WSJT-X Transmit with DDS

Proof of Concept -- First Rough Draft

WB2OSZ 10/31/2018 -- 11/22/2018

The goal of the Tech Night Radio project is to design and build a simple, low cost, yet high quality transceiver as a group effort. Ability to use the FT8 mode is a design goal. SSB is not.

Normally FT8, and the other WSJT-X modes, are sent by generating audio and feeding this into an SSB transmitter. The amplitude remains constant but the tone shifts among many different values. The SSB transmitter basically shifts this audio range up into the RF range.

Rather than sending audio through an SSB transmitter, we want to generate the radio frequency directly using Direct Digital Synthesis (DDS).

**This document describes a “proof of concept” to demonstrate that we can extract the necessary information, from WSJT-X, and instruct a Direct Digital Synthesis chip to generate an equivalent signal.**

The goal is to demonstrate this working:

Modified WSJT-X

Sending

Converter

Application

AD9834

Eval Board

WSJT-X

Receiving

------- Raspberry Pi -----------------

Windows PC

UDP over IP

SPI Port

Audio to Mic Input

The components are:

* A modified WSJT-X generates some sort of message at the beginning of each transmission.
* An application converts this message into commands for the AD9834 DDS chip.
* The DDS chip generates audio.
* Another instance of WSJT-X, on a different computer, decodes the audio.

This document addresses the proposed method of accomplishing this, along with a working proof-of-concept.

Tasks:

1. Figure out how to get required data from inside WSJT-X.
2. Somehow send this information to our application.
3. Beginning of a radio application to listen for these control messages.
4. Generate the tones with proper timing.
5. Talk to the DDS chip.
6. Test with WSJT-X on both ends.
7. Closing remarks on the time of day.
8. Next steps.

# What’s going on inside WSJT-X?

The modulator wasn’t hard to find. It is in Modulator.cpp. Let’s try to understand what the variables mean and look at their values for the different modes.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | FT8 | JT4 | JT9 | JT65 | QRA64 | ISCAT-A | MSK144 | WSPR |
| m\_frameRate | 48000 | 48000 | 48000 | 48000 | 48000 | 48000 | 48000 | 48000 |
| m\_period | 15 | 60 | 60 | 60 | 60 | 30 | 30 | 120 |
| m\_tuning | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| m\_toneFreqency0 | 1500 | 1500 | 1500 | 1500 | 1500 | 1500 | 1500 | 1500 |
| m\_symbolsLengh | 79 | 206 | 85 | 126 | 84 | 1291 | 144 | 162 |
| m\_nsps | 1920 | 2742.857 | 6912 | 4458.231 | 6912 | 557.28 | 6 | 8192 |
| m\_frequency | User (1500) | User  (1500) | User (1500) | User (1500) | User  (1500) | 1012.06 | 1000 | User ± |
| m\_toneSpacing | 6.25 | 4.375 | 1.73611 | 2.69165 | 1.73611 | 21.5332 | 1000 | 1.46484 |
| m\_bFastMode | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| m\_TRperiod | 15 | 60 | 60 | 60 | 60 | 30 | 30 | 120 |
| mstr | 397 | 397 | 397 | 397 | 397 | 397 | 397 | 397 |
| synchronize | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| m\_ic | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| m\_silentFrames | 4944 | 28944 | 28944 | 28944 | 28944 | 0 | 0 |  |
| txdelay | 103 | 603 | 603 | 603 | 603 | 0 | 0 |  |
| Distinct number of tones | 8 | 4 | 9 | 65 | 64 | 42 | 2 | >37? |
| Baud | 6.25 | 4.375 | 1.73611 | 2.69165 | 1.73611 | 21.5332 | 2000 | 1.46484 |
| Actual Time | 12.64 | 47.1 | 48.96 | 46.81 | 48.38 | **59.95** | **13.88** | 110.6 |

**m\_frameRate** – Audio sample rate. Always 48k/sec. Term “frame” probably comes from ALSA but I find it confusing here.

**m\_period** – Time slot for a transmission. FT8 is 15 seconds. Others generally 60 seconds starting at the new minute boundary.

**m\_tuning** – On for “Tune” button. I think this transmits one continuous tone. Needs more study. How do we know when to turn it off?

**m\_toneFrequency0** – Always starts as 1500, regardless of mode or user setting. Used during tone generation. Don’t understand yet. I don’t think we need this.

**m\_symbolsLength** – Number of symbol periods in the transmission.

**m\_nsps** – ¼ of the number of audio samples for one symbol period. E.g. 48000 / (1920 x 4) = 6.25 baud. This fudge factor of 4 shows up in a few places.

**m\_frequency** – Lowest audio tone in the group. Usually user specified. (Initial default of 1500.) For ISCAT it is 47 times the spacing. <http://www.g4jnt.com/ISCAT_Encoding.pdf>

**m\_toneSpacing** – Spacing between the tones used.

**m\_bFastMode** – True for the 30 second modes. Not for FT8 which is even faster.

**m\_TRperiod** – Seems redundant because it is the same as m\_period.

**mstr** – Current time as milliseconds after the beginning of the transmit time slot. Values shown are from a Raspberry Pi 3. Always around 400 mSec.

**synchronize** – Always 1 for all modes.

**m\_ic - Always 0 ???**

**m\_silentFrames** – Number of audio samples needed to bring us up to 500 mSec (FT8) or 1000 mSec (others) after the start of the transmit time slot.

**txdelay** – My addition. m\_silentFrames converted to milliseconds. It seems that the desire is to start the first tone 500 or 1000 mSec after the start of the transmit time slot. We always get here 397 mS after the boundary so transmit delay turns out to be 103 or 603, depending on the mode.

**baud - My addition.** Number of symbols per second. Interestingly, it usually turns out to be the same as the spacing between tones. MSK144 is the exception. <http://www.arrl.org/files/file/QEX_Next_Issue/SeptOct2017/FrankeTaylor.pdf>

I calculate this as

**¼ x m\_frameRate / m\_nsps**

**Actual time** of transmission - As a sanity check we expect it to be at least a couple seconds less than m\_period, to allow some slop, but more than 2/3 m\_period to use most of the time available. I calculate this as

m\_symbolsLength / baud

It is twice as long as expected for ISCAT so something doesn’t add up right. MSK144 is about half of what I was expecting. Both have m\_bFastMode set. Does this indicate they are exceptions to the general rule?

**Open issues:**

* How does CW id work?
* Does the “Tune” button send m\_freqency or something else?
* Details of last 3 modes.

# Send transmission parameters to another application

Thinking ahead….

WSJT-X already uses Hamlib to control the transceiver frequency and other functions. Conceptually, it would make sense to send the transmit parameters through the same mechanism. Hamlib <http://hamlib.sourceforge.net/manuals/hamlib.html> does have a capability to send a “raw command string” to the radio. This might be the way to implement the final version.

Anyhow for the proof-of-concept version, UDP seems like a good mechanism. Modulator.cpp has been modified like this:

* It looks for the environment variable TNR\_PORT. If that is defined a UDP packet will be sent to that port number at the beginning of each transmission.
* If TNR\_HOST is defined, that host name or address will be used instead of localhost.

After pondering various binary and text based formats, I came to the conclusion that JSON would make a lot of sense. It is commonly used, human readable, extensible, easily parsed, and has low overhead. Here is a typical message for FT8, shown as multiple lines for clarity.

{

"period":15,

"txdelay":103,

"freq0":1500,

"spacing":6.25,

"baud":6.25,

"symcount":79,

"tones":[2,5,6,0,4,1,3,4,0,1,3,0,0,1,6,1,3,0,2,5,0,4,5,3,2,0,2,7,4,0,

2,0,3,6,0,4,2,5,6,0,4,1,3,7,6,4,0,4,0,6,1,4,7,3,2,2,6,6,4,1,3,2,5,

4,1,5,0,0,1,7,6,4,2,5,6,0,4,1,3]

}

The “Tune” button, being turned on and off, would send messages like this.

{

"tune":1,

"freq0":1500

}

{

"tune":0

}

Finally we have the “Halt Tx” button which can interrupt a transmission already in progress. In this case we have:

{

"halttx":1

}

Some might be concerned about the overhead of sending a text based format, including value names, instead of tightly packing the binary numeric values. Fear not. Building up the message and sending the UDP packet takes less than 0.1 millisecond.

(Converting the numbers to this text format a million times took 68 seconds. I will prepare a more scientific and convincing experiment.)

# Interpret the transmission commands

A little application was written to listen for UDP messages, parse them, and interpret the contents.

Easy to use JSON parsers are available for all popular programming languages

First we wait for txdelay milliseconds to bring us up the desired ½ second or second boundary.

Each symbol lasts 1 / baud seconds. For JT8 it’s 160000 microseconds.

For each of the symbols (tone numbers) listed the audio frequency is:

freq0 + spacing \* tone\_number

When transmitting upper side band, we simply add the carrier frequency to this.

This is what the temporary debug messages look like:

{"period":15,"txdelay":103,"freq0":1500,"spacing":6.25,"baud":6.25,"symcount":79,"tones":[2,5,6,0,4,1,3,4,0,1,3,0,0,1,6,1,3,0,2,5,0,4,5,3,2,0,2,7,4,0,2,0,3,6,0,4,2,5,6,0,4,1,3,7,6,4,0,4,0,6,1,4,7,3,2,2,6,6,4,1,3,2,5,4,1,5,0,0,1,7,6,4,2,5,6,0,4,1,3]}

Debug: each symbol is 160000 microseconds.

tone start 0 f= 1512.5 dds= 5413

tone start 160000 f= 1531.25 dds= 5481

tone start 320000 f= 1537.5 dds= 5503

tone start 480000 f= 1500 dds= 5369

tone start 640000 f= 1525 dds= 5458

tone start 800000 f= 1506.25 dds= 5391

tone start 960000 f= 1518.75 dds= 5436

…

…

tone start 12000000 f= 1500 dds= 5369

tone start 12160000 f= 1525 dds= 5458

tone start 12320000 f= 1506.25 dds= 5391

tone start 12480000 f= 1518.75 dds= 5436

tone end 12640000

For each of the symbols, we have the time, in microseconds, relative to the start time. After that is the audio frequency and the code send to the DDS frequency control register.

Example: the first symbol is 2 so we have 1500 + 2 \* 6.25 = 1512.5.

My initial goal is to generate audio which can be fed into WSJT-X to verify that the signal is being generated correctly. To generate an SSB RF signal, simply add the in the carrier frequency.

You don’t need to be running WXJT-X, on the sending end, to test this part. You can manually give it a command string from the command line like this: (all one line)

echo '{"period":15,"txdelay":103,"freq0":1500,

"spacing":6.25,"baud":6.25,"symcount":79,"tones":[2,5,6,0,4,1,

3,4,0,1,3,0,0,1,6,1,3,0,2,5,0,4,5,3,2,0,2,7,4,0,2,0,3,6,0,4,2,

5,6,0,4,1,3,7,6,4,0,4,0,6,1,4,7,3,2,2,6,6,4,1,3,2,5,4,1,5,0,0,

1,7,6,4,2,5,6,0,4,1,3]}' | socat - udp-sendto:localhost:8888

# How do we get the timing right?

A question arose: Can we get accurate enough timing from the RPi or would we need a little microprocessor dedicated to setting the DDS frequencies at the proper time?

First, let’s describe a way which is not so good.

Start sending first tone.

Delay 160000 microseconds

Start sending second tone

Delay for 160000 microseconds

Start sending third tone.

Delay for 160000 microseconds.

etc.

When telling the operating system to delay (“sleep” function in many languages) there is no guarantee that it will be exactly that amount of time. Setting up the next tone also takes a finite amount of time. Any errors are cumulative so many small errors can add up to a substantial error over the entire transmission. Later symbols might not be exactly where expected and decoding quality will suffer.

Let’s compare this with a real life analogy. Suppose your plan for the day was to watch football and drink one beer each hour. You start your first beer at noon and it takes 5 minutes. You set the kitchen timer for one hour and watch football until the timer sounds. At 1:05 you start your next beer and this time it takes 6 minutes. Set the timer again. Next beer starts at 2:11 and takes 4 minutes. Set the timer for an hour. This time you hear the timer but there is something interesting happening so you don’t respond for 3 minutes. The next beer starts at 3:18. We are drifting further and further from the planned time.

Let’s try a different strategy. In this case we will schedule specific times for the beers. 12:00, 1:00, 2:00, 3:00, … In this case you continuously watch the clock on the wall. Is it time for the next beer? If no, check again. Repeat until it is the next scheduled time. The problem here is that your attention might not be focused 100% on watching for the next scheduled time. You could get distracted by the football game instead of constantly checking the clock. However, it you were a minute late for one of them, you could be back on schedule for the rest.

Let’s add a refinement. Suppose you had 4 independent brains. One brain could be 100% focused on constantly watching the clock and heading for the next beer the instant the schedule time arrived. Your other 3 brains could be watching football, eating nachos, talking with your friends and other things.

Now let’s bring this back into the computer realm. The Raspberry Pi models 2 & 3 have quad core processers. This means they can literally do 4 things at the same time, rather than one processor constantly jumping between different tasks competing for attention.

First we make note of the start time with microsecond resolution.

Start sending first tone.

Spin around until 160000 microseconds after the start time.

Start sending second tone.

Spin around until 320000 microseconds after the start time.

Start sending third tone.

Spin around until 480000 microseconds after the start time.

etc.

Success depends on two things:

* Other things going on do not need more than 100% of 3 CPU cores. Our time sensitive task should not get pushed out by other things going on.
* Ability to read the content of the real time clock with sufficient resolution.

Linux has a function to return the current time as two integers:

* Number of seconds since January 1, 1970 00:00:00 UTC.
* Number of microseconds since last second boundary.

Let’s try a little scientific experiment. See file timetest1.cpp.

This little program reads the system time clock over and over again as fast as possible. We find that we can read the clock over 6 million times per second. In a one second period we get about 999,000 unique values for the microsecond part. We know that the time of day function returns plenty of resolution.

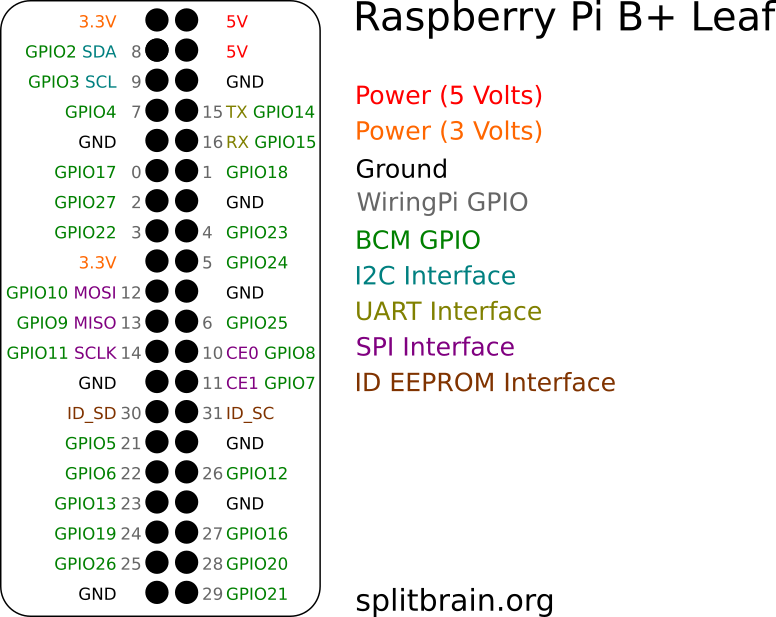
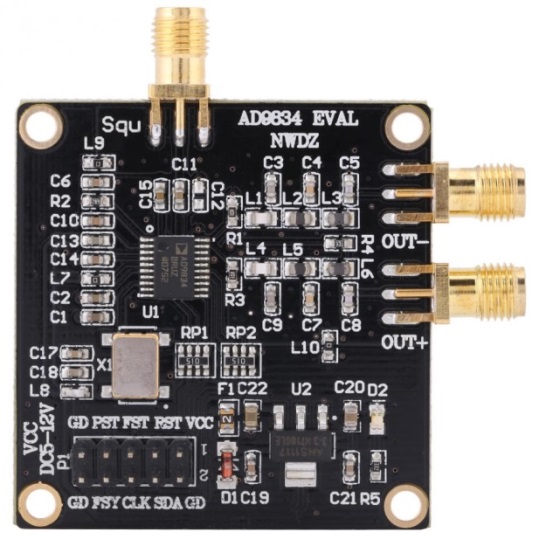
# Talk to the DDS chip

Next we need to command the DDS chip to produce the desired frequency using the SPI interface. The Raspberry Pi has an SPI port which can support two devices. They share common serial data and data clock. Each has its own chip select line. <https://www.raspberrypi.org/documentation/hardware/raspberrypi/spi/README.md>

Raspbian already has an SPI device driver built in, however the SPI port might not be enabled.

If devices **/dev/spidev0.0** and **/dev/spidev0.1** do not exist, run **raspi-config** either from the command line or from the gnome desktop. Enable the SPI port.

To test this, before the radio hardware is available, we will use an AD9834 evaluation board. These range from about $8.41 to $20. I was not able to find a supplier which would ship from within this country so be prepared to wait a few weeks.



From <https://github.com/splitbrain/rpibplusleaf>

Wiring is easy. There are a couple different variations available so you could encounter the pins in different locations. Jumper wires like this <https://www.adafruit.com/product/793> would be helpful.

|  |  |  |
| --- | --- | --- |
| **RPi** | **Purpose** | **AD9834 board** |
| Ground P1-20, 25 | Ground | GD 2, 9, 10 |
| 3.3 volts P1-17 | Power | VCC 1 |
| MOSI P1-19 | Serial data 🡪 | SDA 4 |
| MISO P1-21 | 🡨 Serial data | Not used. |
| SCLK P1-23 | Data clock 🡪 | CLK 6 |
| CE0 P1-24 | Chip select 🡪 | FSY 8 |
| CE1 P1-26 | (chip select for other) |  |

Connect a frequency counter to one of the OUT connectors.

AD9834.cpp contains code to send the necessary commands to the DDS chip. A little test program, at the end of the file, generates a series of different frequencies. Currently 1 kHz, 2 kHz, 3 kHz, 1 MHz, 2 MHz, and 3 MHz for ten seconds each. It’s subject to change so read the source code.

Compile and run it like this:

g++ -DTEST AD9834.cpp

./a.out

The frequency counter reading should be pretty close to the most recent number printed.

# End to end test

Now, let’s put all the pieces together.

Modified WSJT-X

Sending

Converter

Application

AD9834

Eval Board

WSJT-X

Receiving

------- Raspberry Pi -----------------

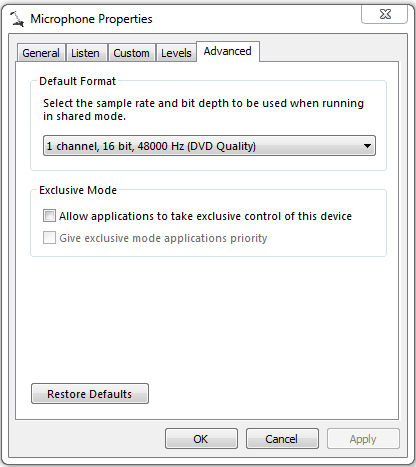
Windows PC

UDP over IP

SPI Port

Audio to Mic Input

On the PC, I happened to use a cheap USB Audio adapter, rather than the built-in audio on the motherboard. That shouldn’t make a difference. You might want to set the hardware sample rate to 48000 so the operating system doesn’t need to do a rate conversion.



(Note: The RC4 version caused me a lot of frustration. Signals would be rarely decoded. I finally saw in the discussion group <https://groups.yahoo.com/neo/groups/wsjtgroup/conversations/topics/33868> that others thought it did not work very well. I went back to RC3, on the PC, and results are now solid.)

On the raspberry Pi, define an environment variable and run our modified WSJT-X. This enables the feature to send a message to that UDP port.

setenv TNR\_PORT=8888

wsjtx

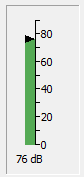
In another command window, compile and run the converter application. It’s currently called “dds” but I will probably change that.

sudo apt-get install libjsoncpp-dev

make

./dds

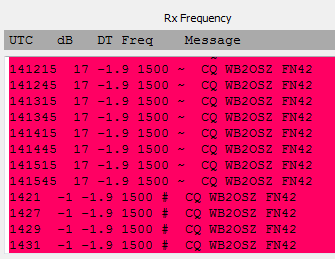
Make sure that both instances of WSJT-X are set to the same mode. On the Raspberry Pi instance, configure with your callsign and grid square. Click “Enable Tx” button.



Adjust the audio input level, on the second computer, so that we get a healthy audio level but don’t get into the yellow or red zone where clipping could occur.

In my experience, it is better to disable any automatic gain control (AGC) for the audio input device.

We are able to receive the audio from the AD9834.



|  |  |  |  |
| --- | --- | --- | --- |
| Mode | S/N dB | Mode character | Comment |
| FT8 | 17 | ~ |  |
| JT4 | \*\*\*\* | $ |  |
| JT9 | 17 | @ |  |
| JT65 | -1 | # |  |
| QRA64 | -16 to -13 | :\* |  |
| ISCAT-A | -6 | \* | Sending too long |
| MSK144 |  | & | Broken. Not surprising. |
| WSPR |  |  | Broken. |

I’m not sure what to make of the S/N dB value.

What does DT mean? “… the signal’s time offset in seconds relative to your computer clock.” Someone is off two seconds. In earlier versions of Raspbian, the network time protocol daemon (ntpd) was running continually. Recently I noticed that the time is set only on reboot and it could then drift. That is very unfortunate, especially for this application.

It turns out that the Raspbian “stretch” version is now using “timesyncd.” We discover that /etc/systemd/timesyncd.conf is not set up properly. Wow. How did that slip thru?

[Time]

#NTP=

#FallbackNTP=0.debian.pool.ntp.org 1.debian.pool.ntp.org 2.debian.pool.ntp.org 3.debian.pool.ntp.org

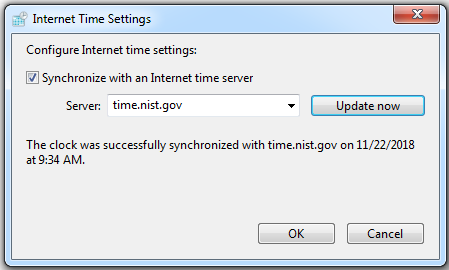
Fix it.

[Time]

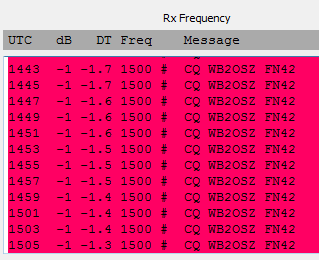
NTP=us.pool.ntp.org

FallbackNTP=0.debian.pool.ntp.org 1.debian.pool.ntp.org 2.debian.pool.ntp.org 3.debian.pool.ntp.org

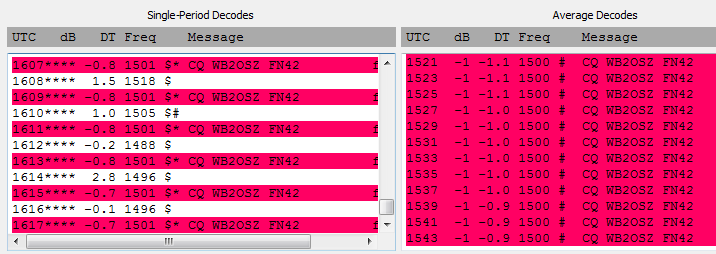
Sync windows time. No difference so that was OK.



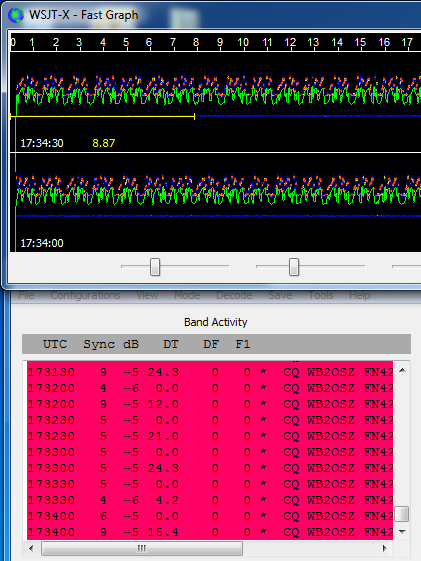
We now see the time being nudged closer. After a few more hours it reached -0.1.



Here is something odd. FT-8 and JR65 showed up in the pane on the right. But JT-4 shows up on the left. Why do we see “\*\*\*\*” in the dB column? Maybe I need to RTFM better. Note that time difference (DT) continues to decrease after fixing the timesyncd configuration. I would be real tempted to discard that and use NTP.



ISCAT-A is different than the others. First it sounds different. A new “Fast Graph” pops up. It looks like the same this is sent several times during the time slot. It should be a 30 second mode but I send it for a full minute. There must be some special case code in the tone generation that behaves differently.



Maybe I should just read the protocol specification and study Modulator.cpp in more detail.

Same for MSK144. And WSPR.

# Time of day is important

For proper operation, you want your time of day system clock to be within about a second of correct time. Transmissions are made within certain time slots, not at random times.

When connected to the Internet, this is not a problem. Your time of day clock gets set automatically. It is a problem when operating portable where you probably don’t have an Internet connection.

There are a couple possible solutions. One is to have a battery powered real time clock which maintains the time when the RPi is turned off. Here are 3 typical examples:

<https://learn.adafruit.com/adding-a-real-time-clock-to-raspberry-pi/wiring-the-rtc>

The cheap ones can drift a couple seconds a day. Completely unacceptable for this application. The most expensive one might be OK for a couple days off the Internet.

For the same price you can buy a GPS receiver and configure software to synchronize your system time to the GPS signal. Just as an example, I have one of these <https://www.amazon.com/Diymall-G-mouse-Glonass-Raspberry-Aviation/dp/B00NWEEWW8> It’s cheap and it works well. Here is one example of how to do it: <https://github.com/wb2osz/direwolf/raw/master/doc/Raspberry-Pi-APRS-Tracker.pdf>

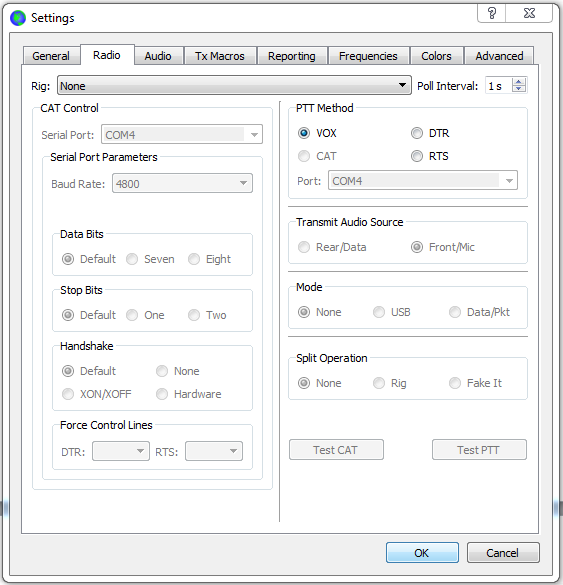
You could even use a mobile phone as a WiFi hotspot and that would provide Internet access to talk to an NTP server.

# Next Steps

We have demonstrated that we can extract the necessary information from WSJT-X and use it to generate equivalent audio using a DDS chip. The approach is feasible but there is still more work to be done.

* Get all of the modes to work properly. Some don’t seem to follow the pattern of the others and need to have some special case handling added.
* Figure out how CW id works and make sure this can be conveyed to our application.
* Verify what is sent for “Tune” button. It is a steady tone but is it the lowest tone of the group or something different?
* Implement the “Halt Tx” function properly. Modulator.cpp doesn’t seem to have enough information to distinguish this from normal end of transmission. I think we will need to make a slight modification to an additional file, before flow of control gets to Modulator.cpp, to get this information.
* Clean up the Modulator.cpp modifications and convince the WSJT-X maintainers to integrate something like this into official version so we don’t have to building our own custom version. As mentioned before, funneling this thru hamlib, using the raw command capability, makes a lot of sense conceptually. We would need only a single control channel between the application and the radio. Pick some existing rig type that uses TCP and emulate that. Add our own extension for the transmission parameters.

If that doesn’t work out, we could fall back on using a dedicated UDP port, as we have done here. Rather than using an environment variable, to activate the feature, it would probably make more sense to add it to the Radio Settings.



* Begin the actual radio “front panel” application. That will be the subject of other documents.

## Final thought, before I forget

When using a typical computer monitor, the WSJT-X main window is 930 x 588 pixels. When operating portable, you will probably have a smaller screen like this:

<https://www.adafruit.com/product/2718>

Question: What happens if you run WSJT-X on a screen with 800 x 480 pixels? Will it detect this situation and scale things down?