# Python Radio 20: The CC1101 Module

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--dd94bf0b09b8---------------------------------------)

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--dd94bf0b09b8---------------------------------------)

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

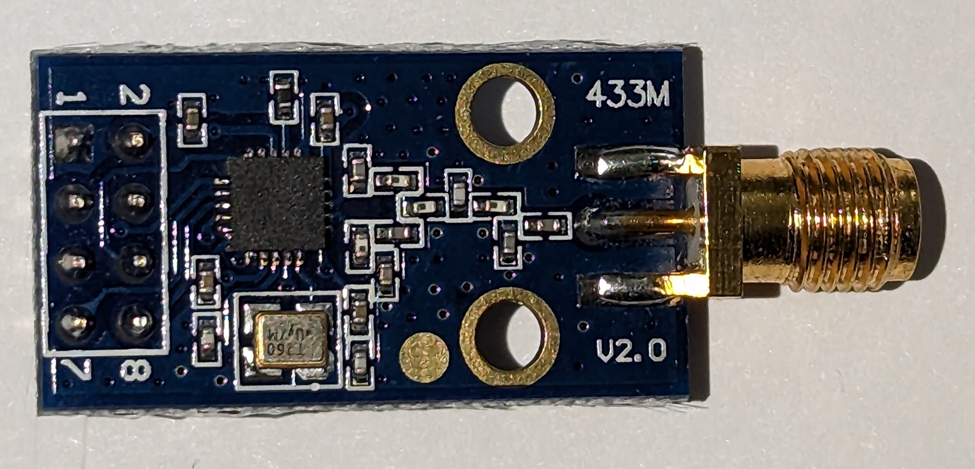


Photo by the author

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 the European license-free ISM band (433.05 MHz to 434.79 MHz), as well as the U.S. Amateur Radio 70 cm 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. 915 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 packetized modes with cyclic redundancy checks, preambles, sync words, forward error correction, interleaving, and more.

Press enter or click to view image in full size

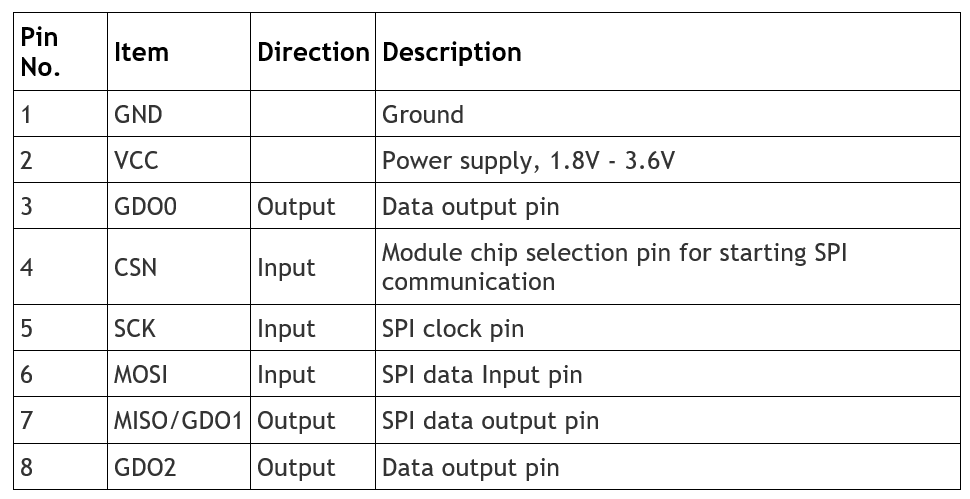


Image by the author

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 13 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 can manage half a megabit (500 kBaud) in MSK mode and a quarter megabit (250 kBaud) in GFSK mode. At my 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 transmitter (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 data 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.