Python Radio 17: FSQ Mode

Simon Quellen Field

Simon Quellen Field

Follow

7 min read

·

Sep 4, 2024

Listen

Share

More

Fast Simple QSO

Press enter or click to view image in full size

Image by author

FSQ stands for Fast Simple QSO (QSO means conversation). It is a chat mode in amateur radio that has several advantages over MFSK and RTTY.

Like MFSK, it uses a varicode, so that frequently used characters are sent faster. It uses 33 tones so that all of the lower-case characters (and a few more, such as space, period, and carriage return) can be sent with a single tone. FSQ6 (the 6 is the baud rate) can send 60 words per minute if lowercase letters are used. The baud rate (number of symbols per second) can be anything from 2 to 6. A symbol is a single letter if in lowercase, and two symbols are needed for other characters. The receiver does not need to change anything as the baud rate changes, so the sender can adjust the baud rate to the propagation conditions, slowing down when there is a lot of noise. The slower speeds are more robust to interference and to weak signals.

Instead of fixed tones, FSQ mode uses the difference from the last tone. This makes it much easier to tune and makes it virtually immune to frequency drift. It also makes it less affected by inter-symbol interference that can occur in Near Vertical Incidence Skywave (NVIS) communication, where people can converse with other nearby radio operators over the horizon by bouncing the signal straight up off of the ionosphere.

It can send 104 ASCII characters, and takes up less than 300 hertz of bandwidth, helping it to have a good signal-to-noise ratio.

People using FSQ mode hang out at 3,588,000 hertz in the 80-meter band, 7,044,000 in the 40-meter band, and 10,144,000 in the 30-meter band.

Very low-power FSQ is also a good mode for exploring radio propagation. If your 50-milliwatt signal can be heard, then it is likely that any mode will work at that frequency and distance. This is known as MEPT. That stands for Manned Experimental Propagation Transmitter. Even though the computer is doing the sending, it is assumed that there is a person controlling and monitoring the transmissions, hence the “manned” part of the acronym.

It is also good for telemetry. You can add a sensor such as the DHT22 temperature-humidity sensor or the BME280 temperature-pressure-humidity sensor and transmit the readings to your whole neighborhood.

In FLDIGI, the reception looks like this:

Press enter or click to view image in full size

Image by author

You can see the 33 tones scattered across the waterfall display in the lower left and the decoded text above it.

We will describe here the modulator (as we did with other modes), but FSQ is usually used with the FSQCall program, which adds several automated functions making it especially useful for networks and emergency communications. We have hooks in the modulator for those features, such as directed messaging (call-sign to call-sign, as opposed to a message for everyone). FSQCall also provides error-corrected file transfers.

The varicode for FSQ has four tables. There is one for the 26 lowercase letters, space, period, and carriage return, making 28 tones. The remaining tones 29, 30, and 31 are used to select the other three tables, respectively. Since FSQ uses the difference between two tones as the symbol, the 33 tones allow for 32 symbols.

Here is the fsq\_varicode.py module:

Fsq\_varicode = {

‘ ‘: ( 0, 0),

‘a’: ( 1, 0),

‘b’: ( 2, 0),

‘c’: ( 3, 0),

‘d’: ( 4, 0),

‘e’: ( 5, 0),

‘f’: ( 6, 0),

‘g’: ( 7, 0),

‘h’: ( 8, 0),

‘i’: ( 9, 0),

‘j’: (10, 0),

‘k’: (11, 0),

‘l’: (12, 0),

‘m’: (13, 0),

‘n’: (14, 0),

‘o’: (15, 0),

‘p’: (16, 0),

‘q’: (17, 0),

‘r’: (18, 0),

‘s’: (19, 0),

‘t’: (20, 0),

‘u’: (21, 0),

‘v’: (22, 0),

‘w’: (23, 0),

‘x’: (24, 0),

‘y’: (25, 0),

‘z’: (26, 0),

‘.’: (27, 0),

‘\r’: (28, 0), # Carriage return and line feed (newline)

‘@’: ( 0, 29),

‘A’: ( 1, 29),

‘B’: ( 2, 29),

‘C’: ( 3, 29),

‘D’: ( 4, 29),

‘E’: ( 5, 29),

‘F’: ( 6, 29),

‘G’: ( 7, 29),

‘H’: ( 8, 29),

‘I’: ( 9, 29),

‘J’: (10, 29),

‘K’: (11, 29),

‘L’: (12, 29),

‘M’: (13, 29),

‘N’: (14, 29),

‘O’: (15, 29),

‘P’: (16, 29),

‘Q’: (17, 29),

‘R’: (18, 29),

‘S’: (19, 29),

‘T’: (20, 29),

‘U’: (21, 29),

‘V’: (22, 29),

‘W’: (23, 29),

‘X’: (24, 29),

‘Y’: (25, 29),

‘Z’: (26, 29),

‘,’: (27, 29),

‘?’: (28, 29),

‘~’: ( 0, 30),

‘1’: ( 1, 30),

‘2’: ( 2, 30),

‘3’: ( 3, 30),

‘4’: ( 4, 30),

‘5’: ( 5, 30),

‘6’: ( 6, 30),

‘7’: ( 7, 30),

‘8’: ( 8, 30),

‘9’: ( 9, 30),

‘0’: (10, 30),

‘!’: (11, 30),

‘”’: (12, 30),

‘#’: (13, 30),

‘$’: (14, 30),

‘%’: (15, 30),

‘&’: (16, 30),

‘\’’: (17, 30),

‘(‘: (18, 30),

‘)’: (19, 30),

‘\*’: (20, 30),

‘+’: (21, 30),

‘-‘: (22, 30),

‘/’: (23, 30),

‘:’: (24, 30),

‘;’: (25, 30),

‘<’: (26, 30),

‘>’: (27, 30),

0: (28, 30), # IDLE

‘=’: ( 0, 31),

‘[‘: ( 1, 31),

‘\\’: ( 2, 31),

‘]’: ( 3, 31),

‘^’: ( 4, 31),

‘\_’: ( 5, 31),

‘`’: ( 9, 31),

‘{‘: ( 6, 31),

‘|’: ( 7, 31),

‘}’: ( 8, 31),

‘`’: ( 9, 31),

‘\u00B1’: (10, 31), # plus/minus

‘\u00F7’: (11, 31), # division sign

‘\u00B0’: (12, 31), # degrees sign

‘\u00D7’: (13, 31), # multiply sign

‘\u00A3’: (14, 31), # pound sterling sign

‘\b’: (27, 31), # BS

‘\u007F’: (28, 31), # DEL

}

As with our other modes, we have an fsq\_config.py module to isolate details of the implementation from other parts of the program:

From fsq import FSQ

From time import sleep

From radio import Radio

Class FSQConfig:

Def \_\_init\_\_(self, frq, baud, call, location):

Self.frequency = frq

Self.baud = baud

Self.mycall = call

Self.location = location

Self.r = FSQ(self.send\_tone, self.baud, self.all\_done)

Self.dds = Radio()

Self.dds.send()

Self.is\_beacon = False

Self.message = ”

Self.spacing = 8.7890625 # 8.7890625 Hz

Self.usb\_offset = 1350.0

Self.is\_directed = False

Self.tocall = “N0CALL”

Self.beacon\_interval = 60.0

Self.incremental\_tone = 0.0

Self.r.set\_frequency(self.frequency)

Self.r.set\_call(self.mycall)

Self.r.set\_location(self.location)

Self.r.set\_call(self.mycall)

Self.r.set\_location(self.location)

Print(”Frequency:”, self.frequency)

Print(”Baud:”, self.baud)

Print(”Beacon?:”, self.is\_beacon)

Print(”Directed?:”, self.is\_directed)

Print(”To callsign:”, self.tocall)

Def get\_radio():

Return dss

Def set\_message(self, msg):

Self.message = msg.format(self.mycall, self.location)

Self.all\_done = False

If self.is\_beacon:

Self.r.set\_message(”\r\n\r\n{}:{}{}\r \b “.format(self.mycall, self.crc(self.mycall), self.message))

Else:

Self.r.set\_message(”{}:{}{}\r \b “.format(self.mycall, self.crc(self.mycall), self.message))

Def send\_code(self):

Self.dds.on()

Sleep(.1)

Self.r.send\_code()

Def send\_tone(self, tone):

Self.incremental\_tone = (self.incremental\_tone + float(tone) + 1.0) % 33

Self.f = int(int(self.frequency) + self.usb\_offset + self.incremental\_tone \* self.spacing)

Self.dds.set\_freq(0, self.f)

Self.dds.send()

Def all\_done(self):

If self.is\_beacon:

Self.r.stop() # stop sending bits

Self.dds.off()

Sleep(float(self.beacon\_interval))

Self.dds.on()

Self.r.send\_code() # Repeat for a beacon

Else:

Self.r.stop() # stop sending bits

Self.dds.off()

Self.all\_done = True

Def crc(self, text):

Self.table = []

For x in range(256):

Byte\_val = x

For y in range(8):

If byte\_val & 0x80:

Byte\_val = (byte\_val \* 2) ^ 7

Else:

Byte\_val = (byte\_val \* 2) ^ 0

Self.table.append(byte\_val & 0xFF)

Val = 0

For ch in text:

Val = self.table[val ^ ord(ch)] & 0xFF

Return “%0.2X” % (val & 0xFF)

The code basically handles setting up the frequency, baud rate, call, and location, and handles formatting the message. The first part of any FSQ transmission is the call sign and a 2-character Cyclic Redundancy Check to ensure that the call sign was properly received.

The fsq.py module is much simpler than the MFSK module was, since it has much less to do:

From machine import Timer

From fsq\_varicode import fsq\_varicode

Class FSQ:

Def \_\_init\_\_(self, baud, send\_tone, report\_message\_end=None):

Self.send\_tone = send\_tone

Self.report\_message\_end = report\_message\_end

Self.set\_baud(baud)

Self.frequency = “7104000”

Self.call = “N0CALL”

Self.location = “CM87xe”

Self.message = “{} {} “

Self.baud = baud

Self.bit\_length = int(1000 / float(baud))

Self.timer = Timer()

Self.all\_done = False

Def set\_call(self, call):

Self.call = call

Def set\_baud(self, baud):

Self.baud = baud

Self.bit\_length = int(1000 / float(self.baud))

Def set\_frequency(self, frequency):

Self.frequency = frequency

Def set\_location(self, location):

Self.location = location

Def set\_message(self, message):

Self.message = message.format(self.call, self.location)

Def bit(self):

For letter in self.message:

Code = fsq\_varicode.get(letter)

If not code:

Code = fsq\_varicode.get(” “) # Make illegal characters send as spaces

Count = 0

For tone in code:

If tone > 0 or count == 0:

Yield tone

Count += 1

Self.all\_done = True

Def stop(self):

Self.timer.deinit()

Self.all\_done = True

Def send\_code(self):

Self.all\_done = False

Self.gen = self.bit()

Self.timer.init(period=self.bit\_length, mode=Timer.PERIODIC, callback=self.bit\_finished)

Def send\_bit(self, unused):

Try:

Tone = next(self.gen)

Except StopIteration as tone:

Self.all\_done = True

Self.stop()

Self.report\_message\_end()

Return

Self.send\_tone(tone)

Def bit\_finished(self, unused):

Self.send\_bit(True)

We have a generator to give us each symbol (we call it a ‘bit’ as we did in the RTTY and MFSK modules, but it is actually a symbol since it is a tone that maps onto one of 32 characters).

We set up the timer to send the bits at the right rate, and the timer callback calls bit\_finished() which simply calls send\_bit().

The main.py module is even simpler:

From fsq\_config import FSQConfig

From time import sleep

# FLDIGI knows these baud rates: 1.5, 2, 3, 4.5, 6

Def main():

Fsq = FSQConfig(7040000, 12, “AB6NY”, “CM87xe”)

While True:

Fsq.set\_message(”{} Testing from {} using a Raspberry Pi Pico RP2040”)

Fsq.send\_code()

While fsq.all\_done == False:

Sleep(5)

Main()

It should be self-explanatory. The radio.py and SI5351.py modules are the same as the ones we used for MFSK.

Fsq Mode

Python Programming