Python Radio 7: The ESP32

Simon Quellen Field
Simon Quellen Field
Follow

3 min read
·
Aug 24, 2024
Listen
Share
More

More power!
Press enter or click to view image in full size
Image: Lilygo
The ESP32 is the big brother to the ESP8266.
It has two 32-bit processor cores on the chip, and a third less powerful processor for handling simple tasks while the more power-hungry cores sleep. It has over six times the RAM (520 KB) as the ESP8266 and runs at 240 MHz instead of 160 MHz. You can get one for about $12 at AliExpress.com.
Because it has several accessible hardware timers and UARTs (serial ports), it makes several radio projects possible that would be difficult to do with the smaller chip.
ASCII text by radio
The ESP32 has several UARTs (Universal Asynchronous Receiver-Transmitter) for sending serial data
This means we can use one for the USB connection to a laptop or desktop computer and still have fully functional serial data capabilities for sending bits to the radio. The ESP8266 only had the transmitting half of the serial port free for us to use.
We will use two ESP32 boards. The board we will call Ichabod will transmit messages to the board we will call Rumpelstiltskin.
We set up the UART to use pin 21 as output and pin 37 as input. If the machines are close together, a baud rate as high as 9600 will work, but for longer distances, a baud rate of 2400 works quite well, communicating without errors using our 433.92 MHz FS1000A at a distance of over 1,000 feet (probably much farther, but I ran out of room).
The program is very simple, since the hardware is doing all the work. We build a message and call uart.write() to send it, or we call uart.read() to read a byte stream from the UART, after which we convert it to a string and print it. Note that while uart.write() will accept a string, uart.read() returns a stream of bytes that have to be converted to utf-8.

```
From machine import UART
From name import name
TX = 21
RX = 37
Def main():
Uart = UART(1, baudrate=2400, tx=TX, rx=RX)
If name == “Ichabod”:
Count = 0
While True:
Msg = “UUU “ + str(count) + “: Hello, world!\r\n”
Print(”Sending”, msg, end=”)
Uart.write(msg)
Count += 1
While True:
Try:
M = uart.read()
```

If m:
Print(m.decode('utf-8'), end=")
Except UnicodeError:
Pass
Main()
It is convenient to have one simple main.py program for both boards, so I made a file called name.py and put the following line in it:
Name = "Rumpelstiltskin"
Before loading it onto one of the boards. Then I changed the line to
Name = "Ichabod"
And loaded it onto the other board. Now, as I make changes to main.py (such as changing the baud rate), I can just load the same file onto both boards.
One of my favourite ESP32 modules is the TTGO TDisplay, which has a built-in OLED display. The coc above and the diagrams below assume you are using that board, although it should be trivial to change the code for a different module.
The circuit is simple:
Press enter or click to view image in full size
Image by author
The receiver circuit looks like this:
Press enter or click to view image in full size
Image by author
In later projects, we will be using the power of the ESP32 to do a lot more work. Some radio modes require a good deal of computing power.
It is also nice to have a display to print things to.
Esp32
Micropython
Python Programming
Radio Communications
Texting
Python Radio 46: Satellites and Atomic Time
Simon Quellen Field
Simon Quellen Field
Follow
5 min read
·
Jun 30, 2025
Listen
Share
More
A GPS Disciplined NTP Server in Micropython
Press enter or click to view image in full size
A GPS satellite over North America.
MidJourney
For some weak signal protocols, such as WSPR, we need a clock that keeps good time. Anyone wishin to receive our signal knows to start receiving at an exact time, and if our clock doesn't match theirs, we can't communicate.
When we have an Internet connection, this is easy. We just use an NTP server to get accurate time from the web, and set our computer's clock by that.
But if we are camping out in the wilderness, we will need something else.
This project uses an inexpensive LilyGo T-Beam microcomputer running Micropython to collect GPS da

from the satellites and become an NTP server over its Wi-Fi radio for any other computers within range.
The T-Beam has a GPS receiver built in. It also has Wi-Fi. And becoming an NTP server is surprisingly easy.
Here is the code:

```
From machine import Pin, I2C, RTC, UART, freq
From time import sleep, sleep_ms, time
From network import WLAN, AP_IF
From socket import socket, AF_INET, SOCK_DGRAM
From struct import pack_into
From micropyGPS import MicropyGPS
Freq(240_000_000)
Last_rtc_sync_time = 0
RTC_SYNC_INTERVAL_SECONDS = 6 * 3600
NTP_PORT = 123
NTP_DELTA = 3155673600
Def setup_wifi_ap():
WIFI_SSID, WIFI_PASS = “T-Beam Time Server”, “gps-time-rocks”
Ap = WLAN(AP_IF)
Ap.active(True)
Ap.config(essid=WIFI_SSID, password=WIFI_PASS)
While not ap.active():
Sleep(1)
Print(f”--- Wi-Fi Access Point Started: SSID={WIFI_SSID}, IP={ap.ifconfig()[0]} ---“)
Def calculate_weekday(y, m, d):
If m < 3:
M += 12
Y -= 1
T = [0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4]
Day = (y + y // 4 – y // 100 + y // 400 + t[m – 1] + d) % 7
Return day if day != 0 else 7
Def set_rtc_from_gps(gps_object):
Global last_rtc_sync_time
Is_initial_sync = (last_rtc_sync_time == 0)
Time_since_last_sync = time() – last_rtc_sync_time
Should_sync = is_initial_sync or (time_since_last_sync > RTC_SYNC_INTERVAL_SECONDS)
If should_sync and gps_object.date[0] != 0:
Y, m, d = gps_object.date[2] + 2000, gps_object.date[1], gps_object.date[0]
H, mn, s = gps_object.timestamp[0], gps_object.timestamp[1], int(gps_object.timestamp[2])
Dt = (y, m, d, calculate_weekday(y, m, d), h, mn, s, int((gps_object.timestamp[2] – s) * 1e6))
RTC().datetime(dt)
Last_rtc_sync_time = time()
Msg = “Initial RTC sync” if is_initial_sync else “RTC re-synchronized”
Print(f”\nINFO: {msg}. Current Time: {y}-{m:02d}-{d:02d} {h:02d}:{mn:02d}:{s:02d} UTC\n”)
Def main():
GPS_RX=34
GPS_TX=12
Setup_wifi_ap()
Uart = UART(2, baudrate=9600, tx=GPS_TX, rx=GPS_RX, timeout=10)
My_gps = MicropyGPS(location_formatting=’dd’)
```

```
Ntp_socket = socket(AF_INET, SOCK_DGRAM)
Ntp_socket.bind((‘’, NTP_PORT))
Ntp_socket.setblocking(False)
Cnt = 0
Lat_avg = 0
Lon_avg = 0
Alt_avg = 0
While True:
If uart.any():
If uart_data := uart.read():
For char_byte in uart_data:
My_gps.update(chr(char_byte))
If my_gps.satellites_in_use > 0 and my_gps.latitude[0] != 0.0:
Set_rtc_from_gps(my_gps)
Lat_data, lon_data = my_gps.latitude, my_gps.longitude
Lat_val, lon_val = lat_data[0], lon_data[0]
If lat_data[1] == ‘S’:
Lat_val = -lat_val
If lon_data[1] == ‘W’:
Lon_val = -lon_val
Alt_val, sats = my_gps.altitude, my_gps.satellites_in_use
If lat_avg == 0: lat_avg = lat_val
If lon_avg == 0: lon_avg = lon_val
If alt_avg == 0: alt_avg = alt_val
If cnt < 20:
Cnt += 1
Lat_avg = (lat_avg * (cnt-1) + lat_val) / cnt
Lon_avg = (lon_avg * (cnt-1) + lon_val) / cnt
Alt_avg = (alt_avg * (cnt-1) + alt_val) / cnt
Feet = alt_avg * 3.280839895
Timestamp = f”{my_gps.timestamp[0]:02d}:{my_gps.timestamp[1]:02d}:{my_gps.timestamp[2]:02.0f}”
Payload_str = f”{lat_avg:.9f}, {lon_avg:.9f}, {alt_avg:.5f} meters ({feet:.5f} feet), UTC:
{timestamp}, {sats:3d} satellites”
Print(f”{payload_str}”)
Sleep_ms(300)
Else:
Print(f”Waiting for GPS fix... Sats: {my_gps.satellites_in_view}”)
Sleep(3)
If last_rtc_sync_time != 0:
Try:
Data, addr = ntp_socket.recvfrom(48)
If data:
Print(f”NTP Request from {addr[0]}”)
Recv_timestamp = time() + NTP_DELTA
Ntp_response = bytearray(48)
Ntp_response[0] = 0x24
Ntp_response[24:32] = data[40:48]
Secs, frac = int(recv_timestamp), int((recv_timestamp % 1) * (2**32))
Pack_into(‘!II’, ntp_response, 32, secs, frac)
Pack_into(‘!II’, ntp_response, 40, secs, frac)
```

```
Ntp_socket.sendto(ntp_response, addr)
Except OSError as e:
If e.args[0] != 11:
Print(f”NTP Socket Error: {e}”)
Main()
```

Starting in main(), we tell the program where to find the GPS receive and transmit pins. The GPS chip on the T-Beam connects to the UART at 9600 baud.

The MicropyGPS class knows how to parse the cryptic GPS sentences that come over the UART and t them into things we can use. More on that later.

Then we bind a socket to the NTP port 123 and set it to non-blocking. This is how we will talk to NTP clients.

The while loop looks for data on the UART, and hands each byte to the update() method of MicropyGP to parse and keep track of.

If the GPS has seen any satellites and has latitude data for us, we set the T-Beam real-time clock (RTC) from that data.

GPS latitude, longitude, and especially altitude are not perfectly accurate. We average the last 20 readings to get more precision.

Then we print out the data.

If we have synced our clock with the GPS satellite’s atomic clock, we are ready to serve NTP clients.

We get the request (if there is one) from the socket, and prepare our reply.

The setup_wifi_ap() routine is very simple, mostly boilerplate. You can set the SSID and password to suit your taste.

The set_rtc_from_gps() routine makes sure we don’t set our clock too often (which can cause jitter problems for some applications). In Micropython, the weekday is ignored, but we include it here for completeness.

The micropyGPS.py module comes from GitHub: . Download it and copy it to the T-Beam along with main.py.

When you boot the T-Beam, it starts printing to the Putty window (the terminal emulator I use on Windows):

```
37.203254700, -122.008956909, 487.72998 meters (1600.16394 feet), UTC: 22:54:26,  10 satellites
37.203254700, -122.008956909, 487.67343 meters (1599.97852 feet), UTC: 22:54:27,  10 satellites
37.203254700, -122.008956909, 487.61972 meters (1599.80225 feet), UTC: 22:54:27,  10 satellites
37.203254700, -122.008956909, 487.56870 meters (1599.63489 feet), UTC: 22:54:27,  10 satellites
37.203254700, -122.008956909, 487.51031 meters (1599.44336 feet), UTC: 22:54:28,  10 satellites
37.203254700, -122.008956909, 487.45483 meters (1599.26123 feet), UTC: 22:54:28,  10 satellites
NTP Request from 192.168.4.2
37.203254700, -122.008956909, 487.40210 meters (1599.08826 feet), UTC: 22:54:28,  10 satellites
37.203254700, -122.008956909, 487.34198 meters (1598.89099 feet), UTC: 22:54:29,  10 satellites
37.203254700, -122.008956909, 487.28491 meters (1598.70374 feet), UTC: 22:54:29,  10 satellites
37.203254700, -122.008956909, 487.23065 meters (1598.52576 feet), UTC: 22:54:29,  10 satellites
37.203254700, -122.008956909, 487.17914 meters (1598.35681 feet), UTC: 22:54:30,  10 satellites
NTP Request from 192.168.4.2
37.203254700, -122.008956909, 487.13019 meters (1598.19617 feet), UTC: 22:54:30,  10 satellites
37.203254700, -122.008956909, 487.08368 meters (1598.04358 feet), UTC: 22:54:30,  10 satellites
37.203254700, -122.008956909, 487.03949 meters (1597.89856 feet), UTC: 22:54:31,  10 satellites
37.203254700, -122.008956909, 486.99750 meters (1597.76086 feet), UTC: 22:54:31,  10 satellites
37.203254700, -122.008956909, 486.95761 meters (1597.63000 feet), UTC: 22:54:31,  10 satellites
37.203254700, -122.008956909, 486.90973 meters (1597.47290 feet), UTC: 22:54:32,  10 satellites
```

Notice how the latitude and longitude quickly converge, but the altitude is still fluctuating. In

addition to the location, the code prints out the Coordinated Universal Time (UTC). The acronym looks dyslexic to English speakers, but it matches most languages' abbreviations for Universal Time. UTC does not have daylight saving time, and reflects the time at the zero degree meridian, as it replaces the older Greenwich Mean Time (GMT).
In the printout above, you can see two NTP requests from my laptop computer. I logged into the T-Beam's Wi-Fi and got the address 192.168.4.2. The T-Beam's address is 192.168.4.1 (the default IP address for the access point server).
In Windows, you use the command net start w32time to start the NTP client on your machine. Then you type:

```
W32tm /config /manualpeerlist:"192.168.4.1" /syncfromflags:manual /reliable:yes /update
```

The T-Beam sees the request and sends the reply, and the NTP client on the laptop updates its clock. Now my laptop computer can be out in the woods and still get the correct time when the UTC system decides we need another leap second.

Space Weather: Imagery via Elektro-L3

Investigator515
Investigator515
Follow
6 min read
·
Aug 25, 2025
Listen
Share
More

The Geostationary Elektro satellites offer a far greater challenge than the POES satellites.
If you aren't a Medium member, you can read with no paywall via Substack
If you're an amateur meteorologist or satellite fan, there's no doubt that today, we are absolutely spoilt for choice when it comes to satellite programs to focus on. While the US-backed NOAA APT satellite program has recently come to an end, the evolution and development of both Chinese and Russian weather satellite programs have moved to fill the gap. While many of these programs aren't as simple as the earlier APT-equipped satellites, they provide a highly relevant platform that's still packed full of sensors and other meteorology equipment.
And, while there's plenty to explore in low-earth orbit, the real fun starts to come when you shift into deep space assets. With the Fengyun and Elektro series, there's plenty to be explored.
In today's article, we'll be taking a look at the Russian-designed Elektro satellite program and seeing what, if anything, it has to offer amateur satellite enthusiasts. Let's go take a look!
Press enter or click to view image in full size
The Elektro series would be Russia's first foray into deep-space Geostationary satellites after the fall of the Soviet Union. Here's an image of Elektro-L in the clean room before launch. Source: Wikipedia.

Background

With Sputnik, the first of many Soviet satellites to enter orbit during the height of the space race, you didn't have to be a genius to realise that this would change the dynamic forever.
As such, the USSR would attempt to capitalise on this immediately, and due to this, all manner of satellites would be launched. Of these, many would be considered to be dual use, in the sense that they would carry both military and civilian payloads.
This meant that civilian amateur radio payloads would fly on the back of nuclear-powered radar ocean reconnaissance satellites (RORSATs) while deep-space transmitters that would help deliver television to remote areas of the USSR would also carry signals intelligence equipment that was capable of attempting to geolocate US military assets like ships and aircraft carriers.
With the space launch technology of the time typically being single-use pieces of equipment like the

Soyuz, the dual nature of most military assets meant that the Soviets could squeeze out every little bit of capability for every system launched. While this would cause problems if a spacecraft were lost, it also meant that for every successful launch, there'd be a broad range of capabilities offered once systems were checked and deemed operational. As you'd imagine, this would lead to som truly unique spacecraft that are typically little-known to those outside amateur space or meteorology circles. The RORSAT program, for instance, contains some extremely interesting reading for those willing to peruse thousands of pages of declassified documentation.

Press enter or click to view image in full size

The Elektro's L-band data link provides a far greater challenge than the VHF downlink of the earlier Meteor-M series. Source: Wikipedia.

The Elektro Series

As technology progressed, we'd see many of these deep-space assets become more complex, but it w be the late 90s and early 2000s shrinking of electronic components that would offer true revolution in modern spacecraft operations and the Elektro series would be one of the first to leverage many of these new technologies.

When the Soviet Union collapsed, it would be the Russian Federation that would inherit the space program, including many high-profile systems like the Mir space station. There's no denying, though, that due to atrophy and lack of finances, the 90s-era space program would be on life support. What had been cutting-edge technology just a few years earlier was fast shifting into a program with dated, obsolete equipment. While there was little room in the budget for new military spacecraft, it would be hard to argue against the benefits that a new weather satellite program would bring. More importantly, the evolution of a new program would help the Russians maintain several vitally important Soviet-era skillsets.

So, with this said, the Elektro series would be born, and as such, the first Elektro L-1 satellite would fly in late 1994 as the first Russian-designed geostationary spacecraft since the collapse of the Soviet Union just a few years prior. As you'd imagine, this would be hugely noteworthy considering the political climate at the time.

While the program would start with just one satellite (Elektro L-1), the program would eventually evolve to include multiple satellites that would be positioned to give near-global, real-time coverage, a huge boost in service in comparison to assets that were available earlier.

Press enter or click to view image in full size

To grab a successful capture, you'll need a dish of at least 80cm in diameter, along with a preamp and frequency-capable software-defined radio unit like the SDRPlay RSP series. Here's a 1.5m prime focus dish for the C-band satellites. Source: Wikipedia.

Amateur Image Captures

While the amateur radio community would make plenty of noise about placing a Geostationary transponder system in orbit, thanks to satellites like the now-defunct AO-40, the reality was that placing niche, small-use systems into orbit was becoming harder as educational launch opportunities would begin to dry up. While several projects would eventually fly on both Russian and European systems, the writing would be on the wall regarding modern iterations of these types of systems, meaning that geostationary hams had little to attract their attention.

However, the launch of the Elektro birds would help to change this, and while they didn't have a usable transponder, one thing they did have was a clear beacon downlink, as well as the chance of capturing geostationary images of Earth from orbit. These systems would operate at around 1690.000MHz, just above the amateur 23cm band and at the beginning of the Super High Frequency (S microwave band.

It's fair to say, though, that while plenty of information would exist regarding the Very High Frequency systems of the POES and Meteor programs, little information was available regarding the Elektro series. This meant that for the most part, amateurs would have to go it alone, and as always, the community would step up.

Coming Soon
There’s no denying that while space can be difficult, the good propagation of VHF signals, as well as the wide array of available data regarding the systems, means that satellites like NOAA-19 and Meteor M2–4 are trivially easy to intercept data from.
With little more than some good software, a capable antenna and in some instances a preamp, most users are away, capturing high-quality images of Earth from space.
It’s fair to say that geo-stationary satellites have little in common with these simpler VHF systems. In fact, even experienced amateurs might find the Elektro satellites to be more of a challenge than expected. The sometimes fickle nature of propagation in the 1600MHz band means tha sometimes, establishing a system that works reliably can be an exercise in frustration.
Not only this, but any signals intercepted would be travelling tens of thousands of kilometres. So, while you might be able to get away with no preamp on a VHF system, for a microwave setup, the LNA is an essential piece of hardware.
Getting your head around this can be a bit tricky, but thankfully, the Elektro & Fengyun series of satellites will be the focus of Radio Hackers articles in the coming months, as we explore what you’ll need to reliably capture signals from these faraway Geostationary satellites. From start to finish, we’ll walk you through selecting a dish, creating a feed and configuring your software properly for use. More importantly, this is done in layman’s terms, meaning you won’t have to be across the jargon to correctly follow.
If you’ve ever had an interest in amateur meteorology or satellite communications, then stick around. Now is your time to shine.

Software Defined Radio & Radio Hacking: Space Communications (Part 3)
Investigator515
Investigator515
Follow
9 min read
·
Feb 6, 2024
Listen
Share

More
It’s time to spy on things that orbit the earth
This is multi part series. To catch up, read Part 1 and Part 2. Or find everything in our publication Radio Hackers.
As we’ve explored the world of Software Defined Radio, we started to explain some fundamentals around transmissions, antennas and receiving systems. It’s time to start putting these new pieces of knowledge to the test by looking at practical ways we can further develop these skills. In earlier articles, we promised a primer on space communications. Today, we deliver! But first, a quick update.
This series was far more popular than anticipated. Which we love, because it means there are plenty of inquisitive minds out there. Because of this, we’ve extended the series so we can focus on more in-depth tutorials around SDR over a longer time frame.
We’ve also launched a new publication where all these articles are posted, so you can streamline updates. Most of all, we’d love to hear about your journey. So if you’re using SDR we’d encourage you to share your journey by doing a write up and adding it to our publication. Contact us direct or drop a comment to arrange this.
The Status:
While exploring space-based assets can be pretty intriguing, it’s fair to say that if you’re a beginner to this world, the array of information to take in can be a little overwhelming. There’s frequency management, research of what to look at and where, as well as many other factors to be successful in your quest.
Today, we’ll look at intercepting communications from the International Space Station. To do this, we’ll have to look at ways to obtain orbital information, find out what frequencies and mode our target is operating with and put all that together to hopefully intercept either digital or voice communications from our space-based platform. Time to plan!
Planning the Event:
To successfully achieve our goal today there are a few critical pieces of information we’ll need. We need to know where our target is (location), when it will be there (timing), how we are to listen in on it (transmission mode) and where we are to listen to it (transmission frequency).
Our target, the International Space Station flies at altitudes of around 400 km above the earth in low earth orbit and moves through space at around 27,000 km/h. While you’ll typically find it will be in range for around 10 minutes or so as it orbits overhead, there’s a few factors that need to be accurate for this to be successfully intercepted. If you’re a beginner, it may take more than one attempt to pull this off, so be persistent.
Now we know what we need to give us the best shot at success, let’s look at how we obtain the data so we can get ready for our pass.
Orbital Information:
In the early days prior to the proliferation of computers and the internet, tracking space-based assets was a niche hobby due to the knowledge requirements of doing so. Over time, the implementation of applications and web interfaces has evolved to give us many pieces of information directly to our device of choice. This means that two of the factors we’ll need to gather information can be met by obtaining orbital information for the ISS over a particular area.
If we wish to do this on a mobile device, our recommendation it to use a mobile app like ISS detector. It’s available on both Apple and Android devices and will give you real-time orbital information as well as advanced future orbits a few days ahead of time. Find ISSDetector on the Play Store via this link.
If you’d prefer to use your laptop that has your SDR device connected you’ll need a computer-based tracking app. There’s an array of programs for doing so, but Gpredict works well for Linux and Windows. Find the Windows version here.
If you’re a Linux user, you can install it with the APT package manager with this command

Apt install gpredict
Once we’ve got our orbital tracker up and running, we can look at it to extract our information that we’ll need to plan our pass. We can see that Gpredict will calculate our position, and then give us a forecast on both timing and orbital inclination relevant to that position. We can also receive mode and frequency information for our transponders, meaning that by implementing the correct tracking suite we obtain all the information needed for our our planning purposes.
Press enter or click to view image in full size
G-predict will do much of the math for you. Source: Author
One relevant point to consider is that inclination will partly dictate how well we receive our transmissions. An orbital inclination of 10 degrees, will provide far weaker signal strength and interception possibility than a 50 or 60-degree inclination. In the early days, while learning, it’s best to focus on the high inclination passes to make things easier as a stronger signal will always be far easier to intercept than something that’s further away.
Press enter or click to view image in full size
There’s a lot of information here, but you’ll only need orbital inclination and timing for now. Source: Author.
In our example calculations we can see our pass is going to peak at an inclination of 51 degrees at 0351UTC on the 27th. We’ve now obtained information for two of our goals. Where it will be. And when it will be there? Let’s look at obtaining what we need to meet our last two.
There’s a vast communications suite onboard but in the interests of streamlining things, we’ll focus on transmissions from the onboard Automatic Packet Reporting System (APRS) as we can then decod
them later on using a plugin.
Transmission Types & Frequencies:
To achieve our last two requirements we need to know frequency information and type and we can obtain this from Gpredict as well.
Press enter or click to view image in full size
Frequency information is easy to find in your tracker of choice. Source: Author
As we see, there’s a vast array of types to explore. Feel free to experiment with your SDR dongle but for today we’ll be looking at Mode V APRS as mentioned.
Press enter or click to view image in full size
Mode V (VHF FM) is what we’re looking for. It’s there on 145.825mhz
We can see in our attached image that the APRS transmitter has a downlink of 145.825mhz, the middl
of the VHF amateur band and uses FM modulation. We also note that there’s a baud rate there, as it’s a digital transmission. We’ll focus on decoding that later on, but this data gives us our transmission frequency and type.
So to clarify this, for successful interception of the APRS transmitter, we’ll need to tune our SDR to receive the downlink on 145.825 as well as keep our mode in FM as well. So when we’re configuring our SDR station to receive these transmissions, you’ll know what to look for when the signal pops up on the waterfall.
One Last Caveat:
One last item to check before you attempt your pass is the current status of the transmitter. Often, the system will be turned off during docking operations or maintenance periods, meaning that despite being overhead no transmissions will be heard. Mitigate this by checking the status of the station via this website. You can also find information via various social media accounts.
Press enter or click to view image in full size
During the Pass:
It’s good practice to have your station set up a few minutes ahead of the pass to ensure any problems are uncovered prior to the pass commencing.
While SDR clients will vary, you’ll typically have access to a waterfall-type display and have the ability to record transmissions. Don’t forget to switch the record on prior to the pass so you’re

able to log received transmissions into a saved data file. You’ll need this file for a future article so we can show you how to decode it using a plug-in.
To get a feel for how the transmitter sounds and works so you know to expect, watch this video for an audible explanation.
And we’ve included a screen shot below so you know what to look for on your waterfall display.
Short, sharp and effective. The transmission is strong and clear in the waterfall display. Source: Author
As we emphasized earlier it’s important to understand that signal strength and quality will vary depending on your station, distance of the ISS and even atmospheric conditions in some instances. To successfully receive you may need to troubleshoot your station. Generally, you’ll receive the strongest signal when the station is at orbital peak, so with a properly configured station you should be receiving strong signals for at least a short period.

Doppler: One Quick Word

The Doppler effect, when applied to space communications, is a phenomenon in which the frequency o electromagnetic waves, such as radio signals, appears to change as the source (e.g., a satellite or spacecraft) and the receiver (e.g., a ground station) move relative to each other. When an object in space moves towards the observer, the waves get compressed, causing a higher observed frequency (upshift), while when the object moves away, the waves get stretched, resulting in a lower observed frequency (downshift). This effect is crucial in space communication as it must be accounted for when calculating signal frequencies to ensure accurate and reliable data transmission between spacecraft and ground stations, especially when high velocities are involved. While we won’t cover that in-depth today, it’s an important concept to understand for future projects where it will be relevant.
Press enter or click to view image in full size
Doppler shift varies according to frequency, with higher shifts noticeable as frequency increases. Source: Wikipedia

In Closing:

As we bring today’s article to a close let’s do a quick recap on what we’ve learnt. At this point, we are starting to understand the concepts behind tracking an object in space and what we need to do to estimate a pass schedule for any location. We also understand that we need to know our desired frequency and transmission type for interception purposes. And to future-proof our skills as we develop, we’ve introduced the concept of the Doppler effect on frequencies as well as having a rough idea of how orbital inclination and direction can affect our chances of properly receiving communications. Lastly, we’ve also looked at how to determine the status of our target and how we are able to check if the station is up for communication or not.
We’ll finish by including one last bonus point. Information is key to anything, particularly in the fields of information / cyber security and open-source intelligence. When we apply this to information gathering and radio hacking that revolves around space-based assets we soon realize that there’s a vast array of various types of satellites that are available for use. While we can look at the frequency band plan and focus the allocations for space communications, there is a better way. The best way to look for this information is to look for catalogues on satellites and space assets and cross-check them to obtain information as needed. NASA has a vast array of information that’s publicly accessible for this and there are many civilian resources as well. One of the best non-official sites for tracking this information is a site managed by a radio amateur. It has Launch, Frequency and Catalogue information for nearly all space-based assets, and includes recordings so you’ll know what to listen for. DD1US Sounds From Space is a valuable resource for those who are interested in pursuing space-based assets further.
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? Support our work and join the community!
Support me on Ko-fi: Investigator515
Join our Telegram channel for exclusive updates or.
Follow us on Twitter
Articles we think you'll like:
OSINT Unleashed: 5 Essential Tools for Cyber Investigators
What the Tech?! GPS Technology
✉■ Want more content like this? Sign up for email updates here
Space
Osint
Hacking
Information Security
Radio
Python Radio 20: The CC1101 Module
Follow
12 min read
·
Sep 7, 2024
Listen
Share
More
Half a megabit per second over a kilometer.
Press enter or click to view image in full size
Photo by the author
The CC1101 is a very flexible sub-gigahertz transceiver. It can transmit and receive in three wide frequency ranges: 300 to 348 MHz, 387 to 464 MHz, and 779 to 928 Mhz. That middle range includes t
European license-free ISM band (433.05 MHz to 434.79 MHz), as well as the U.S. Amateur Radio 70 c
band (420 to 450 MHz). That means that with an Amateur Radio license, you can amplify the CC1101's
10-milliwatt output to as much as 50 watts (but as most communication in this band is line-of-sight, 5 watts is usually more than enough).
The last band includes the European 868 MHz license-free ISM band (863 MHz to 870 MHz) and U.S. 9
Mhz license-free ISM band (902 MHz to 928 MHz).
10-milliwatts can reach a kilometer between two CC1101's in the open with good antennas placed high
above the ground.
Modules containing the chip are usually limited to one of the three ranges. In this section, we will use the 433 MHz version that can reach the U.S. Amateur Radio frequencies.
The module is programmed using the SPI (Serial Peripheral Interface), which needs 5 pins (power, ground, clock, input, and output) as well as a chip select pin, and two general purpose pins called GDO0 and GDO2.
With 8 pins to worry about, this is already one of our most complicated modules. But it doesn't stop there. There are 47 configuration registers, 13 status registers, and many modes and functions.
The chip can support synchronous and asynchronous serial modes up to half a megabit, and packetize
modes with cyclic redundancy checks, preambles, sync words, forward error correction, interleaving, and more.
Press enter or click to view image in full size
Image by the author
Because of this complexity, even something as simple as our Morse code transmitter and receiver

takes quite a bit of configuring.
The code for the main.py module sets up the SPI interface and is divided into two sections we will call “alice” and “bob”:

```
from machine import SoftSPI, SPI, Pin, PWM
from cc1101 import CC1101
from whoami import whoami
from whoami import my_address
from time import sleep
def main():
global radio
spi = SoftSPI(baudrate=200_000, sck=Pin(2), mosi=Pin(3), miso=Pin(4), firstbit=SPI.MSB)
print(”I am”, whoami)
if whoami == “alice”:
from morse import Morse
gdo0 = Pin(17, Pin.OUT)
gdo2 = Pin(18, Pin.OUT)
cs   = Pin( 5, Pin.OUT)
radio = CC1101( spi, cs, gdo0, gdo2, 433_920_000 )
morse = Morse(radio)
morse.speed(20)
radio.transmit()
while True:
morse.send(”Hello, world! This is AB6NY sending via a cc1101 at 10 milliwatts.”)
sleep(1)
elif whoami == “bob”:
gdo0 = Pin(17, Pin.OUT)
gdo2 = Pin(18, Pin.IN)
cs   = Pin( 5, Pin.OUT)
radio = CC1101( spi, cs, gdo0, gdo2, 433_920_000 )
radio.receive()
speaker = PWM(Pin(13), freq=800, duty_u16=0)
while True:
if gdo2.value():
speaker.duty_u16(32768)
else:
speaker.duty_u16(0)
sleep(60 * 60 * 24 * 365 * 100)   # Should be long enough
main()
```

Alice is the transmitter. All of the pins are outputs.
Bob is the receiver. The GDO2 pin is an input and will go high when the CC1101 detects a carrier from Alice. When it does, Bob will send a square wave to the speaker attached to pin 13, and the user will hear an 800-hertz tone.
The morse.py module is only slightly changed. It simply calls the on() and off() methods of the radio module.

```
class Morse:
def __init__(self, radio):
self.radio = radio
self.character_speed = 5
def speed(self, overall_speed):
self.character_speed = overall_speed
```

```
units_per_minute = int(self.character_speed * 50)        # The word PARIS is 50 units of time
OVERHEAD = 2
self.DOT = int(60000 / units_per_minute) - OVERHEAD
self.DASH = 3 * self.DOT
self.CYPHER_SPACE = self.DOT
self.LETTER_SPACE = int(3 * self.DOT) - self.CYPHER_SPACE
self.WORD_SPACE = int(7 * self.DOT) - self.CYPHER_SPACE
def send(self, str):
from the_code import code
from time import sleep_ms
for c in str:
if c == ‘ ‘:
self.radio.off()
sleep_ms(self.WORD_SPACE)
else:
cyphers = code[c.upper()]
for x in cyphers:
if x == ‘.’:
self.radio.on()
sleep_ms(self.DOT)
else:
self.radio.on()
sleep_ms(self.DASH)
self.radio.off()
sleep_ms(self.CYPHER_SPACE)
self.radio.off()
sleep_ms(self.LETTER_SPACE)
```

Our the_code.py module has not changed.

As you might expect, most of the complexity resides in the cc1101.py module:

```
from time import sleep, sleep_ms, sleep_us
from machine import Pin, SPI
class StrobeAddress():
SRES = 0x30
SFSTXON = 0x31
SXOFF = 0x32
SCAL = 0x33
SRX = 0x34
STX = 0x35
SIDLE = 0x36
SWOR = 0x38
SPWD = 0x39
SFRX = 0x3A
SFTX = 0x3B
SWORRST = 0x3C
SNOP = 0x3D
class StatusRegisterAddress:
PARTNUM = 0xF0         # Part number for CC1101
VERSION = 0xF1         # Current version number
FREQEST = 0xF2         # Frequency Offset Estimate
LQI = 0xF3             # Demodulator estimate for Link Quality
```

```
RSSI = 0xF4           # Received signal strength indication
MARCSTATE = 0xF5       # Control state machine state
WORTIME1 = 0xF6        # High byte of WOR timer
WORTIME0 = 0xF7        # Low byte of WOR timer
PKTSTATUS = 0xF8       # Current GDOx status and packet status
VCO_VC_DAC = 0xF9      # Current setting from PLL calibration module
TXBYTES = 0xFA         # Underflow and number of bytes in the TX FIFO
RXBYTES = 0xFB         # Overflow and number of bytes in the RX FIFO
RCCTRL1_STATUS = 0xFC  # Last RC oscillator calibration result
RCCTRL0_STATUS = 0xFD  # Last RC oscillator calibration result
class ConfigurationRegisterAddress:
IOCFG2 = 0x00         # GDO2 output pin configuration
IOCFG1 = 0x01         # GDO1 output pin configuration
IOCFG0 = 0x02         # GDO0 output pin configuration
FIFOTHR = 0x03        # RX FIFO and TX FIFO thresholds
SYNC1 = 0x04          # Sync word, high byte
SYNC0 = 0x05          # Sync word, low byte
PKTLEN = 0x06         # Packet length
PKTCTRL1 = 0x07       # Packet automation control
PKTCTRL0 = 0x08       # Packet automation control
ADDR = 0x09           # Device address
CHANNR = 0x0A         # Channel number
FSCTRL1 = 0x0B        # Frequency synthesizer control
FSCTRL0 = 0x0C        # Frequency synthesizer control
FREQ2 = 0x0D          # Frequency control word, high byte
FREQ1 = 0x0E          # Frequency control word, middle byte
FREQ0 = 0x0F          # Frequency control word, low byte
MDMCFG4 = 0x10        # Modem configuration
MDMCFG3 = 0x11        # Modem configuration
MDMCFG2 = 0x12        # Modem configuration
MDMCFG1 = 0x13        # Modem configuration
MDMCFG0 = 0x14        # Modem configuration
DEVIATN = 0x15        # Modem deviation setting
MCSM2 = 0x16          # Main Radio Control State Machine configuration
MCSM1 = 0x17          # Main Radio Control State Machine configuration
MCSM0 = 0x18          # Main Radio Control State Machine configuration
FOCCFG = 0x19         # Frequency Offset Compensation configuration
BSCFG = 0x1A          # Bit Synchronization configuration
AGCTRL2 = 0x1B        # AGC control
AGCTRL1 = 0x1C        # AGC control
AGCTRL0 = 0x1D        # AGC control
WOREVT1 = 0x1E        # High byte Event 0 timeout
WOREVT0 = 0x1F        # Low byte Event 0 timeout
WORCTRL = 0x20        # Wake On Radio control
FREND1 = 0x21         # Front end RX configuration
FREND0 = 0x22         # Front end TX configuration
FSCAL3 = 0x23         # Frequency synthesizer calibration
FSCAL2 = 0x24         # Frequency synthesizer calibration
FSCAL1 = 0x25         # Frequency synthesizer calibration
FSCAL0 = 0x26         # Frequency synthesizer calibration
```

```
RCCTRL1 = 0x27         # RC oscillator configuration
RCCTRL0 = 0x28         # RC oscillator configuration
FSTEST = 0x29          # Frequency synthesizer calibration control
PTEST = 0x2A           # Production test
AGCTEST = 0x2B         # AGC test
TEST2 = 0x2C           # Various test settings
TEST1 = 0x2D           # Various test settings
TEST0 = 0x2E           # Various test settings
class PatableAddress:
PATABLE = 0x3E
class FIFORegisterAddress:
TX = 0x3F
RX = 0x3F
patable_power_433 = [0x00,0x6C,0x6C,0x6C,0x6C,0x6C,0x6C,0x6C]
WRITE_SINGLE      = 0x00
WRITE_BURST       = 0x40
READ_SINGLE       = 0x80
READ_BURST        = 0xC0
IDLE_STATE        = 0
RX_STATE          = 1
TX_STATE          = 2
FSTXON_STATE      = 3
CAL_STATE         = 4
SETTLING_STATE    = 5
RXOVER_STATE      = 6
TXUNDER_STATE     = 7
class SPIDevice:
def __init__(self, spi, cs):
self.buf = bytearray(1)
self.spi = spi
self.cs = cs
self.state = IDLE_STATE
def reg_cmd_strobe(self, reg):
self.cs(0)
self.spi.readinto(self.buf, reg & 0x3F)
self.cs(1)
sleep_ms(1)
self.get_status(self.buf[0])
return self.buf[0]
def reg_read_bytes(self, reg, buf):
self.cs(0)
self.spi.readinto(buf, READ_BURST | reg)
self.spi.readinto(buf)
self.cs(1)
sleep_ms(1)
return buf
def reg_write(self, reg, value):
self.cs(0)
self.spi.readinto(self.buf, WRITE_SINGLE | reg)
ret = self.buf[0]
```

```
self.get_status(ret)
self.spi.readinto(self.buf, value)
self.cs(1)
sleep_ms(1)
return ret
def reg_write_bytes(self, reg, buf):
self.cs(0)
self.spi.readinto(self.buf, WRITE_BURST | reg)
self.get_status(self.buf[0])
self.spi.write(buf)
self.cs(1)
sleep_ms(1)
def reset(self):
self.cs(0)
sleep_ms(100)
self.cs(1)
sleep_ms(100)
status_byte = self.reg_cmd_strobe(StrobeAddress.SRES)
sleep_ms(100)
self.get_status(status_byte)
sleep_ms(1)
def get_status(self, status_byte):
self.ready = True
if 0x80 & status_byte:
self.ready = False
s = (0x70 & status_byte) >> 4
if   s == 0: self.state = IDLE_STATE
elif s == 1: self.state = RX_STATE
elif s == 2: self.state = TX_STATE
elif s == 3: self.state = FSTXON_STATE
elif s == 4: self.state = CAL_STATE
elif s == 5: self.state = SETTLING_STATE
elif s == 6: self.state = RXOVER_STATE
elif s == 7: self.state = TXUNDER_STATE
sleep_ms(1)
def read_status_reg_and_check(self, reg):
ret = bytearray(1)
check = bytearray(1)
while True:
self.reg_read_bytes(reg, ret)
self.reg_read_bytes(reg, check)
if ret == check:
break
status_byte = self.reg_cmd_strobe(StrobeAddress.SNOP)
self.get_status(status_byte)
return ret[0]
class CC1101:
def __init__(self, spi, cs, gdo0, gdo2, frequency, catch0=None, catch2=None):
self.gdo0 = gdo0
self.gdo2 = gdo2
self.device = SPIDevice(spi, cs)
self.device.reset()
self.device.reg_cmd_strobe(StrobeAddress.SIDLE)
sleep_us(800)
self.device.reg_cmd_strobe(StrobeAddress.SFRX)                              # flush the
RX buffer
self.device.reg_cmd_strobe(StrobeAddress.SFTX)                              # flush the
TX buffer
self.device.reg_write(ConfigurationRegisterAddress.IOCFG2,   0x0D)
self.device.reg_write(ConfigurationRegisterAddress.IOCFG0,   0x0D)
self.device.reg_write(ConfigurationRegisterAddress.FIFOTHR,  0x47)
self.device.reg_write(ConfigurationRegisterAddress.PKTCTRL0, 0x32)
self.device.reg_write(ConfigurationRegisterAddress.FSCTRL1,  0x06)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG4,  0xF5)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG3,  0x75)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG2,  0x30)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG1,  0x72)
self.device.reg_write(ConfigurationRegisterAddress.DEVIATN,  0x14)
self.device.reg_write(ConfigurationRegisterAddress.MCSM0,    0x18)
self.device.reg_write(ConfigurationRegisterAddress.FOCCFG,   0x16)
self.device.reg_write(ConfigurationRegisterAddress.WORCTRL,  0xFB)
self.device.reg_write(ConfigurationRegisterAddress.FREND0,   0x11)
self.device.reg_write(ConfigurationRegisterAddress.FSCAL3,   0xE9)
self.device.reg_write(ConfigurationRegisterAddress.FSCAL2,   0x2A)
self.device.reg_write(ConfigurationRegisterAddress.FSCAL1,   0x00)
self.device.reg_write(ConfigurationRegisterAddress.FSCAL0,   0x1F)
self.device.reg_write(ConfigurationRegisterAddress.TEST2,    0x81)
self.device.reg_write(ConfigurationRegisterAddress.TEST1,    0x35)
self.device.reg_write(ConfigurationRegisterAddress.TEST0,    0x09)
self.device.reg_write(ConfigurationRegisterAddress.CHANNR,   0x00)
self.device.reg_write_bytes(PatableAddress.PATABLE, bytearray(patable_power_433))
self.set_frequency(frequency)
self.device.reg_cmd_strobe(StrobeAddress.SCAL)
sleep_us(800)
if catch0:
self.gdo0.irq(catch0, trigger=(Pin.IRQ_FALLING | Pin.IRQ_RISING))
if catch2:
self.gdo2.irq(catch2, trigger=Pin.IRQ_FALLING)
def get_RSSI(self):
ret = bytearray(1)
self.device.reg_read_bytes(StatusRegisterAddress.RSSI, ret)
return ret[0]
def set_frequency(self, frequency):
frequency_hex = hex(int(frequency * (65536 / 26_000_000)))
byte2 = (int(frequency_hex, 16) >> 16) & 0xff
byte1 = (int(frequency_hex) >>  8) & 0xff
byte0 = int(frequency_hex) & 0xff
self.device.reg_write(ConfigurationRegisterAddress.FREQ2, byte2)
self.device.reg_write(ConfigurationRegisterAddress.FREQ1, byte1)
self.device.reg_write(ConfigurationRegisterAddress.FREQ0, byte0)
```

```
def transmit(self):
self.device.reg_cmd_strobe(StrobeAddress.SIDLE)
sleep_us(800)
self.device.reg_cmd_strobe(StrobeAddress.SCAL)
sleep_us(800)
while self.device.state != IDLE_STATE:
self.device.reg_cmd_strobe(StrobeAddress.SNOP)
while self.device.state != TX_STATE:
status_byte = self.device.reg_cmd_strobe(StrobeAddress.STX)                    ### Start
transmitting
self.device.read_status_reg_and_check(StatusRegisterAddress.TXBYTES)           ### Won't transmit
without this, don't know why
if self.device.state == TXUNDER_STATE:
status_byte = self.device.reg_cmd_strobe(StrobeAddress.SFTX)
txBytes = self.device.read_status_reg_and_check(StatusRegisterAddress.TXBYTES)
while self.device.state != IDLE_STATE and txBytes > 0:
txBytes = self.device.read_status_reg_and_check(StatusRegisterAddress.TXBYTES)
self.device.reg_cmd_strobe(StrobeAddress.SNOP)
if self.device.state == TXUNDER_STATE:
status_byte = self.device.reg_cmd_strobe(StrobeAddress.SFTX)
sleep_us(100)
def receive(self):
self.device.reg_cmd_strobe(StrobeAddress.SIDLE)
sleep_us(800)
self.device.reg_cmd_strobe(StrobeAddress.SCAL)
sleep_us(800)
while self.device.state != RX_STATE:
status_byte = self.device.reg_cmd_strobe(StrobeAddress.SRX)
cnt = self.device.read_status_reg_and_check(StatusRegisterAddress.RXBYTES)
if self.device.state == RXOVER_STATE or (cnt & 0x80):
self.device.reg_cmd_strobe(StrobeAddress.SFRX)
sleep_us(100)
def on(self):
self.gdo0.value(1)
def off(self):
self.gdo0.value(0)
```

The addresses of the 13 commands are found in the StrobeAddress class, and the addresses of the 1
status registers are seen in the StatusRegisterAddress class. The 47 configuration registers are in the ConfigurationRegisterAddress class. Two other classes hold the address of the 8-byte Power Amplifier table, and the address of the FIFO buffer for transmitting and receiving up to 64 bytes.

The SPIDevice class is used to send and receive data between the microprocessor and the CC1101 module. It handles setting and resetting the Chip Select pin, getting the status byte returned from commands, and details of timing.

The CC1101 class is the device driver for the module. It resets the CC1101, flushes anything in the transmit and receive buffers, and sets a number of configuration registers to set up the chip to send an unmodulated (CW) signal. Texas Instruments, the company that designed the chip, has free software for setting up all of these registers. The software is called the SmartRF Studio.

The Power Amplifier table determines the output power for each of 8 parts of each bit to be sent. By shaping the amplitude of a bit in this way, the transmitter can avoid sending out power into unwanted sidebands and thus interfering with other radios on nearby channels. Our PATABLE doesn't

use this feature (since we aren’t sending bits), so it has zero power in the first byte and 0x6C (full power) in the seven ramaining bytes.
It then sets the frequency and calibrates the oscillator. We don’t use the catch0 and catch2 arguments when sending and receiving CW.
The transmit() and receive() methods set the module into those respective modes. This process involves setting the chip into the IDLE state, calibrating the oscillator, sending the STX or SRX command, and waiting for any pending bytes from previous commands to be processed (there won’t be any, since we are sending CW, not bits and bytes). It also flushes the FIFO buffers if there was an error condition (there won’t be in CW).
Finally, the on() and off() methods control whether the transmitter is transmitting or not by sending a signal on the GDO0 pin.
Altogether, almost 300 lines of code just to turn the transmitter on and off. While the module is capable of doing this job, it is not what it was designed for. It wants to send bytes and packets, and at much higher speeds. Let’s let it do that.
The RP2040’s UART can send bytes at just under a megabit per second (961.6 kBaud). Our CC1101 c manage half a megabit (500 kBaud) in MSK mode and a quarter megabit (250 kBaud) in GFSK mode. location, I was getting occasional interference at the highest baud rate from some nearby transmitter (the 433 MHz band is shared with lots of different devices), but at 250 kBaud I was getting no errors at all after the first message was sent (the first message accumulates a lot of noise as the receiver waits for the transmitter to begin).
The main.py module for sending UART bits through the CC1101 looks like this:

```
from machine import SoftSPI, SPI, Pin, UART
from cc1101 import CC1101
from whoami import whoami
from whoami import my_address
from time import sleep
def main():
global radio
spi = SoftSPI(baudrate=200_000, sck=Pin(2), mosi=Pin(3), miso=Pin(4), firstbit=SPI.MSB)
print(”I am”, whoami)
baud = 250_000
if whoami == “alice”:
gdo0 = Pin(8, Pin.OUT)
gdo2 = Pin(18, Pin.OUT)
cs   = Pin( 5, Pin.OUT)
radio = CC1101( spi, cs, gdo0, gdo2, 433_920_000, baud )
serial = UART(1, baudrate=baud, tx=gdo0, rx=Pin(9, Pin.IN))
radio.transmit()
count = 0
preamble = “UUUUABCD”
while True:
serial.write(preamble + str(count) + “: Hello, world! This is AB6NY sending via a cc1101 at 10
milliwatts.\n”)
count += 1
sleep(1)
elif whoami == “bob”:
gdo0 = Pin(17, Pin.OUT)
gdo2 = Pin(9, Pin.IN)
cs   = Pin( 5, Pin.OUT)
radio = CC1101( spi, cs, gdo0, gdo2, 433_920_000, baud )
```

```
serial = UART(1, baudrate=baud, tx=Pin(8), rx=gdo2)
radio.receive()
while True:
if serial.any():
s = serial.read()
try:
msg = s.decode('utf-8')
index = msg.find("ABCD")
if index > 0:
print(msg[index+4:], end='')
#        else:
#          print("No sync:", msg, end='')
#      except:
#        print("Not utf-8:", s, end='')
main()
```

We have added an argument to the CC1101 driver: it now needs to know the baud rate. The transmitte (Alice) sets the UART tx pin to the same pin as GDO0. Alice does not care about the UART receive pin, but sets it to 9 anyway.

The preamble and sync word are probably not necessary for most baud rates, but I found it useful for the 500 kBaud rate, as the first bits of the message were often corrupted. The preamble is just a set of alternating zero and one bits to help synchronize the receiver. That is the four capital U characters. The ABCD is a synchronization sequence to tell us where the real data payload is. In packet modes, the preamble synchronizes at the bit level, and the sync word aligns the bytes.

The receiver (Bob) sets the UART receive pin to the same as GDO2. The CC1101 thus uses GDO0 for in and GDO2 for data out. The UART (of course) sends on GDO0 and receives on GDO2.

Bob waits for serial data to be available, and then reads it. If it is uncorrupted utf-8 and the sync word is found, it prints the payload.

The cc1101.py module's only changes are to the PATABLE and the __init__() method:

```
class CC1101:
def __init__(self, spi, cs, gdo0, gdo2, frequency, baud, catch0=None, catch2=None):
self.gdo0 = gdo0
self.gdo2 = gdo2
self.device = SPIDevice(spi, cs)
self.device.reset()
self.device.reg_cmd_strobe(StrobeAddress.SIDLE)
sleep_us(800)
self.device.reg_cmd_strobe(StrobeAddress.SFRX)                    # flush the
RX buffer
self.device.reg_cmd_strobe(StrobeAddress.SFTX)                    # flush the
TX buffer
self.device.reg_write(ConfigurationRegisterAddress.IOCFG2,   0x0D)
self.device.reg_write(ConfigurationRegisterAddress.IOCFG0,   0x0D)
self.device.reg_write(ConfigurationRegisterAddress.FIFOTHR,  0x47)
self.device.reg_write(ConfigurationRegisterAddress.PKTCTRL0, 0x32)
self.device.reg_write(ConfigurationRegisterAddress.FSCTRL1,  0x06)
self.device.reg_write(ConfigurationRegisterAddress.MCSM0,    0x18)
self.device.reg_write(ConfigurationRegisterAddress.FOCCFG,   0x16)
self.device.reg_write(ConfigurationRegisterAddress.WORCTRL,  0xFB)
self.device.reg_write(ConfigurationRegisterAddress.FSCAL3,   0xE9)
self.device.reg_write(ConfigurationRegisterAddress.FSCAL2,   0x2A)
self.device.reg_write(ConfigurationRegisterAddress.FSCAL1,   0x00)
self.device.reg_write(ConfigurationRegisterAddress.FSCAL0,   0x1F)
self.device.reg_write(ConfigurationRegisterAddress.TEST2,    0x81)
self.device.reg_write(ConfigurationRegisterAddress.TEST1,    0x35)
self.device.reg_write(ConfigurationRegisterAddress.TEST0,    0x09)
self.device.reg_write(ConfigurationRegisterAddress.CHANNR,   0x00)
if baud == 1200:
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG4,  0xF5)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG3,  0x75)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG2,  0x30)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG1,  0x72)
self.device.reg_write(ConfigurationRegisterAddress.DEVIATN,  0x14)
self.device.reg_write(ConfigurationRegisterAddress.FREND0,   0x11)
elif baud == 38400:
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG4,  0xCA)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG3,  0x83)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG2,  0x10)
self.device.reg_write(ConfigurationRegisterAddress.DEVIATN,  0x35)
self.device.reg_write(ConfigurationRegisterAddress.FREND0,   0x17)
self.device.reg_write(ConfigurationRegisterAddress.AGCTRL2,  0x43)
elif baud == 76800:
self.device.reg_write(ConfigurationRegisterAddress.FSCTRL1,  0x08)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG4,  0x7B)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG3,  0x83)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG2,  0x10)
self.device.reg_write(ConfigurationRegisterAddress.DEVIATN,  0x42)
self.device.reg_write(ConfigurationRegisterAddress.FOCCFG,   0x1D)
self.device.reg_write(ConfigurationRegisterAddress.BSCFG,    0x1C)
self.device.reg_write(ConfigurationRegisterAddress.AGCTRL2,  0xC7)
self.device.reg_write(ConfigurationRegisterAddress.AGCTRL1,  0x00)
self.device.reg_write(ConfigurationRegisterAddress.AGCTRL0,  0xB2)
self.device.reg_write(ConfigurationRegisterAddress.FREND0,   0x17)
self.device.reg_write(ConfigurationRegisterAddress.FREND1,   0xB6)
self.device.reg_write(ConfigurationRegisterAddress.FSCAL3,   0xEA)
elif baud == 100000:
self.device.reg_write(ConfigurationRegisterAddress.FSCTRL1,  0x08)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG4,  0x5B)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG3,  0xF8)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG2,  0x10)
self.device.reg_write(ConfigurationRegisterAddress.DEVIATN,  0x47)
self.device.reg_write(ConfigurationRegisterAddress.FOCCFG,   0x1D)
self.device.reg_write(ConfigurationRegisterAddress.BSCFG,    0x1C)
self.device.reg_write(ConfigurationRegisterAddress.AGCTRL2,  0xC7)
self.device.reg_write(ConfigurationRegisterAddress.AGCTRL1,  0x00)
self.device.reg_write(ConfigurationRegisterAddress.AGCTRL0,  0xB2)
self.device.reg_write(ConfigurationRegisterAddress.FREND0,   0x17)
self.device.reg_write(ConfigurationRegisterAddress.FREND1,   0xB6)
self.device.reg_write(ConfigurationRegisterAddress.FSCAL3,   0xEA)
elif baud == 250000:
self.device.reg_write(ConfigurationRegisterAddress.FSCTRL1,  0x0C)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG4,  0x2D)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG3,  0x3B)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG2,  0x10)
self.device.reg_write(ConfigurationRegisterAddress.DEVIATN,  0x62)
self.device.reg_write(ConfigurationRegisterAddress.FOCCFG,   0x1D)
self.device.reg_write(ConfigurationRegisterAddress.BSCFG,    0x1C)
self.device.reg_write(ConfigurationRegisterAddress.AGCTRL2,  0xC7)
self.device.reg_write(ConfigurationRegisterAddress.AGCTRL1,  0x00)
self.device.reg_write(ConfigurationRegisterAddress.AGCTRL0,  0xB0)
self.device.reg_write(ConfigurationRegisterAddress.FREND0,   0x17)
self.device.reg_write(ConfigurationRegisterAddress.FREND1,   0xB6)
self.device.reg_write(ConfigurationRegisterAddress.FSCAL3,   0xEA)
elif baud == 500000:  # MSK
self.device.reg_write(ConfigurationRegisterAddress.FSCTRL1,  0x0E)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG4,  0x0E)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG3,  0x3B)
self.device.reg_write(ConfigurationRegisterAddress.MDMCFG2,  0x70)
self.device.reg_write(ConfigurationRegisterAddress.DEVIATN,  0x00)
self.device.reg_write(ConfigurationRegisterAddress.FOCCFG,   0x1D)
self.device.reg_write(ConfigurationRegisterAddress.BSCFG,    0x1C)
self.device.reg_write(ConfigurationRegisterAddress.AGCTRL2,  0xC7)
self.device.reg_write(ConfigurationRegisterAddress.AGCTRL1,  0x00)
self.device.reg_write(ConfigurationRegisterAddress.AGCTRL0,  0xB0)
self.device.reg_write(ConfigurationRegisterAddress.FREND0,   0x17)
self.device.reg_write(ConfigurationRegisterAddress.FREND1,   0xB6)
self.device.reg_write(ConfigurationRegisterAddress.FSCAL3,   0xEA)
self.device.reg_write_bytes(PatableAddress.PATABLE, bytearray(patable_power_433))
self.set_frequency(frequency)
self.device.reg_cmd_strobe(StrobeAddress.SCAL)
sleep_us(800)
if catch0:
self.gdo0.irq(catch0, trigger=(Pin.IRQ_FALLING | Pin.IRQ_RISING))
if catch2:
self.gdo2.irq(catch2, trigger=Pin.IRQ_FALLING)
patable_power_433 = [0x00,0x12,0x0E,0x34,0x60,0xC5,0xC1,0xC0]
```

Many of the configuration registers changed, and each baud rate causes even more to change. But beyond that, everything else is the same.

The radio is now happily sending and receiving bytes at 250,000 bits per second (25,000 bytes per second).

The whoami.py module:

```
class WhoAmI:
def __init__(self):
self.me = {}
try:
with open("whoami.cfg","rb") as f:
line = f.read(1024)
from json import loads
self.me = loads(line)
except OSError as e:
print("Error reading whoami.cfg:", e )
```

def name(self):
if "name" in self.me:
return self.me["name"]
return "Unknown"   def ssid(self):
if "ssid" in self.me:
return self.me["ssid"]
return None   def ip(self):
if "ip" in self.me:
return self.me["ip"]
return None   def mask(self):
if "mask" in self.me:
return self.me["mask"]
return None   def gateway(self):
if "gateway" in self.me:
return self.me["gateway"]
return None   def dns(self):
if "dns" in self.me:
return self.me["dns"]
return None   def neo_pin(self):
if "neo_pin" in self.me:
return self.me["neo_pin"]
return None   def neo_how_many(self):
if "neo_how_many" in self.me:
return self.me["neo_how_many"]
return None   def set_ip(self, sta):
if "ip" in self.me:
sta.ifconfig((self.me["ip"], self.me["mask"], self.me["gateway"], self.me["dns"]))
The whoami.cfg file for the transmitter:
{"name":"Alice","ssid":"BirdfarmOffice2"}
And for the receiver:
{"name":"Bob","ssid":"BirdfarmOffice2"}
As usual, change BirdfarmOffice2 to your own SSID. In this project, we aren't using Wi-Fi, so it doesn't really matter.
Python Radio 40: Unleashing the CC1101
Advanced Packet Radio
Follow
20 min read
·
May 28, 2025
Listen
Share
More
Press enter or click to view image in full size
All photos by the Author
We used the CC1101 transceiver earlier as a CW transmitter (Morse Code).
But the radio can do so much more.
It is a full transceiver:
It can transmit and receive at data rates from 600 baud up to 600,000 baud.
It has programmable filters to match the signal-to-noise ratio to the baud rate.
It has automated packet handling, with preambles, sync words, checksums and forward error

correction, modulation and demodulation, bit synchronization, byte synchronization, whitening and de-whitening, interleaving, and decoding. All without burdening the computer.
It supports amplitude modulation, frequency modulation, and phase modulation, and on-off keying. It can also handle Manchester coding and decoding.
In frequency shift keying, it can support 2-FSK, 4-FSK, and it has a Gaussian filter (GFSK) to allow higher data rates in narrower bandwidths.
It can perform six different tests to determine link quality.
So let’s explore some of the more powerful aspects of the radio.
Connecting to the ESP32
Since we want both a transmitter and a receiver, let’s use two different ESP32 boards — the ESP32-S3
and the ESP32-C3 Super Mini.
The radio has 8 pins. We will use seven of them.
Pin Connections
Our ESP32-C3 looks like this on the breadboard:
Press enter or click to view image in full size
The ESP32-S3 is a larger board, and needed two breadboards:
Press enter or click to view image in full size
Frequencies
The radio can transmit and receive on several license-free bands:
300 to 348 Megahertz
387 to 464 Megahertz
779 to 928 Megahertz
I chose the middle band, specifically 433.92 MHz, for a few reasons. First, it lies in the
70-centimeter amateur radio band, which extends from 420 MHz to 450 MHz in the U.S. This means I c
choose to amplify the output if I choose (and if I obey the amateur radio rules). It also means my
70-centimeter antennas will work with it. But also, longer wavelengths come with longer ranges, and the 800 and 900 MHz frequencies don’t have the range of 433 MHz.
I get about a kilometer with the tiny antennas that come with the radio, even though the radio only emits 12 dBm (15.85 milliwatts).
The Software
To show the features and power of the radio, we will send and receive packets with preamble words, sync words, forward error correction, and checksums, all handled in the radio without having to code them up in the ESP32.
Here is the code:
```
from machine import Pin, SPI
from time import sleep, sleep_ms, sleep_us, ticks_diff, ticks_ms
from sys import print_exception
from whoami import WhoAmI
w = WhoAmI()
# Configuration Registers
IOCFG2 = 0x00    # GDO2 Output Pin Configuration
IOCFG1 = 0x01    # GDO1 Output Pin Configuration
IOCFG0 = 0x02    # GDO0 Output Pin Configuration
FIFOTHR = 0x03   # RX FIFO and TX FIFO Thresholds
SYNC1 = 0x04     # Sync Word, High Byte
SYNC0 = 0x05     # Sync Word, Low Byte
PKTLEN = 0x06    # Packet Length
PKTCTRL1 = 0x07  # Packet Automation Control
PKTCTRL0 = 0x08  # Packet Automation Control
ADDR = 0x09      # Device Address
```

```
CHANNR = 0x0A    # Channel Number
FSCTRL1 = 0x0B   # Frequency Synthesizer Control
FSCTRL0 = 0x0C   # Frequency Synthesizer Control
FREQ2 = 0x0D     # Frequency Control Word, High Byte
FREQ1 = 0x0E     # Frequency Control Word, Middle Byte
FREQ0 = 0x0F     # Frequency Control Word, Low Byte
MDMCFG4 = 0x10   # Modem Configuration
MDMCFG3 = 0x11   # Modem Configuration
MDMCFG2 = 0x12   # Modem Configuration
MDMCFG1 = 0x13   # Modem Configuration
MDMCFG0 = 0x14   # Modem Configuration
DEVIATN = 0x15   # Modem Deviation Setting
MCSM2 = 0x16     # Main Radio Control State Machine Configuration
MCSM1 = 0x17     # Main Radio Control State Machine Configuration
MCSM0 = 0x18     # Main Radio Control State Machine Configuration
FOCCFG = 0x19    # Frequency Offset Compensation Configuration
BSCFG = 0x1A     # Bit Synchronization Configuration
AGCCTRL2 = 0x1B  # AGC Control
AGCCTRL1 = 0x1C  # AGC Control
AGCCTRL0 = 0x1D  # AGC Control
WOREVT1 = 0x1E   # High Byte Event0 Timeout
WOREVT0 = 0x1F   # Low Byte Event0 Timeout
WORCTRL = 0x20   # Wake On Radio Control
FREND1 = 0x21    # Front End RX Configuration
FREND0 = 0x22    # Front End TX Configuration
FSCAL3 = 0x23    # Frequency Synthesizer Calibration
FSCAL2 = 0x24    # Frequency Synthesizer Calibration
FSCAL1 = 0x25    # Frequency Synthesizer Calibration
FSCAL0 = 0x26    # Frequency Synthesizer Calibration
RCCTRL1 = 0x27   # RC Oscillator Configuration
RCCTRL0 = 0x28   # RC Oscillator Configuration
FSTEST = 0x29    # Frequency Synthesizer Calibration Control
PTEST = 0x2A     # Production Test
AGCTEST = 0x2B   # AGC Test
TEST2 = 0x2C     # Various Test Settings
TEST1 = 0x2D     # Various Test Settings
TEST0 = 0x2E     # Various Test Settings
reg_names = [
"IOCFG2",
"IOCFG1",
"IOCFG0",
"FIFOTHR",
"SYNC1",
"SYNC0",
"PKTLEN",
"PKTCTRL1",
"PKTCTRL0",
"ADDR",
"CHANNR",
"FSCTRL1",
"FSCTRL0",
"FREQ2",
"FREQ1",
"FREQ0",
"MDMCFG4",
"MDMCFG3",
"MDMCFG2",
"MDMCFG1",
"MDMCFG0",
"DEVIATN",
"MCSM2",
"MCSM1",
"MCSM0",
"FOCCFG",
"BSCFG",
"AGCCTRL2",
"AGCCTRL1",
"AGCCTRL0",
"WOREVT1",
"WOREVT0",
"WORCTRL",
"FREND1",
"FREND0",
"FSCAL3",
"FSCAL2",
"FSCAL1",
"FSCAL0",
"RCCTRL1",
"RCCTRL0",
"FSTEST",
"PTEST",
"AGCTEST",
"TEST2",
"TEST1",
"TEST0",
]
# Status Registers (accessed with Read Single/Burst + 0x80/0xC0, or specific status byte commands)
PARTNUM = 0xF0   # Chip ID (Should be 0x00 for CC1101)
VERSION = 0xF1   # Chip ID (Should be 0x04 for CC1101)
FREQEST = 0xF2   # Frequency Offset Estimate from Demodulator
LQI = 0xF3       # Demodulator Estimate for Link Quality
RSSI = 0xF4      # Received Signal Strength Indication
MARCSTATE = 0xF5 # Main Radio Control State Machine State
WORTIME1 = 0xF6  # High Byte of WOR Time
WORTIME0 = 0xF7  # Low Byte of WOR Time
PKTSTATUS = 0xF8 # Current GDOx Status and Packet Status
VCO_VC_DAC = 0xF9# Current Setting from PLL Calibration Module
TXBYTES = 0xFA   # Underflow and Number of Bytes
RXBYTES = 0xFB   # Overflow and Number of Bytes
RCCTRL1_STATUS = 0xFC # Last RC Oscillator Calibration Result
```

```
RCCTRL0_STATUS = 0xFD # Last RC Oscillator Calibration Result
# Strobe Commands
SRES = 0x30      # Reset chip.
SFSTXON = 0x31   # Enable and calibrate frequency synthesizer (if MCSM0.FS_AUTOCAL=1).
SXOFF = 0x32     # Turn off crystal oscillator.
SCAL = 0x33      # Calibrate frequency synthesizer and turn it off.
SRX = 0x34       # Enable RX.
STX = 0x35       # Enable TX.
SIDLE = 0x36     # Exit RX/TX, turn off frequency synthesizer.
SWOR = 0x38      # Start automatic RX polling sequence (Wake-on-Radio)
SPWD = 0x39      # Enter power down mode when CSn goes high.
SFRX = 0x3A      # Flush the RX FIFO buffer.
SFTX = 0x3B      # Flush the TX FIFO buffer.
SWORRST = 0x3C   # Reset real time clock.
SNOP = 0x3D      # No operation.
# PATABLE and FIFO Addresses
PATABLE_ADDR = 0x3E # Address for PATABLE
TXFIFO_ADDR = 0x3F    # TX FIFO address
RXFIFO_ADDR = 0x3F    # RX FIFO address
# SPI Header Bits
WRITE_SINGLE_BYTE = 0x00
WRITE_BURST = 0x40
READ_SINGLE_BYTE = 0x80
READ_BURST = 0xC0
# Note: For status registers (0x30-0x3D, which become 0xF0-0xFD with burst bit),
# the datasheet often shows them accessed with the burst bit set.
# e.g., MARCSTATE (0x35) is read with 0x35 | 0xC0 = 0xF5.
# --- Constants ---
FXOSC = 26000000  # Crystal oscillator frequency in Hz (26MHz)
MAX_PACKET_LEN = 60 # Max payload length. FIFO is 64 bytes. (1 length byte + payload + 2 status
bytes + FEC padding)
class CC1101:
def __init__(self, spi, cs_pin_id, gdo0_pin_id=None):
self.rssi_dbm = -99
self.spi = spi
self.cs = Pin(cs_pin_id, Pin.OUT)
self.cs.on() # Active low, so set high initially
self.use_variable_length_packets = False
self.gdo0 = None
if gdo0_pin_id is not None:
# Configure GDO0 as input with pull-down
# (GDO0 is typically active high from CC1101)
self.gdo0 = Pin(gdo0_pin_id, Pin.IN, Pin.PULL_DOWN)
print(f"GDO0 configured on GPIO {gdo0_pin_id}")
self.reset()
print("Attempting to go IDLE immediately after reset...")
self.idle()
print("--- Basic Register Read Test ---")
try:
marc_state = self._read_status_reg(MARCSTATE) & 0x1F
```

```
partnum = self._read_status_reg(PARTNUM) # PARTNUM is 0x30, so 0x30|0xC0 = 0xF0
version = self._read_status_reg(VERSION) # VERSION is 0x31, so 0x31|0xC0 = 0xF1
# print(f"MARCSTATE after initial idle attempt: 0x{marc_state:02X}")
# print(f"PARTNUM: 0x{partnum:02X} (Expected: 0x00 for CC1101)")
# print(f"VERSION: 0x{version:02X} (Expected: >= 0x04, e.g., 0x14 for rev E)")
# Test basic register write/read
self._write_reg(CHANNR, 0xAA)
channr_read = self._read_reg(CHANNR)
# print(f"Wrote 0xAA to CHANNR, Read: 0x{channr_read:02X}")
if channr_read != 0xAA:
print("ERROR: Basic register write/read test FAILED!")
else:
print("Basic register write/read test PASSED.")
except Exception as e:
print(f"Error during basic register read test: {e}")
print_exception(e)
# self.dump_regs()
# self.snop_test()
self.current_data_rate_kbps = w.baud()
self.has_FEC = w.fec()
self.configure_gfsk(self.current_data_rate_kbps)
def snop_test(self):
SNOP = 0x3D
self.cs.on()
print("Performing SNOP test...")
sleep_ms(10)
self.cs.off()
sleep_us(10)
tx_buf = bytearray([SNOP])
rx_buf = bytearray(1)
self.spi.write_readinto(tx_buf, rx_buf)
status_byte = rx_buf[0]
sleep_us(10)
self.cs.on()
print(f"Status byte after SNOP: 0x{status_byte:02X}")
chip_rdyn = status_byte & 0x80
state = (status_byte >> 4) & 7
fifo_bytes = status_byte & 15
states = ["IDLE", "RX", "TX", "FSTXON", "CALIBRATE", "SETTLING", "RXFIFO_OVERFLOW",
"TXFIFO_OVERFLOW"]
print(f"CHIP_RDYn: {chip_rdyn}")
print(f"STATE    : {state} {states[state]}")
print(f"FIFO     : {fifo_bytes}")
def dump_regs(self):
for i in range(0x2F):
reg = self._read_reg(i)
print(f"{i:02X} {reg_names[i]:8s}: {reg:02X}")
def _strobe(self, cmd):
self.cs.off()
self.spi.write(bytearray([cmd]))
self.cs.on()
sleep_us(50)
def _write_reg(self, addr, value):
self.cs.off()
self.spi.write(bytearray([addr | WRITE_SINGLE_BYTE, value]))
self.cs.on()
sleep_us(50)
def _read_reg(self, addr):
self.cs.off()
wbuf = bytearray([addr | READ_SINGLE_BYTE, 0x00])
rbuf = bytearray([0, 0])
self.spi.write_readinto(wbuf, rbuf)
val = rbuf[1]
self.cs.on()
sleep_us(50)
return val
def _read_status_reg(self, status_reg_addr_with_burst_bit):
self.cs.off()
wbuf = bytearray([status_reg_addr_with_burst_bit, 0x00])
rbuf = bytearray([0, 0])
self.spi.write_readinto(wbuf, rbuf)
val = rbuf[1]
self.cs.on()
sleep_us(50)
return val
def _write_burst_reg(self, addr, data):
self.cs.off()
self.spi.write(bytearray([addr | WRITE_BURST]))
self.spi.write(bytearray(data))
self.cs.on()
sleep_us(50)
def _read_burst_reg(self, addr, length):
self.cs.off()
wbuf = bytearray([addr | READ_BURST] + [0x00]*length)
rbuf = bytearray([addr | READ_BURST] + [0x00]*length)
self.spi.write_readinto(wbuf, rbuf)
data = rbuf[1:]
self.cs.on()
sleep_us(50)
return data
def reset(self):
self.cs.off()
sleep_us(10)
self.cs.on()
sleep_us(45)
self._strobe(SRES)
sleep_ms(2)
def configure_gfsk(self, data_rate_kbps):
print(f"Configuring CC1101 for GFSK at approximately {data_rate_kbps} kBaud...")
current_mdmcfg4  = 0x2D
```

```
current_mdmcfg3  = 0x3B
current_deviatn  = 0x62
current_fsctrl1  = 0x0C
current_agcctrl2 = 0xC7
current_agcctrl1 = 0x00
current_agcctrl0 = 0xB0
if data_rate_kbps == 250:
print("Using 250 kBaud settings...")
# MDMCFG4, MDMCFG3, DEVIATN already set to 250k defaults above
# FSCTRL1, AGCCTRLx also already set to 250k defaults above
elif data_rate_kbps == 38.4:
print("Using ~38.4 kBaud settings...")
current_mdmcfg4  = 0xA8 # DRATE_E=8, CHANBW_E=2, CHANBW_M=2 => RX BW ~135 kHz
current_mdmcfg3  = 0x93 # DRATE_M=147
current_deviatn  = 0x35 # Deviation ~19kHz
current_fsctrl1  = 0x06 # IF ~152 kHz for 38.4k
current_agcctrl2 = 0x03
current_agcctrl1 = 0x40
current_agcctrl0 = 0x92
else:
print(f"ERROR: No pre-defined settings for {data_rate_kbps} kBaud. Using 250k defaults.")
self._write_reg(MDMCFG4,  current_mdmcfg4)
self._write_reg(MDMCFG3,  current_mdmcfg3)
self._write_reg(DEVIATN,  current_deviatn)
self._write_reg(FSCTRL1,  current_fsctrl1)
self._write_reg(FSCTRL0,  0x00)
self._write_reg(MDMCFG2,  0x13)  # MOD_FORMAT=GFSK, SYNC_MODE=30/32
if self.has_FEC:
print("We are using Forward Error Correction")
self._write_reg(MDMCFG1,  0xA2)  # FEC_EN, NUM_PREAMBLE=4 bytes
else:
self._write_reg(MDMCFG1,  0x22)  # FEC_DIS, NUM_PREAMBLE=4 bytes
self._write_reg(MDMCFG0,  0xF8)  # CHANSPC_M
self._write_reg(MCSM2,    0x07)
self._write_reg(MCSM1,    0x30)  # RXOFF_MODE=IDLE, TXOFF_MODE=IDLE
self._write_reg(MCSM0,    0x18)  # FS_AUTOCAL from IDLE
self._write_reg(FOCCFG,   0x1D)
self._write_reg(BSCFG,    0x1C)
self._write_reg(AGCCTRL2, current_agcctrl2)
self._write_reg(AGCCTRL1, current_agcctrl1)
self._write_reg(AGCCTRL0, current_agcctrl0)
self._write_reg(FREND1,   0xB6)
self._write_reg(FREND0,   0x10)
self._write_reg(FSCAL3,   0xEA)
self._write_reg(FSCAL2,   0x2A)
self._write_reg(FSCAL1,   0x00)
self._write_reg(FSCAL0,   0x1F)
self._write_reg(PKTCTRL0, 0x04) # CRC_EN=1, Fixed Length, WHITE_DATA=0
self._write_reg(PKTCTRL1, 0x04) # APPEND_STATUS=1
self._write_reg(PKTLEN,   MAX_PACKET_LEN)
self._write_reg(IOCFG0,   0x06)
self.set_tx_power(0xC0)
self.set_address(0x00) # Not used if PKTCTRL1.ADR_CHK = 0
self.set_channel(0)
self.set_sync_word(0xD3, 0x91)
self.idle()
print(f"CC1101 GFSK Configuration for ~{data_rate_kbps} kBaud Applied.")
return True
def set_frequency_mhz(self, freq_mhz):
freq_hz = int(freq_mhz * 1_000_000)
# Formula: FREQ_REG = (freq_hz / FXOSC) * 2^16
freq_reg_val = int((freq_hz * (1 << 16)) / FXOSC)
f2 = (freq_reg_val >> 16) & 0xFF
f1 = (freq_reg_val >> 8) & 0xFF
f0 = freq_reg_val & 0xFF
self._write_reg(FREQ2, f2)
self._write_reg(FREQ1, f1)
self._write_reg(FREQ0, f0)
print(f"Set Frequency: {freq_hz/1e6:.3f} MHz (Registers: F2:0x{f2:02X}, F1:0x{f1:02X},
F0:0x{f0:02X})")
# It's good practice to recalibrate after frequency change if going to TX/RX
# self._strobe(SCAL)
# while (self._read_status_reg(MARCSTATE) & 0x1F) != 0x01: sleep_us(100) # Wait for CAL to finish
(IDLE state)
def set_tx_power(self, power_val_pa_table_entry=0xC0):
# For GFSK/FSK/MSK, only the first byte of PATABLE is used.
# Common values: 0x00 (-30dBm), 0x12 (-20dBm), ..., 0xC0 (+10dBm)
self._write_reg(PATABLE_ADDR, power_val_pa_table_entry)
def set_address(self, addr_byte):
self._write_reg(ADDR, addr_byte)
def set_channel(self, channr_byte):
self._write_reg(CHANNR, channr_byte)
def set_sync_word(self, sync1_byte, sync0_byte):
self._write_reg(SYNC1, sync1_byte)
self._write_reg(SYNC0, sync0_byte)
def idle(self):
self._strobe(SIDLE)
sleep_ms(1)
# Wait until the chip is in IDLE state (MARCSTATE == 0x01)
for x in range(150):
current_marc_state = self._read_status_reg(MARCSTATE) & 0x1F
final_marc_state = current_marc_state
if current_marc_state == 0x01:
return
sleep_us(200)
print("Warning: CC1101 did not enter IDLE state after SIDLE strobe.")
print(f"Warning: CC1101 did not confirm IDLE state after SIDLE strobe. Last MARCSTATE read:
0x{final_marc_state:02X}")
def enter_rx(self):
self.idle()
self._strobe(SFRX)
self._strobe(SRX)
def pad(self, data, length):
l = len(data)
if l >= length:
return data[:length]
else:
padding_needed = length - l
data.extend(b'\x00' * padding_needed)
return data
def send_packet(self, data, tx_timeout_ms=1000):
data = self.pad(data, MAX_PACKET_LEN)
fixed_payload_length = len(data)
if not (0 < fixed_payload_length <= MAX_PACKET_LEN):
print(f"Error: Packet payload length {fixed_payload_length} invalid (must be 1 to
{MAX_PACKET_LEN}).")
return False
self.idle()
self._strobe(SFTX)
tx_bytes_status = self._read_status_reg(TXBYTES)
if self.use_variable_length_packets:
print(f"TX: Writing length byte: {fixed_payload_length}")
# First byte to TXFIFO is the length of the payload
self._write_reg(TXFIFO_ADDR, fixed_payload_length)
# Then write the payload data
self._write_burst_reg(TXFIFO_ADDR, data)
else:
self._write_burst_reg(TXFIFO_ADDR, data)
# Verify bytes in TX FIFO (optional debug)
tx_bytes_status = self._read_status_reg(TXBYTES)
# print(f"tx_bytes_status is {tx_bytes_status}")
# print(f"MARCSTATE before STX: 0x{self._read_status_reg(MARCSTATE) & 0x1F:02X}")
self._strobe(STX)  # Start transmission
sleep_us(100) # Give strobe time to process
# print(f"MARCSTATE after STX: 0x{self._read_status_reg(MARCSTATE) & 0x1F:02X}")
start_time = ticks_ms()
tx_completed_successfully = False
if self.gdo0:
# Wait for GDO0 to go high (sync word sent) then low (packet sent, chip returns to IDLE or other
state based on MCSM1)
# With IOCFG0 = 0x06, GDO0 goes high when sync is sent, and low when packet is fully sent and radi
leaves TX.
gdo0_high_seen = False
gdo0_val = self.gdo0.value()
while ticks_diff(ticks_ms(), start_time) < tx_timeout_ms:
gdo0_val = self.gdo0.value()
if not gdo0_high_seen and gdo0_val == 1:
gdo0_high_seen = True # Sync word sent, payload transmission in progress
if gdo0_high_seen and gdo0_val == 0:
# GDO0 went low after being high, check MARCSTATE to confirm IDLE
```

```
sleep_us(100)
if (self._read_status_reg(MARCSTATE) & 0x1F) == 0x01: # IDLE state
tx_completed_successfully = True
break
# If not IDLE, it might be an error state or unexpected transition
sleep_us(100) # Poll GDO0
self.idle() # Ensure radio is IDLE after TX attempt (this calls SIDLE)
if not tx_completed_successfully:
final_marc_state_on_timeout = self._read_status_reg(MARCSTATE) & 0x1F
tx_bytes_final = self._read_status_reg(TXBYTES) # Read the full TXBYTES register
print(f"TX Failure: MARCSTATE=0x{final_marc_state_on_timeout:02X},
TXBYTES_REG=0x{tx_bytes_final:02X} (UF: {tx_bytes_final>>7}, Num: {tx_bytes_final&0x7F})")
gdo0_val = self.gdo0.value()
self._strobe(SFTX)
self.dump_regs()
print()
return False
self.idle()
if not tx_completed_successfully:
self._strobe(SFTX)
return False
# print("TX: Packet sent successfully.")
return True
def receive_packet(self, rx_timeout_ms=1000):
self.enter_rx()
start_time = ticks_ms()
packet_detected_by_gdo0 = False
baud = int(w.baud() * 1000)
if self.gdo0:
# Wait for GDO0 to go high (sync word received)
while ticks_diff(ticks_ms(), start_time) < rx_timeout_ms:
if self.gdo0.value() == 1:
packet_detected_by_gdo0 = True
break # Sync word detected, proceed to wait for end of packet
sleep_us(100)
if packet_detected_by_gdo0:
# print("Sync word detected")
# Now wait for GDO0 to go low (end of packet transmission)
eop_start_time = ticks_ms()
# Max EOP wait: roughly time for max packet at current baud rate + buffer
max_eop_wait_ms = (5 * MAX_PACKET_LEN * 8 * 1000 // baud) + 50
while self.gdo0.value() == 1 and ticks_diff(ticks_ms(), eop_start_time) < max_eop_wait_ms:
sleep_us(100)
if self.gdo0.value() == 0:
# print("End of packet detected")
pass
else: # GDO0 stuck high or EOP timeout
current_marc_state_eop_timeout = self._read_status_reg(MARCSTATE) & 0x1F
rx_bytes_at_eop_timeout = self._read_status_reg(RXBYTES) & 0x7F
print(f"RX: GDO0 stuck high or EOP timeout. MARCSTATE: 0x{current_marc_state_eop_timeout:02X}
RXBYTES: {rx_bytes_at_eop_timeout}")
if rx_bytes_at_eop_timeout > 0: # Check if there are actually bytes to peek
try:
peek_len = rx_bytes_at_eop_timeout # Read all available bytes up to what RXBYTES reports
if peek_len > 0 : # Ensure we actually try to read if bytes are reported
fifo_peek = self._read_burst_reg(RXFIFO_ADDR, peek_len)
# fifo_peek = fifo_peek.decode("utf-8", "ignore")
print(f"FIFO Peek ({peek_len} bytes): {fifo_peek}")
except Exception as e:
print_exception(e)
print(f"Error peeking FIFO: {e}")
self.idle()
self._strobe(SFRX) # Flush RX FIFO
return None
else: # Timeout waiting for GDO0 to go high (no packet detected)
self.idle()
return None
if ((self._read_status_reg(RXBYTES) & 0x7F) <= 0):
self.idle()
return None
self.finished = False
if self.gdo0.value() == 0:
self.finished = True
# At this point, GDO0 has indicated a complete packet
num_bytes_in_fifo = self._read_status_reg(RXBYTES) & 0x7F # Mask out MSB (overflow bit)
expected_fixed_payload_len = self._read_reg(PKTLEN)
if num_bytes_in_fifo > 0:
# print(f"RX: {num_bytes_in_fifo} bytes indicated in FIFO.")
expected_fifo_content_len = expected_fixed_payload_len + 2 # 2 for status bytes
if num_bytes_in_fifo < expected_fifo_content_len:
print(f"RX Error: FIFO has {num_bytes_in_fifo} bytes, expected fixed payload
{expected_fixed_payload_len} + 2 status bytes.")
self.idle()
self._strobe(SFRX)
return None
payload = self._read_burst_reg(RXFIFO_ADDR, expected_fixed_payload_len)
status_bytes = self._read_burst_reg(RXFIFO_ADDR, 2)
rssi_val = status_bytes[0]
lqi_and_crc = status_bytes[1]
lqi = lqi_and_crc & 0x7F
crc_ok = (lqi_and_crc >> 7) & 0x01
# Convert RSSI register value to dBm
# (Refer to CC1101 datasheet section 17.1.5 for exact formula and offset)
# A common approximation for the offset is -74 dBm.
rssi_offset = 74
if rssi_val >= 128:
self.rssi_dbm = (rssi_val - 256) / 2.0 - rssi_offset
else:
self.rssi_dbm = rssi_val / 2.0 - rssi_offset
if crc_ok:
```

```
# print(f"RX: Packet - RSSI from reg: {self._read_status_reg(RSSI):02X}, LQI from reg:
{self._read_status_reg(LQI):02X}")
self.idle()
return bytes(payload)
else:
print(f"RX Error: CRC FAILED! RSSI: {self.rssi_dbm:.1f} dBm, LQI: {lqi}")
self.idle()
self._strobe(SFRX)
return None
else:
# print("RX: GDO0 signaled (or poll indicated) but no bytes in FIFO after processing.")
self.idle()
return None
def main():
SPI_BUS_ID  = 1
SCK_PIN_ID  = w.sck()
MOSI_PIN_ID = w.mosi()
MISO_PIN_ID = w.miso()
CS_PIN_ID   = w.csn()
GDO0_PIN_ID = w.gdo0()
print(f"I am {w.name()}")
try:
spi = SPI(SPI_BUS_ID,
baudrate=5_000_000,  # 5 MHz (CC1101 supports up to 10MHz for register access)
polarity=0,          # SPI mode 0: CPOL=0, CPHA=0
phase=0,
sck=Pin(SCK_PIN_ID),
mosi=Pin(MOSI_PIN_ID),
miso=Pin(MISO_PIN_ID), firstbit=SPI.MSB)
except Exception as e:
print_exception(e)
print(f"FATAL: Error initializing SPI on ESP32: {e}")
return
try:
cc_device = CC1101(spi, CS_PIN_ID, GDO0_PIN_ID)
except Exception as e:
print_exception(e)
print(f"FATAL: Error initializing CC1101: {e}")
return
# Set a common operating frequency (e.g., 433.92 MHz)
# This must be the same on both transmitter and receiver.
cc_device.set_frequency_mhz(433.92)
if w.name() == "Alice":
print(f"--- Transmitter Mode (GPIO CS: {CS_PIN_ID}, GDO0: {GDO0_PIN_ID if GDO0_PIN_ID else 'No
---")
packet_counter = 0
msg_num = 0
while True:
try:
if msg_num % 2 == 0:
```

```
message_to_send = f"Hello from ESP32! Message #{msg_num}"
else:
message_to_send = f"Hello from ESP32! Message #{msg_num} This additional part of the message is
make sure we handle longer messages properly without causing problems in either the transmitter or
the receiver."
msg_num += 1
data_to_send = bytearray(message_to_send.encode('utf-8'))
while len(data_to_send):
if len(data_to_send) > MAX_PACKET_LEN:
data = data_to_send[:MAX_PACKET_LEN]
data_to_send = data_to_send[MAX_PACKET_LEN:]
else:
data = data_to_send
data_to_send = b""
print(f"TX: Sending packet #{packet_counter} ({len(data)} bytes): '{data.decode('utf-8',
'ignore')}'")
if cc_device.send_packet(data, tx_timeout_ms=1000):
# print(f"TX: Packet #{packet_counter} sent successfully.")
pass
else:
print(f"TX: Failed to send packet #{packet_counter} (timeout or TX error).")
packet_counter += 1
sleep_ms(10)
except KeyboardInterrupt:
print("\nTX Mode Stopped by user.")
break
except Exception as e:
print_exception(e)
print(f"TX Error: {e}")
sleep(1) # Pause briefly on error
elif w.name() == "Bob":
print(f"--- Receiver Mode (GPIO CS: {CS_PIN_ID}, GDO0: {GDO0_PIN_ID if GDO0_PIN_ID else 'None
---")
print("Waiting for packets...")
received_count = 0
while True:
try:
received_payload = cc_device.receive_packet(rx_timeout_ms=5000) # 5 second RX timeout
if received_payload:
received_count += 1
try:
message = received_payload.decode('utf-8')
print(f"RX #{received_count} {cc_device.rssi_dbm} dBm: Message ({len(received_payload)} bytes):
'{message}'")
except UnicodeError:
print(f"RX #{received_count}: Non-UTF8 Data ({len(received_payload)} bytes): {received_payload}")
except KeyboardInterrupt:
print("\nRX Mode Stopped by user.")
break
except Exception as e:
```

```
print_exception(e)
print(f"RX Error: {e}")
sleep_ms(200)
cc_device.idle()
print("Program finished.")
if __name__ == "__main__":
# Add a small delay to allow USB serial connection to establish if running on boot
# This is helpful when the ESP32 reboots and immediately starts printing.
sleep(10)
main()
```

The first parts are just setting up the addresses for the register on the radio.

Then we define the CC1101 class.

Configuring the Radio

The __init__() method sets up the pins and tests the connection to the radio, reporting any problems.

As with previous programs, we use a WhoAmI class to distinguish the two boards and their functions (transmit or receive).

The SNOP test is an optional debug aid to test the radio. The dump_regs() method is another debug method.

Next come the routines that read and write the registers in the radio to control its operation.

The configure_gfsk() method configures the registers in the radio for packet mode, Gaussian Frequency Shift Keying (GFSK), with Forward Error Correction (FEC). It also sets the baud rate, power, channel, and sync words.

When changing the baud rate, several registers have to cooperate. The bandwidth filters have to match the data rate, and the deviation has to change (the frequency difference between a one and a zero).

The set_frequency_mhz() method calculates the values to put into the three registers that govern the frequency. We need 24 bits of data to cover the huge range of frequencies this radio can cover.

The next three methods set the address (we aren't using address filtering), channel (we only use channel zero), and sync words. Channel spacing is every 100 kilohertz, but you can also just set the frequency to anything you like in the three bands.

Now we come to the packet handling code.

Transmitting a Packet

The send_packet() method first ensures that the packets are all the same length, using the pad() method. The radio can do fixed-length packets, variable-length packets, or infinite-length packets (where the host processor handles the length). But we have chosen to use FEC, and that is only supported in fixed-length packets.

The radio has to be in IDLE state before changing into TX or RX states. We put it in IDLE, then flush any data in the transmit FIFO buffer. Then we fill the FIFO buffer.

When the radio has sent the preamble bits (alternating ones and zeros for the receiver to lock on) and the sync words (which the receiver uses to find out where words start in the binary stream), the radio signals the host by setting the GDO0 pin high.

We loop waiting to see that signal, and to see it drop back to low, indicating that the full packet has been sent.

We change back to IDLE state, and flush the transmit FIFO buffer if there were any errors.

Receiving a Packet

The receive_packet() method enters the RX mode, and waits for the GDO0 pin to go high, indicating that a packet has been seen (the preamble and sync words have been detected).

When GDO0 drops to low, the end of a packet has been detected, the FEC codes have corrected any correctable errors, and the checksum indicates that there are no errors detected.

Next, the bytes are read from the receive FIFO buffer, and the RSSI (Receive Signal Strength Indicator) and LQI (Link Quality Indicator) status bytes are read.

The Main Routine

The main() routine sets up the SPI link to the radio, and gets an instance of the CC1101 class.
It then sets the frequency and uses the WhoAmI class to determine whether to transmit or receive.
Alice transmits, constructing a message to send and calling send_packet(). She sends messages of tw
different lengths to test the code that breaks long messages up into 60-byte packets.
Bob receives, calling receive_packet() and printing out the message.

The Output

Here is what the output looks like:

```
RX #11736 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109177 This additional part
th'
RX #11737 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages
properly'
RX #11738 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the
r'
RX #11739 -12.5 dBm: Message (60 bytes): 'eceiver.'
RX #11740 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109178'
RX #11741 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109179 This additional part
th'
RX #11742 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages
properly'
RX #11743 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the
r'
RX #11744 -13.0 dBm: Message (60 bytes): 'eceiver.'
RX #11745 -13.0 dBm: Message (60 bytes): 'Hello from ESP32! Message #109180'
RX #11746 -13.0 dBm: Message (60 bytes): 'Hello from ESP32! Message #109181 This additional part
th'
RX #11747 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages
properly'
RX #11748 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the
r'
RX #11749 -12.5 dBm: Message (60 bytes): 'eceiver.'
RX #11750 -13.0 dBm: Message (60 bytes): 'Hello from ESP32! Message #109182'
RX #11751 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109183 This additional part
th'
RX #11752 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages
properly'
RX #11753 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the
r'
RX #11754 -12.5 dBm: Message (60 bytes): 'eceiver.'
RX #11755 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109184'
RX #11756 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109185 This additional part
th'
RX #11757 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages
properly'
RX #11758 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the
r'
RX #11759 -12.5 dBm: Message (60 bytes): 'eceiver.'
RX #11760 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109186'
RX #11761 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109187 This additional part
th'
RX #11762 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages
properly'
RX #11763 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the
r'
RX #11764 -12.5 dBm: Message (60 bytes): 'eceiver.'
RX #11765 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109188'
RX #11766 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109189 This additional part
th'
RX #11767 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages
properly'
RX #11768 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the
r'
RX #11769 -12.5 dBm: Message (60 bytes): 'eceiver.'
RX #11770 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109190'
RX #11771 -12.5 dBm: Message (60 bytes): 'Hello from ESP32! Message #109191 This additional part
th'
RX #11772 -12.5 dBm: Message (60 bytes): 'e message is to make sure we handle longer messages
properly'
RX #11773 -12.5 dBm: Message (60 bytes): ' without causing problems in either the transmitter or the
r'
RX #11774 -12.5 dBm: Message (60 bytes): 'eceiver.'
```

Here is what the 38,400 baud signal looks like in our SDR:

Press enter or click to view image in full size

Here is what 250,000 baud looks like:

Press enter or click to view image in full size

We get much better range at the lower baud rate, due to the tighter filtering possible, which increases the signal-to-noise ratio a lot.

Gaussian Frequency Shift Keying uses two frequencies and softens the transition between them using Gaussian filter. This uses less bandwidth than without the filtering and allows higher data rates.

The radio can also handle 4-FSK, where four tones are used, doubling the data rate in the same bandwidth. Here is what that looks like:

All of this capability comes for $1.27, including the antenna. The ESP32-C3 Super Mini adds another $5.79. You can explore sophisticated radio networking for about $7 per node.

Here is the whoami.py module:

```
class WhoAmI:
def __init__(self):
self.me = {}
try:
with open("whoami.cfg","rb") as f:
line = f.read(1024)
from json import loads
self.me = loads(line)
except OSError as e:
print("Error reading whoami.cfg:", e )
def name(self):
if "name" in self.me:
return self.me["name"]
return "Unknown"
```

```
def baud(self):
if "baud" in self.me:
return self.me["baud"]
return 38.4
def fec(self):
if "fec" in self.me:
return self.me["fec"]
return True
def gdo0(self):
if "gdo0" in self.me:
return self.me["gdo0"]
return None
def gdo2(self):
if "gdo2" in self.me:
return self.me["gdo2"]
return None
def csn(self):
if "csn" in self.me:
return self.me["csn"]
return None
def sck(self):
if "sck" in self.me:
return self.me["sck"]
return None
def mosi(self):
if "mosi" in self.me:
return self.me["mosi"]
return None
def miso(self):
if "miso" in self.me:
return self.me["miso"]
return None
def ip(self):
if "ip" in self.me:
return self.me["ip"]
return None
def mask(self):
if "mask" in self.me:
return self.me["mask"]
return None
def gateway(self):
if "gateway" in self.me:
return self.me["gateway"]
return None
def dns(self):
if "dns" in self.me:
return self.me["dns"]
return None
def neo_pin(self):
if "neo_pin" in self.me:
```

```
return self.me["neo_pin"]
return None
def neo_how_many(self):
if "neo_how_many" in self.me:
return self.me["neo_how_many"]
return None
def set_ip(self, sta):
if "ip" in self.me:
sta.ifconfig((self.me["ip"], self.me["mask"], self.me["gateway"], self.me["dns"]))
```

Here is alice.cfg (copy it to whoami.cfg on the transmitter):

```
{"name":"Alice","neo_pin":21,"neo_how_many":1, "gdo0":10, "gdo2":9, "csn":15, "sck":12, "mosi":11,
"miso":13, "baud":38.4, "fec":1}
```

Here is bob.cfg (copied to whoami.cfg on the receiver):

```
{"name":"Bob","neo_pin":21,"neo_how_many":1, "gdo0":4, "gdo2":1, "csn":3, "sck":8, "mosi":10,
"miso":9, "baud":38.4, "fec":1}
```

On Windows, I use this script to copy the files:

```
set comport=%1
if "%comport%" == "com4" copy alice.cfg whoami.cfg
if "%comport%" == "com5" copy bob.cfg whoami.cfg
mpremote connect %comport% rm main.py
mpremote connect %comport% rm whoamy.py
mpremote connect %comport% rm whoamy.cfg
mpremote connect %comport% cp whoami.py :whoami.py
mpremote connect %comport% cp whoami.cfg :whoami.cfg
mpremote connect %comport% cp main.py :main.py
```

You don’t actually need the lines that remove the old files. They are there because, during debugging, sometimes the processor would start up before the next file could be sent, and removing main.py stopped that.