure you don't try to decrypt the message on the same machine you encrypted it on. That machir has already clobbered the one-time pad used to encrypt the message.
Some Words About Downloading
Our code requires the hashlib library. We install that using the mpremote program running on the host machine (a Windows machine in my case):
Mpremote connect com4 mip install hashlib
We build the one-time pad on one esp32, and then copy the two files to the other esp32:
Mpremote connect com4 fs cp :one_time_pad.nxt one_time_pad.nxt
Mpremote connect com4 fs cp :one_time_pad.otp one_time_pad.otp
Mpremote connect com8 fs cp one_time_pad.nxt :one_time_pad.nxt
Mpremote connect com8 fs cp one_time_pad.otp :one_time_pad.otp
The colon tells mpremote we are talking about a file on the remote machine. And (of course) you will change com4 and com8 to the ports you use on your own machine. On a Mac or Linux, they will be fou in /dev.
Since the one_time_pad.otp file is rather large, expect to wait a while for it to copy.
Lastly, you must securely delete the one_time_pad.otp from your host machine. Write over it several times with random data.
Python Radio 41: Radar!
Follow
5 min read
.
May 30, 2025
Listen
Share
More
Exploring microwave signals above 3 GHz.
Press enter or click to view image in full size
All images by the author.
You can buy microwave motion detectors for less than a dollar. I bought a 10-pack on Amazon for $7.99.
Sometimes advertised as "Doppler radar" devices, the RCWL-0516 is a little board that puts out a 3.16 GHz continuous wave and listens for reflections.
They don't operate the same way that aircraft radars or weather radars do. Those send out pulses and time the reflections to get the range and detect wavelength changes to get the speed towards or away (using the Doppler effect).
Despite the advertising, these little boards detect the changes in the phase of the reflected waves by heterodyning them ("beating") with the continuous-wave transmissions. The interference of the forward and reflected waves cause changes in the output amplitude that are in the audio range or lower.
Press enter or click to view image in full size
In the photo above, I have soldered a red LED between the output and ground, and powered the device from a 3.7-volt battery connected to the +3.3-volt input. The device can handle 4.2 volts on that

input, but it also has a VIN input that connects to a voltage regulator, allowing as much as 28 volts to be used. It will run a long time on a 9-volt battery that way, as it consumes less than 3 milliamperes of current (more if the LED is lit).
The output stays on for about 5 seconds. You can increase this by adding a capacitor and/or a resistor on the back of the board. That's the normal mode.
But we aren't going to use the board in its normal mode. We're going to hack it to do much more.
Let's look at a schematic of the board:
Press enter or click to view image in full size
The transmitter is the transistor at the right, forming an oscillator at about 3.175 GHz.
The antenna both sends out the signal and receives the echo. The echo signal is mixed in that same transistor, forming the sum and difference frequencies, just like a superheterodyne receiver. The sum frequency is filtered out on the way to pin 14 of the integrated circuit, leaving the low-frequency difference signal.
Below is the schematic of the integrated circuit:
Press enter or click to view image in full size
We are going to ignore most of it. What we want is the operational amplifier labeled OP1. The weak difference signal from the antenna and mixer comes in on pin 14. We want the amplified version of that on pin 16.
While it is possible to solder a thin wire to pin 16 on the IC, that's a little tricky because the pins are very small and close together.
But we're in luck.
Resistor R3 on the schematic of the entire board is labeled R-GN. It does not exist on the board as it comes in the mail. It is there for the user to provide if she wants more control of the receiver gain.
Notice that one side of the resistor connects to pin 16.
Press enter or click to view image in full size
Here I have soldered a 30-gauge wire to the left solder point of where R3 would go.
Now we have the unprocessed signal from the receiver. We can feed it into an ESP32-C3 Super Mini's analog-to-digital converter and look at the signals.
Let's first look at it on the oscilloscope:
Press enter or click to view image in full size
Here I was moving closer to the board, causing a 3.329 Hz signal on pin 16.
The peak-to-peak voltage was 0.344 volts, well within the range of our ADC.
Now we can connect it to the Super Mini:
Press enter or click to view image in full size
We use pins 5, 6, and 7 on the Super Mini.
Pin 5 will be the ADC input.
Pin 6 we will set to HIGH to power the board.
Pin 7 will act as the ground for the board.
Here is some Micropython code for the Super Mini:

```
def main():
from machine import Pin, PWM, ADC
data_pin = Pin(5, Pin.IN)
power = Pin(6, Pin.OUT, Pin.DRIVE_3)
ground = Pin(7, Pin.OUT, Pin.DRIVE_3)
led = Pin(8, Pin.OUT)
ground.off()
power.on()
data = ADC(data_pin)
while(True):
```

```
# print(data.read())
if data.read() > 3000:
led.off()
else:
led.on()
main()
```

We have turned the device into a 3.16 GHz Morse Code receiver.

You may remember that I bought ten of these little boards. Some of that redundancy was my expectation of destroying one or two while soldering, but that didn't happen. Another reason for wanting more than one is to use one as a Morse Code transmitter:

Press enter or click to view image in full size

As you can see, I went all-out on the construction details. The code key acts as a switch, powering up the board from a 9-volt battery connected to VIN.

The red LED is the one I soldered to the board to use it as a motion detector. It still detects motion, but we ignore that.

Now, when I tap out Morse Code on the transmitter, I see the little blue LED on the Super Mini light up in sympathy.

Why did I use pins 6 and 7 to power the board instead of 3.3 volts and ground?

Because we can use the Super Mini to key the board. We can use the program we built in , and have it use Pin 6. We don't even need the extra transistor we used in that project.

Our 3.16 GHz transmitter has a wavelength of 94 millimeters. About the length of a cigarette. It travels through wood easily, but gets blocked by glass and metal.

Programming

Python

Radar

Radio Hackers: The SATNOGS Network(Pt 2)

Investigator515

Investigator515

Follow

7 min read

.

Mar 21, 2025

Listen

Share

More

Taking our ground station from cold and dark to a functional telemetry station.

We strive to provide informative articles, however, users need to ensure their research is both ethical and responsible. Additionally, it is your responsibility to ensure you're compliant with all applicable laws and regulations for your region. The information provided in this article is intended for educational purposes only.

In this previous article, we started looking at the Satnogs project. This project is an open-source, satellite ground station that collects data from in-orbit satellites. This data is then fed to the central Satnogs website where you'll be able to contribute your data right alongside the data of others. Overall, projects such as this aim to improve the efficiency of space-based assets by providing live, real-time data that can help monitor the health of spacecraft as they pass overhead.

Press enter or click to view image in full size

Satnogs aims to provide a live, real-time dashboard that helps monitor satellite health in real-time. Source: Satnogs.org

Let's Build

If you followed along for part one, you're probably already aware of the fact that to get things up

and running there’s a minimum equipment list. In case you missed it though, here’s what we’ll need to get started.

1x Raspberry Pi + Accessories

1x SDR Unit (HackRF, BladeRF, SDRPlay RTL-SDR etc)

1x. Antenna with Feedline.

You should also have an SD card that is preloaded with the Satnogs disk image. If you haven’t organised that yet, you can find it here

To ensure that we can still feed data correctly, we’ll also need to sign up for a Satnogs Account. This means we can configure and change our settings as needed, as well as set up the location of a new ground station feeder.

Press enter or click to view image in full size

To do this, we’ll simply visit the Libre Space community and register a new account as per the image above. From here, we can start to make changes to our station and apply the final configurations.

The Ground Station

Once we’re into our account and the email has been verified we can move to the next step, which is adding our ground station. There are a few ways that you can look at doing this and the block diagram below gives a great breakdown of how this might work and what we’ll need.

Press enter or click to view image in full size

If you aren’t using ground-based preamplifiers or antenna rotators then your own block diagram will look a little different to this. Remember though, you won’t need to have everything. While an omnidirectional antenna without a rotator has a distinct disadvantage in comparison to a tracked Yagi, you can still get started with something simple and carry out upgrades later on. Just remember that each time you add a change, update your details in the Satnogs portal so they remain current.

Let’s visit the portal now and add our new ground station. Then, we can get the configuration sorted and start sending data to the community.

When we’re in the portal, we’ll want to select the Add Ground Station tab as seen in the image below.

Once we’ve done that step, you’ll be taken to a fresh page where you’ll put in the technical information regarding how your station runs. Here, we’ll need to add info about our overall configuration that’s in use, a description of any antennas we might be running as well as any other relevant information.

Press enter or click to view image in full size

Most of this is pretty simple stuff, but there are a few things to be aware of. Firstly, an accurate location is required to ensure that your station is correctly integrated into the network. You’ll need to set the location in the “Advanced Edit” in the format of Lat/Long

Press enter or click to view image in full size

Once you’ve completed this step, the next is to add and configure an antenna. This is an important step, regardless of what type of antenna you might be using.

If you’re using separate antennas for each band, that isn’t a problem either. You will, however, have to add each antenna as well as the frequency range it covers. Check out the image below if you’re looking for some more insight on this.

Press enter or click to view image in full size

Web Portal Setup

The next step is ensuring that we’re able to feed data properly from our SDR unit into the Satnogs web portal. To do this, simply open your terminal and hit it with the following command

Sudo satnogs-setup

The Satnogs setup menu. Source: Wikipedia.com

If we’re using Raspbian to configure a system, typically we will need to run the usual update & upgrade commands. In this instance though, the Satnogs software is pretty efficient. By running the setup command, we’ve also instructed the system to fetch all relevant updates as needed.

Most of this configuration will happen automatically with little input needed from your good selves. However, there are a few things we'll need to double-check before we put things online.

Don't Forget This!

Satnogs works as a global feeder so to ensure it doesn't go hungry, we'll need to keep it fed with plenty of data. To do this, we'll need an API key so that we can feed it data via the backend.

Press enter or click to view image in full size

To set this up correctly, we'll need to visit the portal again and then acquire our key. First, we'll visit the dashboard, then hit the tab that says API Key.

Press enter or click to view image in full size

This will generate an API code that you can use to access and feed data to the web portal. Once you've completed this, work your way through the last of the setup steps. You can choose between basic and advanced configurations but if you're using a simple system with no antenna, it's probably better to stick with basic for now.

Press enter or click to view image in full size

Sticking with the basic configuration is the way to go for most users. Source: satnogs.org

The Test Run

When setup is complete and you've got your ground station up and running, you'll need to check to ensure that it shows up as online.

Once that's done, really there is only one thing left to do and that's to start testing by capturing packets!

It's a good idea to SSH into your Pi for this part as you'll be able to use the terminal to show what is happening and identify any other problems or issues. We've covered this before in earlier articles, so you should be good to gain access.

Press enter or click to view image in full size

When you have a satellite scheduled to arrive overhead, you should see your station going through the following motions as per the image above. If you're able to see this in your own terminal, then you can rest assured that everything is working as it should on your end as well.

Reminder: The best way to log in via SSH is by using a key instead of a password. If you must use a password though, ensure it is strong and unique.

The Backup

Pi's are great for low-cost devices to run receive stations, weather stations or what ever else you might like to build. However, they do come with one small flaw. In the interest of keeping costs low, the Pi relies on an SD card-based flash memory to run the operating system.

Unfortunately, these aren't optimised particularly well for long-term use and as such, it's not uncommon for the SD card to stop writing, at which point the device typically stops working.

Because of this, it's worth grabbing a backup of the image on the SD card so that the current card can be flashed should circumstances require it. You can do this by doing a straight duplication, however it's also wise to copy your configuration file.

Did You Build?

Once your station is assembled and tested, it should run fairly reliably. To optimise it though, you can look at changing antennas, adding a rotator for tracking or even using preamplifiers to help increase your signal strength.

You'll find some great resources in the Satnogs wiki that discusses these upgrades and these are particularly helpful in understanding exactly what kind of improvements each change will make.

Space communication can often be the pinnacle of weak signal work. Small transmitters, with small antennas in small satellites thousands of km away, mean that when you're stuck with omnidirectional antennas, the challenge is on.

You'd be surprised at the difference some small, well-thought-out changes can make to your received signal strength levels. Also, if you're building your own Satnogs ground station, don't forget to post in on socials with a tag to show off your build.

Stay tuned for part 3, where we look at some easy-to-construct antenna options to finalise your station.

■ We’re now on Bluesky!

Articles we think you’ll like:

What The Tech?! Space Shuttles

Shodan: A Map of the Internet

The Satnogs Project: Selecting Your Antenna (Pt 3)

Investigator515
Investigator515
Follow

6 min read

.

Jul 22, 2025

Listen

Share

More

The right antenna will determine the success (or failure) of your satellite receiving station.

If you aren’t a medium member, you can read with no paywall via substack

In this previous article, we took a look at some of the steps we’d need to cover to set up and configure a satellite receiving station that fed data to the Satnogs network as an independent node.

However, there are two distinct issues here that we’ll need to cover. Firstly, signals from space and more particularly cubesat signals tend to be quite weak. This is partly due to the distance travelled as well as the fact that on a cubesat board, space is at a premium (pardon the pun). This means that the output power levels are typically quite low.

The second problem is that the stock antenna design that comes as standard fitment to the RTL-SDR is absolutely awful. This is due to the fact that it tries to cover all bands effectively, meaning that for the most part, it isn’t good at covering any of them.

Press enter or click to view image in full size

The stock antenna is pretty useless for space communications. Be prepared to build. Source: ebay.com

Thankfully we can offset both of these problems by selecting & building a design that will be much more suitable for our new Satnogs station.

Broadband or Fixed

Part of the reason the stock antenna is so terrible is that it’s designed to be a broadband system. However, the Satnogs network will typically aim to collect data from satellites that have downlink frequencies within the 2 meter or 70 cm amateur radio bands.

So, rather than having to have broadband antennas optimised for entire RF bands, we can optimise our antenna for just 2 m or just 70cm.

Alternatively, if space is at a premium, we can even try our luck using a dual-band system, that means we can stick with a single antenna.

While the option is available to use tracked, directional antennas like Yagi antennas on a rotator, due to the complexity of such a system we’ll be sticking with well-designed, yet basic omnidirectional systems.

Press enter or click to view image in full size

Antenna Options

Now that steered yagis are eliminated, we can get to looking at our available options.

Thanks to the SAtnogs community, we have 3 options to choose from, with each varying slightly in terms of overall build difficulty.

If you’re after an easy option to get started, you might find the simple-to-make Turnstile antenna to be your best bet.

Press enter or click to view image in full size

If you’re willing to spend some extra effort and don’t mind working with coax, the Lindenblad might be a better option. You’ll need a phasing harness to make it work correctly, but you can make these yourself if you’re patient enough.

Press enter or click to view image in full size

The top shelf option, though, would be the Quadrafilar Helix Antenna or QFH. Providing great performance at weak signal levels, the QFH is a more advanced design that will put your homebrew skills to the test.

Making antennas can be a fun part of the radio journey. While antenna theory alone can and does fill entire books, you’ll find enough info to get you started in the Satnogs Wiki.

If At First You Don’t Succeed…Use Preamps

If you haven’t played with radios before, you might be surprised at just how much of an improvement adding a preamp gives. While tracked, high-gain antennas come with their own performance benefits, even systems like that can benefit from adding a preamp to the end design.

As you’d expect, though, these add both additional cost and complexity to any system, so they may not be suitable for users on a budget or in areas where easy access to parts is not relevant.

Remember, though, there’s no rule that says you need to build your entire station in one go. So, if you’d like, you can build yourself a basic system now and then add on extra components when you’re ready to.

One Last Thing

Now we have downloaded our image, configured our station, given it a great set of ears and then put it through testing to ensure the station works properly. We’re now well on the way to being able to put our station into service as a Satnogs feeder node.

There is one more thing you need to consider before we finish, though. In fact, some radio nerds might say it’s the most important part of the system, and it’s a topic we’ve covered in recent Radio Hackers articles.

While the choice of feedline might not be as important in a ground-based station, for our Satnog’s node, we’ll want to use a quality, low-loss feedline to ensure the signals move from antenna to receiver with minimal loss along the way.

Using a marginal coax like RG58 leaves us susceptible to signal loss levels that are unacceptable for a weak signal station. Aim to use a quality-built and well-insulated coax like LMR-400 to minimise your signal loss. Or, better still, omit the feed line and put your receiver at the antenna.

Press enter or click to view image in full size

Did You Build?

Have you explored the Satnogs project or built your own receive station? Maybe you’ve found another interesting radio project to experiment with using the RTL-SDR or have been hacking on a hardware project of your own.

Leave a comment and let us know what you’ve been working on lately. If it’s RF-based, why not consider writing it up and turning it into your own Radio Hackers feature article? We’re always looking for new writers to share their adventures and experiences!

■ We’re now on Bluesky!
Articles we think you’ll like:
What The Tech?! Space Shuttles
Shodan: A Map of the Internet
✉■ Want more content like this? Sign up for email updates
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 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/>

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

<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

LoRaWAN — Everything You Need to Know About The Global IoT Standard

What you should know before starting any LoRaWAN project.

Armando Rodrigues

Armando Rodrigues

Follow

6 min read

.

Oct 31, 2024

Listen

Share

More

Press enter or click to view image in full size

Blue and black logo, spelling LoRaWAN.

Source.

Trying to start a DIY IoT project but don’t know where to start? Or maybe just starting to learn about IoT technologies but getting overwhelmed with all the different protocols and standards? Don’t worry, I’ve got you covered! In this article I cover one of the most popular IoT standards among

hobbyist and professionals: LoRaWAN, a standard built on top of LoRa. Here you'll find some of the most important concepts on this topic. I did my best to present them in an easy-to-digest manner.

LoRaWAN Standard

LoRaWAN is a LPWAN (Low-Power Wide Area Network) standard maintained by the LoRa Alliance, a non-profit association with the mission of promoting the standard, which the organization claims to be the leading IoT LPWAN specification.

The LoRaWAN specification defines 3 things: the protocol stack, the regional parameters and the network architecture. The protocol stack comprises the LoRaPHY physical layer protocol and the LoRaWAN network protocol, which actually covers both the link and network layers. The network parameters specify important variables. Namely, channel frequency, channel bandwidth and the transmission windows for LoRaWAN communications all over the world. The network architecture standardizes the overall network structure, the function and naming of each device type and the relations and interactions between network nodes.

Network Architecture

Press enter or click to view image in full size

Diagram displaying network servers in the center gateways on the left and application servers on the right.

LoRaWAN network in a star topology.

LoRaWAN networks use a star topology. These networks consist of end-nodes/end-devices, gateways, network servers and application servers.

The working of gateways is explained in The Things Network (TTN) website:

Each gateway is registered (using configuration settings) to a LoRaWAN network server. A gateway receives LoRa messages from end devices and simply forwards them to the LoRaWAN network server. Gateways are connected to the Network Server using a backhaul like Cellular (3G/4G/5G), WiFi, Ethernet, fiber-optic or 2.4 GHz radio links.

Network servers manage the whole network. They filter duplicate messages received from several gateways, route uplink application payloads to the correct application server, provide acknowledgments of messages, send Adaptive Data Rate (ADR) messages, handle join requests and execute many more security- and network-related functions.

Application servers process application-layer payloads and generate downlink application messages (downlink Medium Access Control (MAC) commands, which are network and end-node control messag generated by network servers).

End-nodes are usually made up of microcontrollers connected to sensors, actuators, or both. These devices are usually battery-powered, have LoRa modulators and implement the LoRaWAN protocol sta in firmware. Every end-device must be registered with a network before sending and receiving messages. The process of joining a network is called "end-device activation". An end-node can be permanently tied to a pre-selected network using Activation By Personalization (ABP) or it can search and request to join a network using over-the-air activation (OTAA).

Devices that are part of a network with location-aware gateways can have GPS-free geolocation capabilities using a trilateration technique based on timestamps sent from the network.

LoRaWAN Network Protocol

Press enter or click to view image in full size

Diagram with a representation of LoRaPHY radio packets.

Representation of LoRaPHY radio packets, LoRaWAN link layer frames and network layer packets. The size of the physical radio packets is not specified because it is measured in symbols (not bytes) and it is highly variable as preamble and header length depends on the hardware (not on the LoRaWAN standard).

The LoRaWAN Network Protocol is the link and network layer protocol of the LoRaWAN protocol stack. Link layer frames are the payload of the physical layer LoRaPHY radio packets described in Subsection 2.1.2. Some frame types are used only during the join process of new end-nodes. However

the main type of link layer message, the one that is shown in Figure 2.5, contains three fields: a header, a payload and a message integrity code (MIC). Link layer frame headers include some information about the message type. Message types include confirmed and unconfirmed data message join and rejoin requests, MAC command and even proprietary message types, used to implement non-standard types. The payload contains the network layer packets. The MIC serves a different purpose than the CRC fields in the physical-layer protocol. Whilst CRC fields are used for error detection and correction, MIC fields are used for security purposes, allowing the receiver to determine if the message has been changed.
The link layer payload can be considered the network layer packet of the LoRaWAN specification. Still, the official documentation refers to these network layer packets as “MACPayload” frames, since the ISO/IEC 7498–1 model definitions of link and network layers do not strictly apply to LoRaWAN networks. These frames contain a frame header, an optional port field and an optional payload. The header contains the end-device device address, a frame counter, an adaptive data rate field and several other frame options fields. The port field allows for protocol testing and for future standardized application extensions. The payload contains the encrypted application-layer data.
Device Classes
Press enter or click to view image in full size
Diagram with a representation of LoRaWAN classes’ transmission windows
LoRaWAN classes’ transmission windows.
The LoRaWAN standard supports three device classes: class A (from “All End-Devices”), class B (from “Beacon”) and class C (from “Continuously Listening”). Device classes are backwards compatible, meaning that class C devices are class A and class B compatible and class B devices are class A compatible (but not class C compatible). The main difference between classes has to do with reception and transmission windows.
Class A focuses on obtaining extremely low power consumption at the cost of device availability. For devices from this class, the default state is a low-power one, such as a light sleep or even complete device hibernation. Essentially, class A end-nodes only wake for transmission, having two reception windows available after the end of transmission. The time interval between the transmission window and the reception windows is specified in the regional parameters’ specification of the LoRaWAN standard. Even though two reception windows are open after transmission, only one of them can actually be used for downlink messages, meaning that if a message is received in the first reception window the second one will not be available. Class A devices are very rarely available to receive downlinked commands from the network. Thus, most class A end-nodes are unidirectional or telemetry-oriented devices that broadcast data without accepting or accepting very few, commands. These include meters such as water, temperature, noise, current, voltage and light sensors or monitoring devices like parking, Time of Flight and motion sensors. Class A devices are commonly included in time-insensitive and loss-tolerant applications.
Class C is the complete opposite of class A. It focuses on device availability and neglects energy saving. Unlike most class A end nodes, devices from this class are usually connected to mains power and are not battery-powered. They are continuously available for reception, except during transmission. Class C end-nodes usually need to receive large amounts of data or several commands from downstream messages and are implemented in more time and response-sensitive applications.
Class B devices are an intermediate class that aims to achieve low power consumption while still offering frequent, but intermittent, reception windows, called ping slots. Devices from this class need to search and receive network beacons that allow synchronization with the network and avoid drifts between the device’s internal clock and network timing. A device might be unable to receive beacons due to being out of a gateway’s range, the network being down or interference. In such an event, class B devices gradually widen their reception windows to accommodate a possible drift of the device’s internal clock and to try to recover the network beacon. Class B devices ought to

operate “beaconless” for at least two hours before downgrading to class A operation, resetting or turning off.

The Bigger Picture — An Amazing Ecosystem for Hobbyists!

The LoRaWAN standard is part of a huge IoT ecosystem regulated by the LoRa Alliance. Companies a projects such as Meshtastic, The Things Network and many others provide everything from open-sourc and public to community-managed and fully private LoRaWAN networks. There are literally thousands o LoRa-compatible devices available on the market and the open-source nature of the standard makes it suitable for hobbyists. I myself have built my own LoRaWAN end nodes from the ground up and hope t document that journey in future Medium stories. Stay tuned for that!

This article is based on my master’s dissertation, where I explored the development of compact and efficient LoRaWAN end nodes with a focus on antenna miniaturization and power optimization for IoT applications.

All the diagrams in this article were created by the author.

IoT

Lorawan

Lora