# Python Radio 19: Thor Mode

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--e087e368f3aa---------------------------------------)

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--e087e368f3aa---------------------------------------)

Follow

7 min read

·

Sep 6, 2024

Listen

Share

More

The best of both worlds…

Press enter or click to view image in full size

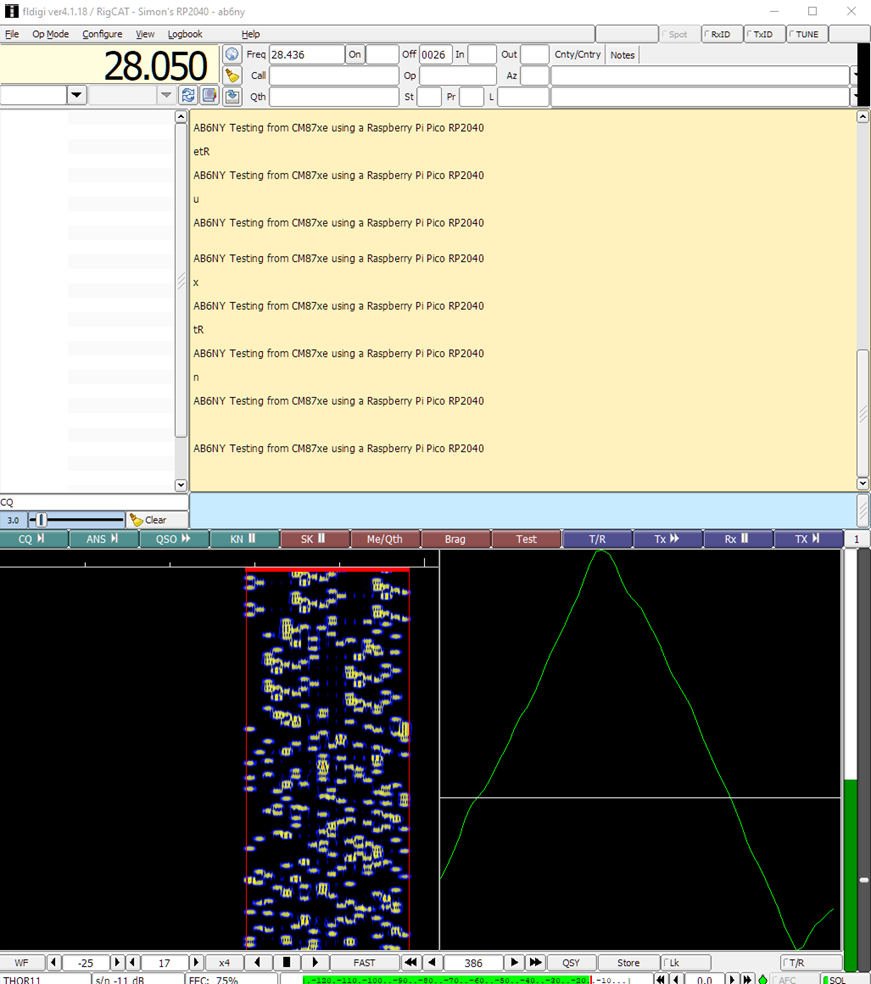


Image by the author

What if we combined the forward error correction of MFSK with the robustness and ease of tuning of FSQ and DominoEX?

We would get Thor.

The screenshot above shows Thor11, the 11 baud mode that gave us trouble in DominoEX.

The random characters in between the lines of perfect copy are noise from the radio during the periods when we weren’t sending.

Thor uses the same varicode as MFSK. It also has a shorter Thor varicode that is used to send callsigns when the system is idling. That looks like this:

thor\_varicode = [  
 0b101110000000, # 032 - <SPC>   
 0b101110100000, # 033 - !   
 0b101110101000, # 034 - ’   
 0b101110101100, # 035 - #   
 0b101110110000, # 036 - $   
 0b101110110100, # 037 - %   
 0b101110111000, # 038 - &   
 0b101110111100, # 039 - ‘   
 0b101111000000, # 040 - (   
 0b101111010000, # 041 - )   
 0b101111010100, # 042 - \*   
 0b101111011000, # 043 - +   
 0b101111011100, # 044 - ,   
 0b101111100000, # 045 - -   
 0b101111101000, # 046 - .   
 0b101111101100, # 047 - /   
 0b101111110000, # 048 - 0   
 0b101111110100, # 049 - 1   
 0b101111111000, # 050 - 2   
 0b101111111100, # 051 - 3   
 0b110000000000, # 052 - 4   
 0b110100000000, # 053 - 5   
 0b110101000000, # 054 - 6   
 0b110101010100, # 055 - 7   
 0b110101011000, # 056 - 8   
 0b110101011100, # 057 - 9   
 0b110101100000, # 058 - :   
 0b110101101000, # 059 - ;   
 0b110101101100, # 060 - <   
 0b110101110000, # 061 - =   
 0b110101110100, # 062 - >   
 0b110101111000, # 063 - ?   
 0b110101111100, # 064 - @   
 0b110110000000, # 065 - A   
 0b110110100000, # 066 - B   
 0b110110101000, # 067 - C   
 0b110110101100, # 068 - D   
 0b110110110000, # 069 - E   
 0b110110110100, # 070 - F   
 0b110110111000, # 071 - G   
 0b110110111100, # 072 - H   
 0b110111000000, # 073 - I   
 0b110111010000, # 074 - J   
 0b110111010100, # 075 - K   
 0b110111011000, # 076 - L   
 0b110111011100, # 077 - M   
 0b110111100000, # 078 - N   
 0b110111101000, # 079 - O   
 0b110111101100, # 080 - P   
 0b110111110000, # 081 - Q   
 0b110111110100, # 082 - R   
 0b110111111000, # 083 - S   
 0b110111111100, # 084 - T   
 0b111000000000, # 085 - U   
 0b111010000000, # 086 - V   
 0b111010100000, # 087 - W   
 0b111010101100, # 088 - X   
 0b111010110000, # 089 - Y   
 0b111010110100, # 090 - Z   
 0b111010111000, # 091 - [   
 0b111010111100, # 092 - \   
 0b111011000000, # 093 - ]   
 0b111011010000, # 094 - ^   
 0b111011010100, # 095 - \_   
 0b111011011000, # 096 - `   
 0b111011011100, # 097 - a   
 0b111011100000, # 098 - b   
 0b111011101000, # 099 - c   
 0b111011101100, # 100 - d   
 0b111011110000, # 101 - e   
 0b111011110100, # 102 - f   
 0b111011111000, # 103 - g   
 0b111011111100, # 104 - h   
 0b111100000000, # 105 - i   
 0b111101000000, # 106 - j   
 0b111101010000, # 107 - k   
 0b111101010100, # 108 - l   
 0b111101011000, # 109 - m   
 0b111101011100, # 110 - n   
 0b111101100000, # 111 - o   
 0b111101101000, # 112 - p   
 0b111101101100, # 113 - q   
 0b111101110000, # 114 - r   
 0b111101110100, # 115 - s   
 0b111101111000, # 116 - t   
 0b111101111100, # 117 - u   
 0b111110000000, # 118 - v   
 0b111110100000, # 119 - w   
 0b111110101000, # 120 - x   
 0b111110101100, # 121 - y   
 0b111110110000 # 122 - z   
]

The thor\_config.py module looks familiar by now:

from thor import THOR  
from time import sleep\_ms, sleep  
from machine import Timer  
from radio import Radio  
  
class ThorConfig:  
 def \_\_init\_\_(self, baud, frq, call, location):  
 self.dds = Radio()  
 self.dds.on()  
 self.dds.send()  
  
 self.radio\_timer = Timer()  
  
 self.baud = baud  
 self.message = ''  
 self.frequency = frq  
 self.usb\_offset = 1133  
 self.all\_done = False  
 self.call = call  
 self.location = location  
  
 self.r = THOR(self.radio\_timer, self.send\_tone, self.report\_all\_done)  
  
 self.r.set\_call(call)  
 self.r.set\_location(location)  
 self.r.set\_frequency(float(frq))  
 self.r.symbits = 4  
 self.r.depth = 10  
 self.r.numtones = 18  
 self.r.preamble = 4  
  
 if baud == 2: # 2.0 baud  
 self.r.samplerate = 8000.0  
 self.r.symlen = 4000.0  
 self.r.doublespaced = 1  
 self.r.depth = 4  
 elif baud == 4: # 3.90625 baud  
 self.r.samplerate = 8000.0  
 self.r.symlen = 2048.0  
 self.r.doublespaced = 2  
 elif baud == 5: # 5.38330078125 baud  
 self.r.samplerate = 11025.0  
 self.r.symlen = 2048.0  
 self.r.doublespaced = 2  
 elif baud == 8: # 7.8125 baud  
 self.r.samplerate = 8000.0  
 self.r.symlen = 1024.0  
 self.r.doublespaced = 2  
 elif baud == 11: # 10.7666015625 baud  
 self.r.samplerate = 11025.0  
 self.r.symlen = 1024.0  
 self.r.doublespaced = 1  
 elif baud == 16: # 15.625 baud  
 self.r.samplerate = 8000.0  
 self.r.symlen = 512.0  
 self.r.doublespaced = 1  
 elif baud == 22: # 21.533203125 baud  
 self.r.samplerate = 11025.0  
 self.r.symlen = 512.0  
 self.r.doublespaced = 1  
  
 self.r.tonespacing = self.r.samplerate / self.r.symlen  
  
 self.r.init\_params()  
  
 print(”Frequency:”, self.frequency)  
 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(”Bit length is”, self.r.bit\_length)  
 print(”Tonespacing is”, str(self.r.tonespacing))  
  
 def get\_radio(self):  
 return self.dss  
  
 def set\_message(self, msg):  
 self.r.set\_message(msg.format(self.call, self.location))  
 self.dds.on()  
 self.dds.send()  
 self.r.send\_code()  
  
 def send\_code(self):  
 self.r.send\_code()  
   
 def send\_tone(self, tone):  
 self.f = float(float(self.frequency) + self.usb\_offset + float(tone))  
 self.dds.set\_freq(0, self.f)  
 self.dds.on()  
 self.dds.send()  
  
 def report\_all\_done(self):  
 self.all\_done = True  
 print(”All done!”)

As with DominoEX, most of the code is just setting up the different baud rates.

The real work is done in the thor.py module:

from thor\_varicode import thor\_varicode  
from mfsk\_varicode import mfsk\_varicode  
from machine import Timer  
  
class THOR:  
  
 NASA\_K = 7  
 POLY1 = 0x6D  
 POLY2 = 0x4F  
  
 def \_\_init\_\_(self, timr, send\_tone, report\_message\_end=None):  
 self.timer = timr  
 self.send\_tone = send\_tone  
 self.report\_message\_end = report\_message\_end  
 self.QUEUE\_LENGTH = 80  
  
 #  
 # Default is THOR MICRO  
 #  
 self.symbits = 4  
 self.symlen = 4000  
 self.samplerate = 8000  
 self.depth = 4  
 self.doublespaced = 1  
 self.basetone = 256  
 self.numtones = 18  
 self.preamble = 10  
 self.timer\_running = False  
 self.bandwidth = 0  
 self.tonespacing = 0  
 self.secondary = False  
 self.previous\_tone = 0  
  
 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 = “\r” + chr(2) + “\r” + message + “\r” + chr(0) + “\r”  
 self.message += chr(0) + chr(0) + chr(0) + chr(0) + chr(0) + chr(0) + chr(0)  
 self.message += chr(0) + chr(0) + chr(0) + chr(0) + chr(0) + chr(0) + chr(0)  
 self.has\_bits = True  
  
 def init\_params(self):  
 self.set\_baud(self.samplerate / self.symlen)  
 self.tonespacing = self.samplerate \* self.doublespaced / self.symlen  
 self.bandwidth = (self.numtones - 1) \* self.tonespacing  
 self.basetone = int(1500.0 \* self.symlen / self.samplerate + 0.5)  
 self.bit\_length = 1000000.0 / float(self.baud)  
  
 def bit(self):  
 for letter in self.message:  
 code = mfsk\_varicode[0]  
 if self.secondary:  
 if ord(letter) >= 0 and ord(letter) < 256:  
 code = thor\_varicode[ord(letter) & 255]  
 else:  
 code = mfsk\_varicode[ord(letter) & 255]  
  
 for bit in bin(code)[2:]:  
 yield bit  
  
 def stop(self):  
 self.timer.deinit()  
 self.timer\_running = False  
  
 def send\_code(self):  
 self.init\_params()  
 self.init\_encoder()  
  
 self.is\_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  
  
 # Send 64 zero bits to flush the receive decoder  
 for x in range(16):  
 self.bit\_state = 0  
 self.send\_symbol()  
  
 while self.has\_bits:  
 bit = self.get\_bit()  
 self.send\_bit(bit)  
  
 self.flush\_tx(self.preamble)  
  
 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(1400):  
 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):  
 for x in range(preamble):  
 self.sendidle();  
 self.bit\_state = 0  
 self.is\_done = True  
 self.report\_message\_end()  
  
 def send\_symbol(self):  
 from time import sleep\_ms  
 sym = self.bit\_state  
 while len(self.sym\_queue) > self.QUEUE\_LENGTH:  
 sleep\_ms(int(self.bit\_length / 50000)) # Needed so ^C works  
 self.sym\_queue.append(sym)  
  
 def sendchar(self, letter, secondary):  
 code = mfsk\_varicode[0]  
 if self.secondary:  
 if ord(letter) >= 0 and ord(letter) < 256:  
 code = thor\_varicode[ord(letter) & 255]  
 else:  
 code = mfsk\_varicode[ord(letter) & 255]  
  
 for bit in bin(code)[2:]:  
 self.send\_bit(bit);  
  
 def sendidle(self):  
 for x in range(8):  
 self.sendchar(chr(0), 0);  
  
 def next\_tone(self, unused):  
 if self.sym\_queue:  
 sym = self.sym\_queue.pop(0)  
 tone = (self.previous\_tone + 2 + sym) % self.numtones  
 self.previous\_tone = tone  
 self.send\_tone(self.basetone + tone \* self.tonespacing)

A lot of the code is the NASA 7 forward error correction code borrowed from MFSK. Then we have our bit() generator and our send\_code() method.

The send\_code() method is a little bit larger than we are used to, since it has to worry about flushing the receiver’s decoder before sending real bits, and then sending the idle preamble.

Because of the forward error correction, the symbols are placed in a queue for sending, which is what send\_symbol() manages. The send\_char() method sends from either the Thor or the MFSK varicode tables, depending on its argument.

[Amateur Radio](https://medium.com/tag/amateur-radio?source=post_page-----e087e368f3aa---------------------------------------)

[Python Programming](https://medium.com/tag/python-programming?source=post_page-----e087e368f3aa---------------------------------------)