# Python Radio 16: MFSK Mode

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--3d1aa2669c71---------------------------------------)

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--3d1aa2669c71---------------------------------------)

Follow

20 min read

·

Sep 3, 2024

Listen

Share

More

Bigger and faster…

Press enter or click to view image in full size

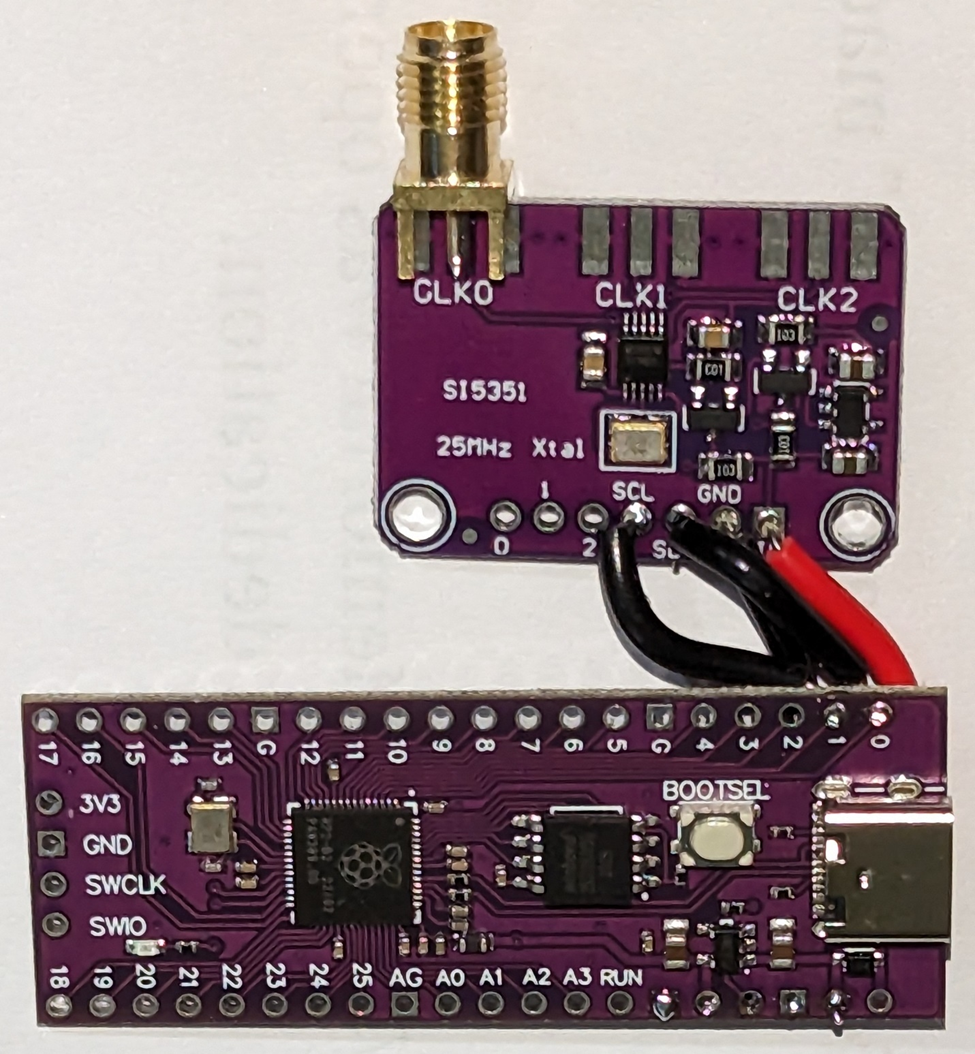


Photo by the author

In the last article, we hinted at the Si5351 module that will allow us to send on any frequency we like, with a resolution of less than one Hertz.

In this article, we get to use it.

We will connect the Si5351 module (available for about a dollar on AliExpress, or $8 on Amazon.com) to the RP2040 using four wires.

The module uses the I2C interface, so all it needs are five-volt power, ground, a clock signal, and a data signal. The ESP32 and ESP8266 can also use this module, as they both support the I2C interface.

Like those microprocessors, the SI5351 also produces square waves by dividing a clock by integers, but it has much better resolution. Below 18 megahertz, it will never be more than half a hertz off of the target frequency. Even at 150 MHz the error is only 4 hertz, and 18% of the frequencies in that range don’t miss at all. The module has a nominal range of 8 kilohertz to 160 megahertz, but it can actually reach the 1.25-meter band up around 220 megahertz.

With the SI5351, we can transmit in modes that send multiple tones only a few hertz apart, such as MFSK.

As an added bonus, the SI5351 puts out 50 milliwatts of power, instead of the 3.8 milliwatts we got out of our raw RP2040. You can claim the 1,000 miles per watt award by having a conversation with someone 50 miles away, with no amplifier, powered by USB. QRP is anything 5 watts or less. QRPp is less than a watt, down to 100 milliwatts. Below that is called QRPpp, and our 50-milliwatt transmitter is in that class. You can, of course, add an amplifier if you feel the need for more power.

To use the module, we need a driver. Device drivers are difficult to understand without reading the entire specification sheet for the device, and so I won’t be going into any detail here. The following code is in the module SI5351.py:

from machine import I2C  
import math  
  
SI5351\_REGISTER\_0\_DEVICE\_STATUS = 0  
SI5351\_REGISTER\_1\_INTERRUPT\_STATUS\_STICKY = 1  
SI5351\_REGISTER\_2\_INTERRUPT\_STATUS\_MASK = 2  
SI5351\_REGISTER\_3\_OUTPUT\_ENABLE\_CONTROL = 3  
SI5351\_REGISTER\_9\_OEB\_PIN\_ENABLE\_CONTROL = 9  
SI5351\_REGISTER\_15\_PLL\_INPUT\_SOURCE = 15  
SI5351\_REGISTER\_16\_CLK0\_CONTROL = 16  
SI5351\_REGISTER\_17\_CLK1\_CONTROL = 17  
SI5351\_REGISTER\_18\_CLK2\_CONTROL = 18  
SI5351\_REGISTER\_19\_CLK3\_CONTROL = 19  
SI5351\_REGISTER\_20\_CLK4\_CONTROL = 20  
SI5351\_REGISTER\_21\_CLK5\_CONTROL = 21  
SI5351\_REGISTER\_22\_CLK6\_CONTROL = 22  
SI5351\_REGISTER\_23\_CLK7\_CONTROL = 23  
SI5351\_REGISTER\_24\_CLK3\_0\_DISABLE\_STATE = 24  
SI5351\_REGISTER\_25\_CLK7\_4\_DISABLE\_STATE = 25  
SI5351\_REGISTER\_42\_MULTISYNTH0\_PARAMETERS\_1 = 42  
SI5351\_REGISTER\_43\_MULTISYNTH0\_PARAMETERS\_2 = 43  
SI5351\_REGISTER\_44\_MULTISYNTH0\_PARAMETERS\_3 = 44  
SI5351\_REGISTER\_45\_MULTISYNTH0\_PARAMETERS\_4 = 45  
SI5351\_REGISTER\_46\_MULTISYNTH0\_PARAMETERS\_5 = 46  
SI5351\_REGISTER\_47\_MULTISYNTH0\_PARAMETERS\_6 = 47  
SI5351\_REGISTER\_48\_MULTISYNTH0\_PARAMETERS\_7 = 48  
SI5351\_REGISTER\_49\_MULTISYNTH0\_PARAMETERS\_8 = 49  
SI5351\_REGISTER\_50\_MULTISYNTH1\_PARAMETERS\_1 = 50  
SI5351\_REGISTER\_51\_MULTISYNTH1\_PARAMETERS\_2 = 51  
SI5351\_REGISTER\_52\_MULTISYNTH1\_PARAMETERS\_3 = 52  
SI5351\_REGISTER\_53\_MULTISYNTH1\_PARAMETERS\_4 = 53  
SI5351\_REGISTER\_54\_MULTISYNTH1\_PARAMETERS\_5 = 54  
SI5351\_REGISTER\_55\_MULTISYNTH1\_PARAMETERS\_6 = 55  
SI5351\_REGISTER\_56\_MULTISYNTH1\_PARAMETERS\_7 = 56  
SI5351\_REGISTER\_57\_MULTISYNTH1\_PARAMETERS\_8 = 57  
SI5351\_REGISTER\_58\_MULTISYNTH2\_PARAMETERS\_1 = 58  
SI5351\_REGISTER\_59\_MULTISYNTH2\_PARAMETERS\_2 = 59  
SI5351\_REGISTER\_60\_MULTISYNTH2\_PARAMETERS\_3 = 60  
SI5351\_REGISTER\_61\_MULTISYNTH2\_PARAMETERS\_4 = 61  
SI5351\_REGISTER\_62\_MULTISYNTH2\_PARAMETERS\_5 = 62  
SI5351\_REGISTER\_63\_MULTISYNTH2\_PARAMETERS\_6 = 63  
SI5351\_REGISTER\_64\_MULTISYNTH2\_PARAMETERS\_7 = 64  
SI5351\_REGISTER\_65\_MULTISYNTH2\_PARAMETERS\_8 = 65  
SI5351\_REGISTER\_66\_MULTISYNTH3\_PARAMETERS\_1 = 66  
SI5351\_REGISTER\_67\_MULTISYNTH3\_PARAMETERS\_2 = 67  
SI5351\_REGISTER\_68\_MULTISYNTH3\_PARAMETERS\_3 = 68  
SI5351\_REGISTER\_69\_MULTISYNTH3\_PARAMETERS\_4 = 69  
SI5351\_REGISTER\_70\_MULTISYNTH3\_PARAMETERS\_5 = 70  
SI5351\_REGISTER\_71\_MULTISYNTH3\_PARAMETERS\_6 = 71  
SI5351\_REGISTER\_72\_MULTISYNTH3\_PARAMETERS\_7 = 72  
SI5351\_REGISTER\_73\_MULTISYNTH3\_PARAMETERS\_8 = 73  
SI5351\_REGISTER\_74\_MULTISYNTH4\_PARAMETERS\_1 = 74  
SI5351\_REGISTER\_75\_MULTISYNTH4\_PARAMETERS\_2 = 75  
SI5351\_REGISTER\_76\_MULTISYNTH4\_PARAMETERS\_3 = 76  
SI5351\_REGISTER\_77\_MULTISYNTH4\_PARAMETERS\_4 = 77  
SI5351\_REGISTER\_78\_MULTISYNTH4\_PARAMETERS\_5 = 78  
SI5351\_REGISTER\_79\_MULTISYNTH4\_PARAMETERS\_6 = 79  
SI5351\_REGISTER\_80\_MULTISYNTH4\_PARAMETERS\_7 = 80  
SI5351\_REGISTER\_81\_MULTISYNTH4\_PARAMETERS\_8 = 81  
SI5351\_REGISTER\_82\_MULTISYNTH5\_PARAMETERS\_1 = 82  
SI5351\_REGISTER\_83\_MULTISYNTH5\_PARAMETERS\_2 = 83  
SI5351\_REGISTER\_84\_MULTISYNTH5\_PARAMETERS\_3 = 84  
SI5351\_REGISTER\_85\_MULTISYNTH5\_PARAMETERS\_4 = 85  
SI5351\_REGISTER\_86\_MULTISYNTH5\_PARAMETERS\_5 = 86  
SI5351\_REGISTER\_87\_MULTISYNTH5\_PARAMETERS\_6 = 87  
SI5351\_REGISTER\_88\_MULTISYNTH5\_PARAMETERS\_7 = 88  
SI5351\_REGISTER\_89\_MULTISYNTH5\_PARAMETERS\_8 = 89  
SI5351\_REGISTER\_90\_MULTISYNTH6\_PARAMETERS = 90  
SI5351\_REGISTER\_91\_MULTISYNTH7\_PARAMETERS = 91  
SI5351\_REGISTER\_092\_CLOCK\_6\_7\_OUTPUT\_DIVIDER = 92  
SI5351\_REGISTER\_165\_CLK0\_INITIAL\_PHASE\_OFFSET = 165  
SI5351\_REGISTER\_166\_CLK1\_INITIAL\_PHASE\_OFFSET = 166  
SI5351\_REGISTER\_167\_CLK2\_INITIAL\_PHASE\_OFFSET = 167  
SI5351\_REGISTER\_168\_CLK3\_INITIAL\_PHASE\_OFFSET = 168  
SI5351\_REGISTER\_169\_CLK4\_INITIAL\_PHASE\_OFFSET = 169  
SI5351\_REGISTER\_170\_CLK5\_INITIAL\_PHASE\_OFFSET = 170  
SI5351\_REGISTER\_177\_PLL\_RESET = 177  
SI5351\_REGISTER\_183\_CRYSTAL\_INTERNAL\_LOAD\_CAPACITANCE = 183  
  
SI5351\_CRYSTAL\_FREQ\_25MHZ = 25000000  
SI5351\_CRYSTAL\_FREQ\_27MHZ = 27000000  
SI5351\_CRYSTAL\_LOAD\_6PF = 1<<6  
SI5351\_CRYSTAL\_LOAD\_8PF = 2<<6  
SI5351\_CRYSTAL\_LOAD\_10PF = 3<<6  
  
si5351\_15to92 = bytearray(b’\x00OOo\x80\x80\x80\x80\x80\x00\x00\x00\x05\x00\x0cf\x00\x00\x02\x02q\x00\x0c\x1a\x00\x00\x86\x00\x01\x00\x01\x00\x00\x00\x00\x00\x01\x00\x1c\x00\x00\x00\x00\x00\x01\x00\x18\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00’)  
  
SI5351\_MULTISYNTH\_DIV\_4 = 4  
SI5351\_MULTISYNTH\_DIV\_6 = 6  
SI5351\_MULTISYNTH\_DIV\_8 = 8  
  
class SI5351:  
 def \_\_init\_\_(self, i2c, address=0x60, crystalFreq=25000000):  
 self.i2c = i2c  
 self.address = address  
   
 self.initialized = False  
 self.crystalFreq = crystalFreq  
 self.crystalLoad = SI5351\_CRYSTAL\_LOAD\_10PF  
 self.crystalPPM = 30  
 self.plla\_configured = False  
 self.plla\_freq = 0  
 self.pllb\_configured = False  
 self.pllb\_freq = 0  
 return  
  
 def write8(self, register, value):  
 ret = True  
 self.i2c.start() # only available in SoftI2C  
 buffera = bytearray(1)  
 buffera[0] = value & 0xff  
 try:  
 self.i2c.writeto\_mem(self.address, register, buffera)  
 except Exception as e:  
 print(”Exception”, e, “when writing to Si5351”)  
 print(”Address: “, self.address)  
 print(”Scan: “, self.i2c.scan())  
 ret = False  
 self.i2c.stop() # only available in SoftI2C  
 return ret  
  
 def read8(self, register, value):  
 ret = True  
 self.i2c.start() # only available in SoftI2C  
 buffera = bytearray(1)  
 try:  
 self.i2c.readfrom\_mem\_into(self.address, register, buffera)  
 except Exception as e:  
 print(”Exception”, e, “when writing to Si5351”)  
 print(”Address: “, self.address)  
 print(”Scan: “, self.i2c.scan())  
 ret = False  
 self.i2c.stop() # only available in SoftI2C  
 return ret  
  
  
 def begin(self):  
 self.write8(SI5351\_REGISTER\_3\_OUTPUT\_ENABLE\_CONTROL, 0xFF)  
 # Power down all output drivers \*/  
 self.write8(SI5351\_REGISTER\_16\_CLK0\_CONTROL, 0x80)  
 self.write8(SI5351\_REGISTER\_17\_CLK1\_CONTROL, 0x80)  
 self.write8(SI5351\_REGISTER\_18\_CLK2\_CONTROL, 0x80)  
 self.write8(SI5351\_REGISTER\_19\_CLK3\_CONTROL, 0x80)  
 self.write8(SI5351\_REGISTER\_20\_CLK4\_CONTROL, 0x80)  
 self.write8(SI5351\_REGISTER\_21\_CLK5\_CONTROL, 0x80)  
 self.write8(SI5351\_REGISTER\_22\_CLK6\_CONTROL, 0x80)  
 self.write8(SI5351\_REGISTER\_23\_CLK7\_CONTROL, 0x80)  
  
 # Set the load capacitance for the XTAL \*/  
 self.write8(SI5351\_REGISTER\_183\_CRYSTAL\_INTERNAL\_LOAD\_CAPACITANCE,  
 self.crystalLoad)  
  
 # Set interrupt masks as required (see Register 2 description in AN619).  
 # By default, ClockBuilder Desktop sets this register to 0x18.  
 # Note that the least significant nibble must remain 0x8, but the most  
 # significant nibble may be modified to suit your needs.   
  
 # Reset the PLL config fields just in case we call init again  
 self.plla\_configured = False  
 self.plla\_freq = 0  
 self.pllb\_configured = False  
 self.pllb\_freq = 0  
  
 # All done!  
 self.initialized = True  
  
 return  
  
 def setClockBuilderData(self ):  
 i = 0  
  
 # Make sure we’ve called init first  
  
 assert self.initialized == True, “you have not initialized the object”  
  
 # Disable all outputs setting CLKx\_DIS high  
 self.write8(SI5351\_REGISTER\_3\_OUTPUT\_ENABLE\_CONTROL, 0xFF)  
  
 # Writes configuration data to device using the register map contents  
 # generated by ClockBuilder Desktop (registers 15-92 + 149-170)  
 for i, x in enumerate(range(15,93)):   
 #print(x, si5351\_15to92[i] )  
 self.write8(x, si5351\_15to92[i] )  
  
 for i in range(149, 171):  
 self.write8(i, 0x00)   
  
 # Apply soft reset  
 self.write8(SI5351\_REGISTER\_177\_PLL\_RESET, 0xAC)  
  
 # Enabled desired outputs (see Register 3)  
 self.write8(SI5351\_REGISTER\_3\_OUTPUT\_ENABLE\_CONTROL, 0x00)  
 return None  
  
 def setupPLL(self, mult, num, denom, pllsource = ‘A’, reset=True):  
 assert self.initialized == True, “you have not initialized the object”  
 assert ((mult > 14) and (mult < 91) ), “invalid mult parameter”  
 assert denom > 0, “denom must be > 0”  
 assert num <= 0xfffff, “invalid parameter num”  
 assert denom <= 0xfffff, “invalid parameter denom”  
 if num ==0:  
 P1 = 128\*mult -512  
 P2 = num  
 P3 = denom  
 else:  
 P1 = 128\*mult + math.floor( 128 \* num/denom ) -512  
 P2 = 128\*num - denom \* math.floor( 128 \* num/denom)  
 P3 = denom   
  
 if pllsource == ‘A’:  
 baseaddr = 26  
 else:  
 baseaddr = 34  
  
 P1 = int(P1)  
 self.write8( baseaddr, (P3 & 0x0000FF00) >> 8)  
 self.write8( baseaddr+1, (P3 & 0x000000FF))  
 self.write8( baseaddr+2, (P1 & 0x00030000) >> 16)  
 self.write8( baseaddr+3, (P1 & 0x0000FF00) >> 8)  
 self.write8( baseaddr+4, (P1 & 0x000000FF))  
 self.write8( baseaddr+5, ((P3 & 0x000F0000) >> 12) | ((P2 & 0x000F0000) >> 16) )  
 self.write8( baseaddr+6, (P2 & 0x0000FF00) >> 8)  
 self.write8( baseaddr+7, (P2 & 0x000000FF))  
  
 if reset:  
 self.write8(SI5351\_REGISTER\_177\_PLL\_RESET, (1<<7) | (1<<5) )  
  
 if pllsource ==’A’:  
 fvco = self.crystalFreq\*( mult + num/denom)  
 self.plla\_configured = True  
 self.plla\_freq = int(math.floor( fvco))  
 else:  
 fvco = self.crystalFreq\*(mult + num/denom)  
 self.pllb\_configured = True  
 self.pllb\_freq = int(math.floor(fvco))  
 return None  
  
 def setupRdiv( self, output, div):  
 assert output in [0,1,2], “output value invalid”  
 assert div in [1,2,4,8,16,32,64,128], “div invalid”  
 divdict = {1: 0, 2: 1, 4: 2, 8: 3, 16: 4, 32: 5, 64: 6, 128: 7}  
 registers = [ 44, 52, 60]  
 Rreg = registers[output]  
 buf = bytearray( 1)  
  
 self.read8(Rreg, buf)  
  
 regval = buf[0] & 0x0F  
 divider = divdict[div]  
 divider &= 0x07  
 divider <<= 4  
 regval |= divider  
 self.write8(Rreg, regval)  
  
 return None  
  
 def setupMultisynth( self, output, div, num, denom, pllsource, power=3):  
 assert self.initialized == True, “device not initialized”  
 assert output in [0,1,2], “output out of range”  
 assert div > 3, “div out of range”  
 assert denom >0, “denom out of range”  
 assert num <= 0xfffff, “num has a 20-bit limit”  
 assert denom <= 0xfffff, “denom as a 20-bit limit”  
 if pllsource==”A”:  
 assert self.plla\_configured == True, “plla has not been configured”  
 else:  
 assert self.pllb\_configured == True, ‘pllb has not been configured’  
  
 # Output Multisynth Divider Equations  
 # where: a = div, b = num and c = denom  
 #  
 # P1 register is an 18-bit value using following formula:  
 #  
 # P1[17:0] = 128 \* a + floor(128\*(b/c)) - 512  
 #  
 # P2 register is a 20-bit value using the following formula:  
 #  
 # P2[19:0] = 128 \* b - c \* floor(128\*(b/c))  
 #  
 # P3 register is a 20-bit value using the following formula:  
 #  
 # P3[19:0] = c  
  
 if num==0:  
 # integer mode  
 P1 = int(128 \* div - 512)  
 P2 = num  
 P3 = denom  
 else:  
 # Fractional mode \*/  
 P1 = int( 128 \* div + math.floor(128 \* (num/denom)) - 512 )  
 P2 = int( 128 \* num - denom \* math.floor(128 \* (num/denom)))  
 P3 = denom  
  
  
 baseaddrs = [ 42, 50, 58]  
 baseaddr = baseaddrs[output]  
  
 self.write8( baseaddr, (P3 & 0x0000FF00) >> 8)  
 self.write8( baseaddr+1, (P3 & 0x000000FF))  
 self.write8( baseaddr+2, (P1 & 0x00030000) >> 16) # ToDo: Add DIVBY4 (>150MHz) and R0 support (<500kHz) later \*/  
 self.write8( baseaddr+3, (P1 & 0x0000FF00) >> 8)  
 self.write8( baseaddr+4, (P1 & 0x000000FF))  
 self.write8( baseaddr+5, ((P3 & 0x000F0000) >> 12) | ((P2 & 0x000F0000) >> 16) )  
 self.write8( baseaddr+6, (P2 & 0x0000FF00) >> 8)  
 self.write8( baseaddr+7, (P2 & 0x000000FF))  
  
 # Configure the clk control and enable the output   
 clkControlReg = 0x0F # 8mA drive strength, MS0 as CLK0 source, Clock not inverted, powered up   
 if pllsource == ‘B’:  
 clkControlReg |= (1 << 5) # /\* Uses PLLB \*/  
 if num == 0:  
 clkControlReg |= (1 << 6) # Integer mode \*/  
  
 #  
 # Set power level bits  
 # 0: 2mA -8dB  
 # 1: 4mA -3dB  
 # 2: 6mA -1dB  
 # 3: 8mA -0dB (default)  
 #  
 # clkControlReg |= power & 3  
  
 if output == 0:   
 self.write8(SI5351\_REGISTER\_16\_CLK0\_CONTROL, clkControlReg)  
 if output == 1:  
 self.write8(SI5351\_REGISTER\_17\_CLK1\_CONTROL, clkControlReg)  
 if output == 2:  
 self.write8(SI5351\_REGISTER\_18\_CLK2\_CONTROL, clkControlReg)  
  
  
 def enableOutputs( self, enabled=True):  
 assert self.initialized == True, “Error Device not initialized”  
 if enabled:  
 ret = self.write8( SI5351\_REGISTER\_3\_OUTPUT\_ENABLE\_CONTROL, 0x00)  
 else:  
 ret = self.write8( SI5351\_REGISTER\_3\_OUTPUT\_ENABLE\_CONTROL, 0xff)  
  
 return ret

Our radio.py module handles the details of using the SI5351 to output square waves at the frequencies we choose. The SI5351 has two clocks, called A and B in the spec sheet, but we will refer to them as 0 and 1. For the most part, we will only deal with clock 0. Likewise, it has three outputs, but we will only use the first one, labeled CLK0.

Here is the code:

import utime  
  
class Radio:  
 def \_\_init\_\_(self):  
 from SI5351 import SI5351  
 from machine import Pin, SoftI2C  
  
 # self.i2c = SoftI2C(scl=Pin(22), sda=Pin(21), freq=400000) # ESP32  
 self.i2c = SoftI2C(scl=Pin(1), sda=Pin(0), freq=400000) # RP2040  
  
 # self.i2c = SoftI2C(scl=Pin(22), sda=Pin(21), freq=800000) # The ESP32 can do up to 5 MHz best case  
 print( “I2C.scan():”, self.i2c.scan())  
 for x in range(5):  
 self.clockgen = SI5351(self.i2c, 0x60 + x)  
 status = 0  
 if self.clockgen.read8(0, status):  
 break  
 self.clockgen.begin()  
 self.clockgen.setClockBuilderData()  
 self.key\_state = False  
 self.actual\_freq\_a = 0  
 self.actual\_freq\_b = 0  
 self.nominal\_freq\_a = 0  
 self.nominal\_freq\_b = 0  
 # self.last\_time = utime.ticks\_ms()  
  
 self.old\_mult = 0  
 self.old\_num = 0  
 self.old\_denom = 0  
 self.old\_src = 0  
  
 def gcd(self, x, y):  
 while(y):   
 x, y = y, x % y   
 return x   
  
 def send(self):  
 pass  
  
 def info(self):  
 print( “I2C.scan():”, self.i2c.scan())  
   
 def on(self):  
 if self.clockgen.enableOutputs(True):  
 self.key\_state = True  
  
 def off(self):  
 if self.clockgen.enableOutputs(False):  
 self.key\_state = False  
  
 def key\_down(self):  
 if self.clockgen.enableOutputs(True):  
 self.key\_state = True  
  
 def key\_up(self):  
 if self.clockgen.enableOutputs(False):  
 self.key\_state = False  
  
 def get\_freq(self, which):  
 if which == 0:  
 return self.nominal\_freq\_a  
 else:  
 return self.nominal\_freq\_b  
  
 def set\_freq(self, which, f):  
 f = float(f)  
 div = int(900000000.0 / f) # Values under a megahertz need an extra divide step  
 r = 1  
 while div > 900:  
 r \*= 2  
 div /= 2  
   
 if div % 2: # Make sure it is an even number  
 div -= 1  
   
 pllFreq = div \* r \* f  
   
 xtal\_freq = 25000000 # Our board uses a 25 MHz crystal  
 fmult = pllFreq / xtal\_freq # The full multiplier  
 mult = int(fmult) # The integer part of the multiplier  
 frac = fmult - mult  
 off = int(frac \* xtal\_freq)  
 divisor = self.gcd(off, xtal\_freq)  
 num = int(off / divisor)  
 denom = int(xtal\_freq / divisor)  
   
 if num > 0xFFFFF or denom > 0xFFFFF:  
 denom = 0xFFFFF # Use the maximum value for the denominator  
 num = int((pllFreq % xtal\_freq) \* denom / xtal\_freq)  
  
  
 # Below 18 MHz, we will never be more than half a Hertz off  
 # Below 37.5 MHz, we will never be more than a Hertz off  
 # Below 75 MHz, we will never be more than two Hertz off  
 # Below 112.5 MHz, we will never be more than three Hertz off  
 # Below 150 MHz, we will never be more than four Hertz off  
 # Below 222 MHz, we will never be more than six Hertz off  
 # A little over 18% of the frequencies were right on the money  
 # This is better than the frequency stability of a temperature controlled crystal oscillator  
 # so any failure of accuracy here will be swamped by the variability in the oscillator  
 # Of course, if you have a nice OCXO crystal oscillator in an oven that has a parts-per-billion accuracy  
 # then you might want to know that you will never be off by more than 4 Hz in the 200 Hz wide WSPR window  
 # in the 2 meter band.  
  
 # All that assumes that we have double precision arithmetic. Micropython on the ESP8266 only has single precision 32 bit floats.  
 # So we can think we are off by as much as 5 Hz in the 40-meter band, when the Si5351 is actually much more accurate.  
 # On the ESP32, I have built special micropython firmware that supports double precision artithmetic.  
  
 # If r is 1, we will never be less than our target frequency  
  
 if which == 0:  
 self.actual\_freq\_a = (mult \* xtal\_freq + xtal\_freq \* num / denom) / div \* r  
 self.nominal\_freq\_a = f  
 src = "A"  
 else:  
 self.actual\_freq\_b = (mult \* xtal\_freq + xtal\_freq \* num / denom) / div \* r  
 self.nominal\_freq\_b = f  
 src = "B"  
  
 # print("Mult is", mult, "Num is", num, "Denom is", denom, "Src is", src, "Div is", div)  
  
 reset = False  
 if mult != self.old\_mult and num != self.old\_num and denom != self.old\_denom and src != self.old\_src:  
 reset = True  
  
 self.clockgen.setupPLL(mult, num, denom, pllsource=src, reset=reset)  
 self.old\_mult = mult  
 self.old\_num = num  
 self.old\_denom = denom  
 self.old\_src = src  
  
 self.clockgen.setupMultisynth(output=0, div=div, num=0, denom=1, pllsource=src)  
 if r > 1:  
 self.clockgen.setupRdiv(output=0, div=r)

Most of the code is in the set\_freq() method, and half of that is a comment.

The \_\_init\_\_() method handles setting up the I2C interface. We use the software version of I2C, because it is the same on all the microprocessors, so we don’t have to special case any code (other than which pins to use).

To get the extra resolution that we bought the module for, the code sets up more than the one integer divider we had in the RP2040. There is a numerator integer, a denominator integer, and a multiplier. There is an extra division step if the frequency is under a megahertz. All of this is so we have enough bits of precision to hit any frequency we wish in a large range.

MFSK stands for Multiple Frequency Shift Keying. Instead of just two tones, as we had in RTTY, sets of either 16 tones or 32 tones are used. The spacing between the tones is very narrow (such as 3.9065 hertz apart), so the bandwidth is quite low. Baud rates range from just under 4 to 125.

Unlike RTTY MFSK has a large character set, made up of 255 characters. All of the ASCII characters are there, as well as many characters for international keyboards, symbols for currency, degrees, and many more.

The characters are encoded in what is called “varicode”, where the more frequently used characters can be sent using fewer bits. Letters like “e” and “t” use only four bits. Characters such as “%” and “&” use ten bits.

Here is the whole table, in a module called mfsk\_varicode.py:

# -\*- coding: latin-1 -\*-  
mfsk\_varicode = [  
 0b11101011100, # 000 - <NUL>  
 0b11101100000, # 001 - <SOH>  
 0b11101101000, # 002 - <STX>  
 0b11101101100, # 003 - <ETX>  
 0b11101110000, # 004 - <EOT>  
 0b11101110100, # 005 - <ENQ>  
 0b11101111000, # 006 - <ACK>  
 0b11101111100, # 007 - <BEL>  
 0b10101000, # 008 - <BS>  
 0b11110000000, # 009 - <TAB>  
 0b11110100000, # 010 - <LF>  
 0b11110101000, # 011 - <VT>  
 0b11110101100, # 012 - <FF>  
 0b10101100, # 013 - <CR>  
 0b11110110000, # 014 - <SO>  
 0b11110110100, # 015 - <SI>  
 0b11110111000, # 016 - <DLE>  
 0b11110111100, # 017 - <DC1>  
 0b11111000000, # 018 - <DC2>  
 0b11111010000, # 019 - <DC3>  
 0b11111010100, # 020 - <DC4>  
 0b11111011000, # 021 - <NAK>  
 0b11111011100, # 022 - <SYN>  
 0b11111100000, # 023 - <ETB>  
 0b11111101000, # 024 - <CAN>  
 0b11111101100, # 025 - <EM>  
 0b11111110000, # 026 - <SUB>  
 0b11111110100, # 027 - <ESC>  
 0b11111111000, # 028 - <FS>  
 0b11111111100, # 029 - <GS>  
 0b100000000000, # 030 - <RS>  
 0b101000000000, # 031 - <US>  
 0b100, # 032 - <SPC>  
 0b111000000, # 033 - !  
 0b111111100, # 034 - '"'  
 0b1011011000, # 035 - #  
 0b1010101000, # 036 - $  
 0b1010100000, # 037 - %  
 0b1000000000, # 038 - &  
 0b110111100, # 039 - '  
 0b111110100, # 040 - (  
 0b111110000, # 041 - )  
 0b1010110100, # 042 - \*  
 0b111100000, # 043 - +  
 0b10100000, # 044 - ,  
 0b111011000, # 045 - -  
 0b111010100, # 046 - .  
 0b111101000, # 047 - /  
 0b11100000, # 048 - 0  
 0b11110000, # 049 - 1  
 0b101000000, # 050 - 2  
 0b101010100, # 051 - 3  
 0b101110100, # 052 - 4  
 0b101100000, # 053 - 5  
 0b101101100, # 054 - 6  
 0b110100000, # 055 - 7  
 0b110000000, # 056 - 8  
 0b110101100, # 057 - 9  
 0b111101100, # 058 - :  
 0b111111000, # 059 - ;  
 0b1011000000, # 060 - <  
 0b111011100, # 061 - =  
 0b1010111100, # 062 - >  
 0b111010000, # 063 - ?  
 0b1010000000, # 064 - @  
 0b10111100, # 065 - A  
 0b100000000, # 066 - B  
 0b11010100, # 067 - C  
 0b11011100, # 068 - D  
 0b10111000, # 069 - E  
 0b11111000, # 070 - F  
 0b101010000, # 071 - G  
 0b101011000, # 072 - H  
 0b11000000, # 073 - I  
 0b110110100, # 074 - J  
 0b101111100, # 075 - K  
 0b11110100, # 076 - L  
 0b11101000, # 077 - M  
 0b11111100, # 078 - N  
 0b11010000, # 079 - O  
 0b11101100, # 080 - P  
 0b110110000, # 081 - Q  
 0b11011000, # 082 - R  
 0b10110100, # 083 - S  
 0b10110000, # 084 - T  
 0b101011100, # 085 - U  
 0b110101000, # 086 - V  
 0b101101000, # 087 - W  
 0b101110000, # 088 - X  
 0b101111000, # 089 - Y  
 0b110111000, # 090 - Z  
 0b1011101000, # 091 - [  
 0b1011010000, # 092 - \  
 0b1011101100, # 093 - ]  
 0b1011010100, # 094 - ^  
 0b1010110000, # 095 - \_  
 0b1010101100, # 096 - `  
 0b10100, # 097 - a  
 0b1100000, # 098 - b  
 0b111000, # 099 - c  
 0b110100, # 100 - d  
 0b1000, # 101 - e  
 0b1010000, # 102 - f  
 0b1011000, # 103 - g  
 0b110000, # 104 - h  
 0b11000, # 105 - i  
 0b10000000, # 106 - j  
 0b1110000, # 107 - k  
 0b101100, # 108 - l  
 0b1000000, # 109 - m  
 0b11100, # 110 - n  
 0b10000, # 111 - o  
 0b1010100, # 112 - p  
 0b1111000, # 113 - q  
 0b100000, # 114 - r  
 0b101000, # 115 - s  
 0b1100, # 116 - t  
 0b111100, # 117 - u  
 0b1101100, # 118 - v  
 0b1101000, # 119 - w  
 0b1110100, # 120 - x  
 0b1011100, # 121 - y  
 0b1111100, # 122 - z  
 0b1011011100, # 123 - {  
 0b1010111000, # 124 - |  
 0b1011100000, # 125 - }  
 0b1011110000, # 126 - ~  
 0b101010000000, # 127 - <DEL>  
 0b101010100000, # 128 -   
 0b101010101000, # 129 -   
 0b101010101100, # 130 -   
 0b101010110000, # 131 -   
 0b101010110100, # 132 -   
 0b101010111000, # 133 -   
 0b101010111100, # 134 -   
 0b101011000000, # 135 -   
 0b101011010000, # 136 -   
 0b101011010100, # 137 -   
 0b101011011000, # 138 -   
 0b101011011100, # 139 -   
 0b101011100000, # 140 -   
 0b101011101000, # 141 -   
 0b101011101100, # 142 -   
 0b101011110000, # 143 -   
 0b101011110100, # 144 -   
 0b101011111000, # 145 -   
 0b101011111100, # 146 -   
 0b101100000000, # 147 -   
 0b101101000000, # 148 -   
 0b101101010000, # 149 -   
 0b101101010100, # 150 -   
 0b101101011000, # 151 -   
 0b101101011100, # 152 -   
 0b101101100000, # 153 -   
 0b101101101000, # 154 -   
 0b101101101100, # 155 -   
 0b101101110000, # 156 -   
 0b101101110100, # 157 -   
 0b101101111000, # 158 -   
 0b101101111100, # 159 -   
 0b1011110100, # 160 -   
 0b1011111000, # 161 - ¡  
 0b1011111100, # 162 - ¢  
 0b1100000000, # 163 - £  
 0b1101000000, # 164 - ¤  
 0b1101010000, # 165 - ¥  
 0b1101010100, # 166 - ¦  
 0b1101011000, # 167 - §  
 0b1101011100, # 168 - ¨  
 0b1101100000, # 169 - ©  
 0b1101101000, # 170 - ª  
 0b1101101100, # 171 - «  
 0b1101110000, # 172 - ¬  
 0b1101110100, # 173 - ­  
 0b1101111000, # 174 - ®  
 0b1101111100, # 175 - ¯  
 0b1110000000, # 176 - °  
 0b1110100000, # 177 - ±  
 0b1110101000, # 178 - ²  
 0b1110101100, # 179 - ³  
 0b1110110000, # 180 - ´  
 0b1110110100, # 181 - µ  
 0b1110111000, # 182 - ¶  
 0b1110111100, # 183 - ·  
 0b1111000000, # 184 - ¸  
 0b1111010000, # 185 - ¹  
 0b1111010100, # 186 - º  
 0b1111011000, # 187 - »  
 0b1111011100, # 188 - ¼  
 0b1111100000, # 189 - ½  
 0b1111101000, # 190 - ¾  
 0b1111101100, # 191 - ¿  
 0b1111110000, # 192 - À  
 0b1111110100, # 193 - Á  
 0b1111111000, # 194 - Â  
 0b1111111100, # 195 - Ã  
 0b10000000000, # 196 - Ä  
 0b10100000000, # 197 - Å  
 0b10101000000, # 198 - Æ  
 0b10101010000, # 199 - Ç  
 0b10101010100, # 200 - È  
 0b10101011000, # 201 - É  
 0b10101011100, # 202 - Ê  
 0b10101100000, # 203 - Ë  
 0b10101101000, # 204 - Ì  
 0b10101101100, # 205 - Í  
 0b10101110000, # 206 - Î  
 0b10101110100, # 207 - Ï  
 0b10101111000, # 208 - Ð  
 0b10101111100, # 209 - Ñ  
 0b10110000000, # 210 - Ò  
 0b10110100000, # 211 - Ó  
 0b10110101000, # 212 - Ô  
 0b10110101100, # 213 - Õ  
 0b10110110000, # 214 - Ö  
 0b10110110100, # 215 - ×  
 0b10110111000, # 216 - Ø  
 0b10110111100, # 217 - Ù  
 0b10111000000, # 218 - Ú  
 0b10111010000, # 219 - Û  
 0b10111010100, # 220 - Ü  
 0b10111011000, # 221 - Ý  
 0b10111011100, # 222 - Þ  
 0b10111100000, # 223 - ß  
 0b10111101000, # 224 - à  
 0b10111101100, # 225 - á  
 0b10111110000, # 226 - â  
 0b10111110100, # 227 - ã  
 0b10111111000, # 228 - ä  
 0b10111111100, # 229 - å  
 0b11000000000, # 230 - æ  
 0b11010000000, # 231 - ç  
 0b11010100000, # 232 - è  
 0b11010101000, # 233 - é  
 0b11010101100, # 234 - ê  
 0b11010110000, # 235 - ë  
 0b11010110100, # 236 - ì  
 0b11010111000, # 237 - í  
 0b11010111100, # 238 - î  
 0b11011000000, # 239 - ï  
 0b11011010000, # 240 - ð  
 0b11011010100, # 241 - ñ  
 0b11011011000, # 242 - ò  
 0b11011011100, # 243 - ó  
 0b11011100000, # 244 - ô  
 0b11011101000, # 245 - õ  
 0b11011101100, # 246 - ö  
 0b11011110000, # 247 - ÷  
 0b11011110100, # 248 - ø  
 0b11011111000, # 249 - ù  
 0b11011111100, # 250 - ú  
 0b11100000000, # 251 - û  
 0b11101000000, # 252 - n  
 0b11101010000, # 253 - ý  
 0b11101010100, # 254 - þ  
 0b11101011000 # 255 - ÿ  
]

The mfsk\_config.py module connects the Radio and MFSK classes and isolates the rest of the program from their details. Most of it is concerned with setting up the nine different modes, each with a different baud rate and number of tones and symbols. Otherwise, it looks much like the config module for RTTY:

from mfsk import MFSK  
from time import sleep\_ms, sleep  
  
from radio import Radio  
  
class MfskProcess:  
 def \_\_init\_\_(self, pin, frequency, baud, message, call, location):  
 from machine import Timer  
   
 self.osc = Radio()   
 self.radio\_timer = Timer()  
 self.old\_tone = -1  
 self.baud = baud  
 self.usb\_offset = 1133  
 self.frequency = frequency  
 self.message = message  
  
 self.r = MFSK(self.radio\_timer, self.send\_tone)  
 self.r.stop()  
 self.r.set\_call(call)  
 self.r.set\_location(location)  
 self.r.set\_frequency(frequency)  
 self.r.set\_message(message)  
  
 if self.baud == 4: # 3.90625 baud  
 self.r.samplerate = 8000.0  
 self.r.symlen = 2048.0  
 self.r.symbits = 5  
 self.r.depth = 5  
 self.r.basetone = 256  
 self.r.numtones = 32  
 self.r.preamble = 107  
 elif self.baud == 8: # 7.8125 baud  
 self.r.samplerate = 8000.0  
 self.r.symlen = 1024.0  
 self.r.symbits = 5  
 self.r.depth = 5  
 self.r.basetone = 128  
 self.r.numtones = 32  
 self.r.preamble = 107  
 elif self.baud == 11: # 10.7666015625 baud  
 self.r.samplerate = 11025.0  
 self.r.symlen = 1024.0  
 self.r.symbits = 4  
 self.r.depth = 10  
 self.r.basetone = 93  
 self.r.numtones = 16  
 self.r.preamble = 107  
 elif self.baud == 16: # 15.625 baud  
 self.r.samplerate = 8000.0  
 self.r.symlen = 512.0  
 self.r.symbits = 4  
 self.r.depth = 10  
 self.r.basetone = 64  
 self.r.numtones = 16  
 self.r.preamble = 107  
 elif self.baud == 22: # 21.533203125 baud  
 self.r.samplerate = 11025.0  
 self.r.symlen = 512.0  
 self.r.symbits = 4  
 self.r.depth = 10  
 self.r.basetone = 46  
 self.r.numtones = 16  
 self.r.preamble = 107  
 elif self.baud == 31: # 31.25 baud  
 self.r.samplerate = 8000.0  
 self.r.symlen = 256.0  
 self.r.symbits = 3  
 self.r.depth = 10  
 self.r.basetone = 32  
 self.r.numtones = 8  
 self.r.preamble = 107  
 elif self.baud == 32: # 31.25 baud  
 self.r.samplerate = 8000.0  
 self.r.symlen = 256.0  
 self.r.symbits = 4  
 self.r.depth = 10  
 self.r.basetone = 32  
 self.r.numtones = 16  
 self.r.preamble = 107  
 elif self.baud == 64: # 62.5 baud  
 self.r.samplerate = 8000.0  
 self.r.symlen = 128.0  
 self.r.symbits = 4  
 self.r.depth = 10  
 self.r.basetone = 16  
 self.r.numtones = 16  
 self.r.preamble = 180  
 elif self.baud == 128: # 125 baud  
 self.r.samplerate = 8000.0  
 self.r.symlen = 64.0  
 self.r.symbits = 4  
 self.r.depth = 20  
 self.r.basetone = 8  
 self.r.numtones = 16  
 self.r.preamble = 214  
  
 self.r.tonespacing = self.r.samplerate / self.r.symlen  
  
 print(”Frequency:”, self.frequency)  
 print(”Message:”, self.message)  
   
 print(”Symbits is”, self.r.symbits)  
 print(”Depth is”, self.r.depth)  
 print(”Bandwidth is”, (self.r.numtones - 1) \* self.r.tonespacing)  
 print(”Symbol length is”, self.r.symlen)  
 print(”Baud is”, self.r.samplerate / self.r.symlen)  
 print(”Tonespacing is”, str(self.r.tonespacing) + “:”)  
 self.send\_code()  
  
 def get\_radio(self):  
 return self.osc  
  
 def set\_message(self, msg):  
 self.message = msg  
   
 def send\_code(self):  
 self.f = float(self.r.basetone + float(self.frequency) + self.usb\_offset)  
 self.osc.set\_freq(0, self.f)  
  
 start\_of\_transmission\_length = int(8 \* (1000 / (self.r.samplerate / self.r.symlen)))  
 sleep\_ms(start\_of\_transmission\_length)  
 self.r.send\_code()  
  
 def send\_tone(self, tone):  
 if tone != self.old\_tone:  
 f = float(float(self.frequency) + self.usb\_offset + float(tone))  
 self.osc.set\_freq(0, f)  
 self.old\_tone = tone

The mfsk.py module handles the actual encoding of the data. This is quite involved, as there is forward error correction to allow it to handle noisy environments. The error correction code was developed by NASA for space probes. The bits are also interleaved (high bits are sent, then next high, etc.) to allow the error correction code to handle bursts of static that would otherwise damage several bits in a row. The bits are also Gray coded, so that adjacent symbols differ by only one bit, which helps to reduce errors in the decoding.

The code looks like this:

from mfsk\_varicode import mfsk\_varicode  
from machine import Timer  
  
class MFSK:  
  
 NASA\_K = 7  
 POLY1 = 0x6D  
 POLY2 = 0x4F  
  
 def \_\_init\_\_(self, timr, send\_tone):  
 self.timer = timr  
 self.send\_tone = send\_tone  
  
 #  
 # Default is MFSK4  
 #  
 self.symbits = 5  
 self.symlen = 2048  
 self.samplerate = 8000  
 self.depth = 5  
 self.basetone = 256  
 self.numtones = 32  
 self.preamble = 107  
 self.timer\_running = False  
  
 self.frequency = 7104000.0  
 self.call = ”  
 self.location = ”  
 self.message = “{} {} “  
 self.count\_tabs = 0  
 self.has\_bits = False  
 self.sym\_queue = []  
  
 # Initialization for the forward error correction  
 self.encoder\_output = [0] \* (1 << self.NASA\_K)  
 self.mask = (1 << self.NASA\_K) - 1  
 self.encode\_state = 0  
 self.bit\_count = 0  
 self.bit\_state = 0  
  
 # Code for the forward error correction  
 def init\_encoder(self):  
 self.interleave\_table = [8] \* (self.symbits \* self.symbits \* self.depth)  
 for x in range(1 << self.NASA\_K):  
 self.encoder\_output[x] = (self.parity(self.POLY1 & x) | (self.parity(self.POLY2 &x) << 1))  
 self.flush\_interleave\_table()  
  
 # Hamming weight (the number of bits that are ones)  
 def hamming\_weight(self, w):  
 w = (w & 0x55555555) + ((w >> 1) & 0x55555555)  
 w = (w & 0x33333333) + ((w >> 2) & 0x33333333)  
 w = (w & 0x0F0F0F0F) + ((w >> 4) & 0x0F0F0F0F)  
 w = (w & 0x00FF00FF) + ((w >> 8) & 0x00FF00FF)  
 w = (w & 0x0000FFFF) + ((w >> 16) & 0x0000FFFF)  
 return w  
  
 def parity(self, w):  
 return self.hamming\_weight(w) & 1  
  
 def encode(self, bit):  
 self.encode\_state <<= 1  
 if bit == “1”:  
 self.encode\_state |= 1  
  
 return self.encoder\_output[self.encode\_state & self.mask]  
  
 def set\_call(self, call):  
 self.call = call  
  
 def set\_baud(self, baud):  
 self.baud = float(baud)  
  
 def set\_bit\_length(self, len):  
 self.bit\_length = 1000000.0 / float(self.baud)  
  
 def set\_frequency(self, frequency):  
 self.frequency = float(frequency)  
  
 def set\_location(self, location):  
 self.location = location  
  
 def set\_message(self, message):  
 self.message = message.format(self.call, self.location)  
 self.message = “\r” + chr(2) + “\r” + self.message + “\r” + chr(0) + “\r”  
 self.has\_bits = True  
  
 def bit(self):  
 global mfsk\_varicode  
  
 for letter in self.message:  
 code = mfsk\_varicode[ord(letter) & 255]  
 for bit in bin(code)[2:]:  
 yield bit  
  
 def stop(self):  
 self.timer.deinit()  
 self.timer\_running = False  
 self.all\_done = True  
  
 def send\_code(self):  
 self.set\_baud(self.samplerate / self.symlen)  
 self.bit\_length = 1000000.0 / float(self.baud)  
 self.tonespacing = self.samplerate / self.symlen  
 self.bandwidth = (self.numtones - 1) \* self.tonespacing  
 self.init\_encoder()  
  
 self.all\_done = False  
 self.clearbits()  
 self.gen = self.bit()  
 if self.timer\_running == False:  
 self.timer.init(period=int(self.bit\_length/1000), mode=Timer.PERIODIC, callback=self.next\_tone)  
 self.timer\_running = True  
 self.reported\_end = False  
 self.has\_bits = True  
 while self.has\_bits:  
 bit = self.get\_bit()  
 self.send\_bit(bit)  
  
 self.flush\_tx(self.preamble)  
 self.reported\_end = True  
  
 def get\_bit(self):  
 try:  
 bit = next(self.gen)  
 except StopIteration as e:  
 self.has\_bits = False  
 return None  
 return bit  
  
 def send\_bit(self, bit):  
 try:  
 data = self.encode(bit)  
 for x in range(2):  
 self.bit\_state = (self.bit\_state << 1) | ((data >> x) & 1)  
 self.bit\_count += 1  
  
 if self.bit\_count == self.symbits:  
 self.interleave()  
 self.send\_symbol()  
 self.bit\_count = 0  
 self.bit\_state = 0  
 except Exception as e:  
 print(”Error:”, e)  
  
 def clearbits(self):  
 data = self.encode(0)  
 for x in range(self.preamble):  
 for y in range(2):  
 self.bit\_state = (self.bit\_state << 1) | ((data >> x) & 1)  
 self.bit\_count += 1  
 if self.bit\_count == self.symbits:  
 self.interleave()  
 self.bit\_count = 0  
 self.bit\_state = 0  
  
 def interleave\_get(self, x, y, z):  
 index = self.symbits \* self.symbits \* x + self.symbits \* y + z  
 return self.interleave\_table[index]  
  
 def interleave\_put(self, x, y, z, val):  
 index = self.symbits \* self.symbits \* x + self.symbits \* y + z  
 self.interleave\_table[index] = val  
  
 def symbols(self):  
 for x in range(self.depth):  
 for y in range(self.symbits):  
 for z in range(self.symbits - 1):  
 self.interleave\_put(x, y, z, self.interleave\_get(x, y, z + 1))  
  
 for y in range(self.symbits):  
 self.interleave\_put(x, y, self.symbits-1, self.syms[y])  
  
 for y in range(self.symbits):  
 self.syms[y] = self.interleave\_get(x, y, self.symbits - y - 1)  
  
 def interleave(self):  
 self.syms = []  
 for x in range(self.symbits):  
 self.syms.append(self.bit\_state >> ((self.symbits - x - 1)) & 1)  
  
 self.symbols()  
  
 self.bit\_state = 0  
 for x in range(self.symbits):  
 self.bit\_state = (self.bit\_state << 1) | self.syms[x]  
  
 def flush\_interleave\_table(self):  
 for x in range(len(self.interleave\_table)):  
 self.interleave\_table[x] = 0  
  
 def flush\_tx(self, preamble):  
 self.send\_bit(chr(1));  
 for x in range(preamble):  
 self.send\_bit(chr(0));  
 self.bit\_state = 0  
 self.all\_done = True  
  
 #  
 # In order to reduce the number of bit errors in a digital modem,  
 # all symbols are automatically Gray encoded such that adjacent  
 # symbols in a constellation differ by only one bit.  
 #  
 def gray\_encode(self, data):  
 bits = data;  
 bits ^= data >> 1;  
 bits ^= data >> 2;  
 bits ^= data >> 3;  
 bits ^= data >> 4;  
 bits ^= data >> 5;  
 bits ^= data >> 6;  
 bits ^= data >> 7;  
 return bits;  
  
 def send\_symbol(self):  
 from uasyncio import sleep\_ms  
 import utime  
 sym = self.bit\_state & (self.numtones - 1)  
 sym = self.gray\_encode(sym)  
 while len(self.sym\_queue) > 10:  
 # 256000 / 500 is 512 milliseconds (2 symbols at 3.90625 baud)  
 sleep\_ms(int(self.bit\_length / 500)) # Needed so the web server gets some time  
 utime.sleep\_ms(int(self.bit\_length / 500)) # Needed so ^C works  
 self.sym\_queue.append(sym)  
  
 def sendchar(self, ch):  
 code = mfsk\_varicode[ord(ch) & 255]  
 for bit in bin(code)[2:]:  
 self.send\_bit(bit);  
  
 def sendidle(self):  
 self.sendchar(chr(0));  
  
 def next\_tone(self, unused):  
 if self.sym\_queue:  
 sym = self.sym\_queue.pop(0)  
 self.send\_tone(self.basetone + sym \* self.tonespacing)

Finally, we get to our tiny little main.py module:

from mfsk\_config import MfskProcess  
from machine import Pin  
from time import sleep  
   
def main():  
 mp = MfskProcess(15, 7040000, 4, ”, “AB6NY”, “CM87xe”)  
 mp.set\_message(”{} Testing from {} using a Raspberry Pi Pico RP2040”)  
   
 while True:  
 mp.send\_code()  
 while mp.r.all\_done == False:  
 sleep(5)  
  
main()

It sets up pin 15 as the output, 7040000 as the frequency, and adds the call sign and Maidenhead Locator code to the message. Then it loops, sending the message over and over again, so that we can tune it in on the receiver.

The free software Fldigi program can receive and decode MFSK signals from your RTL-SDR. This makes testing much easier.

[Mfsk](https://medium.com/tag/mfsk?source=post_page-----3d1aa2669c71---------------------------------------)

[Python Programming](https://medium.com/tag/python-programming?source=post_page-----3d1aa2669c71---------------------------------------)