Remote Control Quisk Design

Version 0.2, 5/31/2022

Ben Cahill, AC2YD

Purpose/Goal/Status:

A) Provide a tightly-coupled full “Quisk Experience” remotely via network, with latency and timing fast and accurate enough for use with break-in CW using straight key, bug, or (untested) keyer, or other modes (e.g. SSB, digital), during day-to-day or contest operations.
B) Provide full front panel control support, e.g. filter settings, split mode, recording, etc.
C) Minimize data bandwidth requirements over the remote link. Data consists of control, graph/waterfall data, radio sound, and mic sound. I/Q data is not transported (see next goal).
D) Interface network traffic at high level, i.e. Quisk application level, not radio-specific level.
E) Leverage Quisk support for various OS platforms and sound systems.
F) Leverage Quisk support for various radios.
G) Leverage Quisk support for various methods of rig control, keying, etc., and not get in their way.
H) Keep installation, setup, and operation as simple as possible.

Major ham radio manufacturers are providing network-based remote control support for their gear, and remote operation is becoming more and more popular as time goes by. This design provides similar remote capability for Quisk. While there are a number of remote radio projects out there, AFAICT, none of these could provide the full “Quisk Experience” remotely, either because of control surface, latency, or both.

My personal goal for this project was local WiFi control of my outdoor “rig-in-a-box” from within my house, which works well at this point. However, feedback from Quisk users on groups.io in early 2021 indicated that this design likely has what it takes to additionally operate remotely over internet; it’s all network operation. I have not tested internet-based remote control.

As of this writing, the project’s goal has been achieved for CW and SSB operation controlled from the following types of Control Head platforms:

1) Linux desktop computer
2) Windows laptop computer

with a Raspberry Pi (3B+ or 4) serving as the Remote Radio computer, connected via WiFi to a dedicated Access Point, running the following types of radio hardware:

1) Softrock RXTX Ensemble
2) Hermes Lite 2

Operation feels comfortable and responsive, as if the radio hardware is directly attached to the Control Head, especially if the Control Head computer is attached to the WiFi Access Point via ethernet cable. WiFi connection of Control Head also works well; but the user should make sure the Control Head computer does not perform WiFi scans (more on that in “WiFi Considerations”, below).

All work was done based on quisk-4.1.80.

Note that this design does not attempt to do the following:

A) It does not touch I/Q data. It does not connect any particular radio to Quisk via network, e.g. it does not attempt to connect a Hermes Lite 2 to Quisk via WiFi instead of Ethernet.
B) It does not attempt to perform Quisk configuration nor radio calibration via remote control. Configuration/calibration can be done in-person at the Remote Radio computer, or via a remote desktop app (e.g. VNC).

Background/Inspiration/Now:

Background: My rig was in an outdoor shed, close to my “maple tree vertical” antenna, to which it attached via a 30’ coax cable. I originally had wanted to avoid running a long coax cable from inside the house, and the shed was sitting there unused … perfect! I set it up in Spring 2020, and it’s a fun place to operate in warm months, but it gets cold in the winter, and hot in the summer!

Inspiration: I wanted to stay warm in the winter, and cool in the summer! And be able to check the bands quickly and easily from indoors. Plus, I wanted to experiment with setting the rig up outdoors, right at the base of antenna, and controlling it remotely. This avoids running coax around the yard, and avoids cable losses due to impedance mismatches or long cable length. This also avoids lightning paths into the house, in-house grounding issues, holes in the house, trip hazards, attraction for rodents, etc.

In early experiments, remote desktop applications such as No Machine or VNC provided poor real-time response; they delayed mouse clicks, sound, and video by an uncomfortable amount of time, much more time than could be tolerated for CW using straight key or bug. I gave up, and started looking for ways to communicate more directly, and provide immediate CW sidetone.

Now: My headless Raspberry Pi / Softrock (now Hermes Lite 2) rig is in a weatherproof box at the base of the vertical antenna. Remote control allows me to stay warm and comfortable, and operate conveniently from my desktop or laptop computer inside the house (even if it’s not as much fun as the shed). CW Tx response is virtually instant; listening through an old superhet (no delay) receiver, the transmitted CW sounds simultaneous with the Quisk sidetone.

Even better, on a nice day, it is fun to use a laptop on the back deck or front porch to control the rig at the base of the antenna, 100 feet away.

Design Overview:

This design runs two separate instances of Quisk, one at each end, i.e., the “Control Head” end and the “Remote Radio” end. Running separate entire instances of Quisk works very well; it presents a full GUI at both ends, which has some very useful advantages in terms of ease of installation (it’s generic Quisk) and configuration (in-person or via remote desktop app), flexibility of operation (in-person or via Remote Quisk), and real-time monitoring of Remote operation from the Control Head computer (via remote desktop app such as VNC).

I considered splitting Quisk into separate front end and back end packages. However, Quisk is fairly small and efficient; splitting Quisk into separate front end and back end would provide little in terms of code size savings or computing efficiencies, and would require deep knowledge about knowing where to split it, always with the risk that some needed resource is in “the other” package. To split Quisk, separate installation packages (or at least options) would be needed for each end, both different from the generic full Quisk. I never saw a compelling reason to split it, and keeping it together has great advantages, as mentioned in prior paragraph.

To the operator, the design goal is that the Control Head end looks and sounds and feels as if Quisk is connected directly to a “real” radio. In addition to providing GUI and radio/mic sound to the operator, the Control Head end provides locally-generated (i.e., fast) sidetone for CW keying, and is designed to support use of any usual Quisk source for CW keying (e.g., serial port, MIDI, software). As of this writing, the only tested CW keying source is serial port.

The Remote Radio end does most of the signal processing, and directly controls the radio hardware connected to it, based on commands and mic sound from the Control Head. The Remote Radio sends radio sound, waterfall/graph data, and S-meter text to the Control Head.

Two new Python modules (files), quisk_hardware_control_head.py and quisk_hardware_remote_radio.py, provide the bulk of the remote-control functionality. These install into Quisk in the same way a Python module for a hardware radio, or e.g. quisk_hardware_hamlib.py, would be installed, i.e. by creating a new Radio in Quisk’s configuration menu, and using “Hardware File Path” to point to, e.g., quisk_hardware_control_head.py.

The Control Head Python module is self-sufficient; it serves as the entire “radio” for the Control Head end; no radio hardware is required to be attached to the Control Head computer, except a CW key or microphone, if you use one.

The Remote Radio Python module typically serves as a layer between Quisk and your usual radio hardware; you may need to edit an “import” line within quisk_hardware_remote_radio.py to invoke your usual radio module, e.g. “softrock.hardware_usb” for SoftRock radios.

Only a very modest amount of network address and port configuration is needed within quisk_hardware_control_head.py and quisk_hardware_remote_radio.py, to enable network connection between the two ends. These, of course, could be implemented as Quisk configuration menu options, but this is not done as of this writing.

Four channels of data flow between the Control Head and the Remote Radio. Each channel uses a separate network port, within a contiguous group of four ports:

1. Control, including CW keying (control head to remote), and Status, including S-meter (remote to control head)
2. Graph/Waterfall data (remote to control head)
3. Radio sound (stereo; remote to control head)
4. Microphone sound (mono/stereo?; control head to remote)

The control/status channel is TCP, but the others are UDP. Connection between the two ends is established via the control channel, for which the Remote Radio is a server (with a static IP address), and the Control Head is a client (and can be any address, e.g. DHCP). Either end can power up first; the Control Head keeps attempting to connect to the Remote Radio until it succeeds. You must configure quisk_hardware_control_head.py with the static IP address of the Remote Radio computer, and configure both quisk_hardware_control_head.py and quisk_hardware_remote_radio.py with a base port for the four contiguous data channels.

Radio Architectures:

I developed the remote Quisk software using my Softrock RXTX Ensemble, but it took very little (basically a bug fix regarding hardware CW keying vs software CW keying) to get a loaner Hermes Lite 2 to work well. This is very encouraging, since the architectures of the two radios are very different from one another. Support for other radios is TBD.

Quisk Resources:

Quisk has a vast pool of internal software resources, in both Python and C code, made available through several different hooks in the code. Quisk already has in place a number of calls to "Hardware" functions that may optionally be implemented within quisk_hardware_*.py modules, through which Quisk provides updates to values such as tuning frequency and operating mode, and frequent periodic opportunities to check values such as CW key state. And, through imports and variables passed in the call to __init__(), Quisk provides the quisk_hardware_*.py files with access to a broad range of functions and variables within Quisk.

Some of the bridges between Quisk internals and "Hardware" files such as quisk_hardware_*.py:

- Calls to "Hardware": Notifications from Quisk of state changes, e.g. ChangeFrequency(), or periodic calls, e.g. PollCwKey().
- import _quisk as QS: Access certain C functions within Quisk, using the table PyMethodDef QuiskMethods[] within quisk.c.
- __init__(self, app, conf):
    - self: This instance of a Hardware (radio) Python object, for static variables and functions created within quisk_hardware_*.py. self is separate from Quisk internals.
    - app: Quisk's "App" class, Python code within quisk.py, mostly having to do with GUI.
    - conf: Configuration variables within Quisk.

Through these rich resources/hooks, some modest remote-control functionality can be achieved without modifying the pre-existing Quisk code. However, to really provide the full "Quisk Experience", patches are required for GUI interaction (hooks for each control panel widget), CW key timing (a small new function in quisk.c), graph/waterfall data (minor changes to Python code) and handling sound (modestly extensive changes to C code in sound.c, including several new functions as well as hooks within pre-existing functions).

Control Channel:

In early experiments, I attempted to adapt the hamlib/rigctl protocol to use for Remote Quisk. However, this proved to be an awkward match for the front panel operational characteristics of Quisk. I gave up on the hamlib approach, and ended up creating a very simple protocol that essentially conveys all Quisk front panel user interactions (mouse clicks, buttons, sliders, etc.), including state and parameters, along with CW keying, from the Control Head to the Remote Radio.

Using Python calls to “Hardware” functions, Quisk notifies quisk_hardware_control_head.py immediately upon changes made by the operator. For each of these changes, the Control Head immediately sends a control message to the Remote Radio. These include:

- Frequency change (including band change): ChangeFrequency(self, tune, vfo, source, band). This goes beyond HamLib/RigCtl conventions, and sends all provided info to the Remote Radio in one message. VFO is especially important, or else (with hamlib protocol, which does not comprehend separate tune and VFO frequencies, which are vital for many SDR radios) the Remote Radio comes up with its own VFO value to support the desired tuning frequency. This can often mis-align the Remote Radio graph/waterfall data vs. the Control Head.
- Standard Quisk control panel buttons and sliders: Since version 0.1 of this document, I implemented support for all standard front panel control buttons and sliders, including split mode, adjustable filter bandwidth, modes, etc. Most of these required adding simple hooks, within the various support functions within quisk.py, to call new Hardware functions within quisk_hardware_control_head.py. NOTE: Front panel remote support does not include the calibrate button; calibration can be done via remote desktop software, e.g. VNC, or in person at the remote radio.
- Additional front panel functionality includes S-meter text data from the Remote Radio, displayed on the Control Head front panel.
- Radio-specific control panel widget support is TBD, e.g. LNA gain for Hermes Lite 2, etc.

The Remote Radio checks for new control commands quite frequently (e.g. 200 Hz, depending on settings), within Quisk’s call to PollCwKey(). See “CW Keying”.

Currently, the Remote Radio is assumed to track the Control Head; no attempt is made to verify or track the Remote Radio tuning frequency, mode, etc. at the Control Head end., although these sorts of status messages could be added in the future.

This remote-control design does not utilize nor override any pre-existing connectivity resources within Quisk (e.g. hamlib). This is so that the abundant connectivity provided by Quisk to work interactively with other software (e.g. N1MM+ logger, digital mode software, hamlib/rigctl, MIDI) is not compromised by the remote control operation. I use N1MM+ for contesting, but I have not tested any other interfaces.

CW Keying:

Quisk’s frequent calls to function “PollCwKey()” provides the basis for the most time-critical operation of the remote-control link. This call is implemented (differently, for different purposes) within both of the new quisk_hardware_*.py files, and occurs with a frequency established by Quisk's configuration parameter "HW Poll usec". I set this to 5000 (5 msec) in both Control Head and Remote Radio, so PollCWKey() gets called at a frequency of ~200 Hz at both ends. Note that no attempt is made to synchronize the timing of these calls between Control Head and Remote Radio; the high 200 Hz frequency works well enough, along with CW timestamps (see below), to keep things operating smoothly.

The Control Head uses PollCWKey() to check CW Key state, using a new function within Quisk, is_cwkey_down(). When the CW key state changes, the Control Head sends a CW Key control message, including up/down state and a timestamp, to the Remote Radio via the network.

The Remote Radio uses PollCWKey() to read any/all control messages from the network, including CW Key, as well as changes in other parameters, e.g. tuning frequency. If the control message is for operating parameters such as tuning frequency or mode, these values get updated in the Remote Radio Quisk immediately.

If the control message is for CW keying, the Remote Radio buffers the keying command, including timestamp, in a queue for a configurable short delay, e.g 20 msec. This buffer time serves to smooth over nominal jitter in the network link, as well as the lack of direct timing synchronization between Control Head and Remote Radio ends. The Remote Radio "plays" the buffered CW commands based on the timestamps captured by the Control Head, rather than network message arrival time at the Remote Radio computer.

In version 0.1 of this document, I reported some sluggishness in Tx enable within the Remote end. This has been fixed, and response is now quite good using the pre-existing resources built into Quisk. The sluggishness, as well as some other instability, was caused by me directly calling control panel related functions in another thread (the GUI "app" thread), instead of using the proper call wrapper wx.CallAfter(). Using wx.CallAfter() "magically" fixed weird behavior in Tx enable and elsewhere.

Sound:

Quisk has solved many problems regarding sound, including installation, connection, and latency in different sound systems in different operating systems. These are worth leveraging! The hooks for sound in Remote Quisk are in the heart of the Quisk sound system, independent of OS or flavor of platform sound system.

Transport of Radio sound and Microphone sound between Control Head and Remote Radio are implemented by new code within sound.c. For radio sound, the Remote Radio is the UDP server, and the Control Head is the UDP client. For mic sound, the reverse is true. The server is implemented in new function send_remote_sound_socket(), and the client is implemented in new function read_remote_sound_socket(). These get called selectively within quisk_read_sound() (the main function called from Quisk's high level sound loop), to intercept-and-send sound, or receive-and-inject sound.

New setup/teardown functions within sound.c get called by quisk_hardware_*.py when the TCP control connection is established or disconnected. The setup functions are quisk_start_control_head_remote_sound(), which calls open_remote_sound_client() for radio sound, and open_remote_sound_server() for mic sound. Similarly, quisk_start_remote_radio_remote_sound() calls open_remote_sound_server() for radio sound, and open_remote_sound_client() for mic sound.

Similar symmetric teardown functions are quisk_stop_control_head_remote_sound() and quisk_stop_remote_radio_remote_sound().

Once network communications are set up, the Remote Radio intercepts radio sound just after it is created by quisk_process_samples(), which processes RX-IQ samples to create the radio sound. These radio sound samples get sent immediately via UDP to the Control Head, which places them in a (typically) short circular buffer, and injects them at the same point within sound.c, instead of performing quisk_process_samples(). From here, radio sound can be played through any sound device supported by Quisk. Playback volume and latency are controlled at the Control Head end.

I tried to avoid the short circular buffer mentioned in the prior paragraph, but I could not get radio sound to work well with ALSA on my development system (Linux). The circular buffer helps to manage network latencies, by inserting 0 values if sample reception is not timely and the circular buffer becomes empty, or by throwing away samples (during Tx) if too many 0 values have accumulated, and overdue samples have since arrived, causing long latency within the buffer. Note that, in addition to other possible reasons, this scenario could occur if a WiFi scan happens; see "WiFi Considerations".

Radio sound sample timing on the Control Head computer is implemented by a floating point version of QuiskDeltaMsec(), surrounded by floating point math and 0.5-sample rounding, to avoid over- and under-runs of sound samples relative to the sound rendering device sample rate.

In version 0.1 of this document, Microphone sound was not yet implemented, but it is now, and I have used it successfully for SSB.

The Control Head generates a CW sidetone locally, for "instant" response. Pre-existing Quisk code for ALSA and Windows Fast Sound (WASAPI) have their own system-specific sidetone generators that minimize latency by generating the sidetone just before radio sound samples go to the sound device. Other sound systems rely on a higher level sidetone generator within Quisk, which is not in-line with the remote Radio Sound path. So, CW sidetone is generated within read_remote_sound_socket() when necessary, using Quisk's quisk_make_sidetone() function.

The placement of the sound interception and injection hooks should support Quisk sound recording/playback as well, but I have not tested this. I/Q sample recording and playback would need to happen on the Remote Radio end, and is not currently implemented for Remote Quisk.

Sound is currently transferred at 48KHz sample rate, and this works well in my setup over WiFi, but it may be possible/desirable to reduce this to 8 KHz as a network bandwidth optimization.

Graph/Waterfall:

Graph/waterfall data requires a small patch within "OnReadSound()" in quisk.py to hook into the flow of graph data, which is also used to generate the waterfall display. If quisk_hardware_control_head.py or quisk_hardware_remote_radio.py is installed, the patch calls function "Hardware.GetGraph()", which is implemented differently in the two ends of quisk_hardware_*.py.

In the Remote Radio, GetGraph() calls QS.get_graph() (in quisk.c, accessible via "import _quisk as QS"), to get the graph data, returns it to the normal flow (so graph/waterfall is visible on the Remote Radio computer), and also converts it, using the "numpy" Python library, from a Python tuple of floating point values into a block of single-byte integer values (to save network bandwidth) and sends it immediately to the Control Head via UDP. If there is objection to using numpy for any reason, my experience with the 48K-sample audio working well suggests that this bandwidth optimization might not be necessary. Actually, there are currently some visual artifacts for low signal levels when using the single-byte values, so the original floating point values may be better after all.

In the Control Head, GetGraph(), receives graph data from the network, converts it back to a tuple of floating point values, and returns it to the caller to be used for the Control Head graph and waterfall.

Snap-to-signal Rx tuning for CW works in the Control Head by virtue of graph/waterfall data received from the Remote Radio.

TODO List (shrinking!):

As of this writing, a large number of items on the “To Do” list of this document version 0.1 have been COMPLETED. These include:

- Non-CW modes, e.g., SSB
- PTT
- Split Frequency
- S-meter
- Spot and Spot Level (and make sure it works locally at remote radio computer for “on-site” calibration!)
- Radio sound audio filter variable bandwidth button
- CW tone frequency
- RIT
- Display remote connection status (connected/trying/whatever)
- More reliable support within Quisk for Tx-enable timing
- Build/test for Windows (high priority for my back deck on nice days!)
- Support radios other than Softrock (Hermes Lite 2 works!)

The items on the 0.1 “To Do” list that remain as “To Do” include:

- 8 KHz audio bandwidth over remote connection (reduce WiFi bandwidth)
- Build/test for Mac (I don’t have one)

WiFi Considerations and Experiences:

Wifi Scans:

Computers at either end of the remote connection must *not* perform periodic “background” WiFi scans! WiFi scans look for available WiFi network Access Points (APs), and in doing so, tune away from the WiFi channel used for remote Quisk. Tuning off-channel, even briefly, interferes severely with timely data exchange! Using ping (see Tools), I found round trips between my desktop computer (Control Head) and Raspberry Pi (Remote Radio) were typically 1.5-2 msec (nice!), but got as bad as 100+ msec due to scans; this is catastrophic for real-time CW operation.

Scans are necessary when a device is looking for an AP to which to attach, i.e. before making a connection, or after losing a connection. These are good. “Background” scans happen after connection has successfully occurred, to keep looking for a possibly better connection. These are bad.

By default, most computers/devices are set up to roam, which requires background scanning. For example, my Raspberry Pi 3 B+ had not just one, but three independent sources for triggering WiFi scans as frequently as once per minute. These sources include the WiFi control applet, the wpa_supplicant layer, and the WiFi device driver. You can find my post of April 3, 2021 on the Raspberry Pi forum on how to disable all three:

https://www.raspberrypi.org/forums/viewtopic.php?t=175480

Since a few months after writing version 0.1 of this document, I have used a Raspberry Pi 4 for a borrowed Hermes Lite 2. This newer Raspberry Pi did not require 3 levels of scan intervention. All that was required was to modify the file ~/.config/lxpanel/LXDE-pi/panels/panel, as described in the link above. I don’t know if this is because of a different OS version, different software stack, different hardware, or ?? from Pi to Pi. My OS info is below, found by invoking “uname -a”:

My Pi 3B+: Linux raspberrypi 5.10.17-v7+ #1403 SMP Mon Feb 22 11:29:51 GMT

My Pi 4: Linux raspberrypi 5.10.60-v7l+ #1449 SMP Wed Aug 25 11:29:51 GMT

NOTE: If you edit and save preferences on the panel applet (right-click on the panel bar), it may re-write the “panel” file mentioned above, and erase the line you added!

Windows computers may be helped by the use of “WLAN Optimizer”, a simple app that requests Windows not to scan. It may also be possible to perform the same request from within Quisk, using a Windows API call, but I have not attempted it.

Bluetooth uses 2.4GHz band, and often shares the same radio hardware used for WiFi, so it likely tunes away from the WiFi channel when Bluetooth is active. It might be a good idea to disable Bluetooth on any device used for Remote Quisk via WiFi. I have not tested this.

My WiFi Setup:

Disclaimer: I am not a WiFi expert. A lot of what follows is the result of trial and error. I’m writing it in hopes that it might help someone to find a better way. The following discussion includes setups that did not work great, along with those that worked well, and what I did to make things better. The setup is still somewhat work-in-progress, although I am getting good results at this point. Sorry, all words (too many) and no diagrams!

The setup I use for remote Quisk includes a dedicated access point (Linksys WRT600N router) that communicates with the Remote Radio computer (Raspberry Pi, in my case). This AP creates a WiFi network, named “AC2YD-QTH”, on channel 149 in the 5 GHz wireless band. This provides an isolated WiFi channel with no interference from household or neighborhood WiFi traffic (AFAIK), since most, if not all, of that local traffic is on the 2.4 GHz band. NOTE: A good Windows-based tool for assessing the presence of local APs is “WiFi Analyzer”, available free from Microsoft store. See “Tools” section of this document.

Channel 149 is at 5.745 GHz, within the 5.65-5.925 GHz amateur radio band, which may provide good possibilities for increasing range of the WiFi connection, should it be necessary (see Signal Strength, below).

Inspiration for using this particular part of 5 GHz band, and for calling the network “AC2YD-QTH”, came from the ARRL book “High Speed Multimedia for Amateur Radio”. As a radio amateur, I will legally be able to use more EIRP on this (and some neighboring) channels, should I need to improve the wireless connection, as long as I keep identifying the network as “AC2YD” via the broadcast SSID (see configuration below), and don’t encrypt the content (that’s the challenging part, while connected to my household network; see below!).

So that I can access the internet from AC2YD-QTH (e.g. for Quisk upgrades on the Raspberry Pi out in the shed, or for on-line lookups during operation from shed, office, or back deck), I connected the AC2YD-QTH router to my household router via LAN (not WAN) jacks on *both* routers. This makes the WRT600N behave as a switch on my local household network, instead of a router connecting to the outside world; it simply extends the household network to be able to reach the dedicated AC2YD-QTH WiFi network, and vice-versa.

My desktop computer is connected to the household 2.4 GHz router (not the 5GHz AC2YD-QTH router) via ethernet. In spite of this extra level of indirection between desktop and Remote Radio, all connections between my desktop and the AC2YD-QTH router are hard-wired ethernet, and the only traffic through the AC2YD-QTH router is (I think!) Remote Quisk traffic, so pings between my desktop and the Raspberry Pi are typically < 2 msec (nice!).

If I use a Windows laptop as the Control Head in my office, or the back deck, I typically connect it via ethernet cable (dropped out the office window for the back deck usage) to the household 2.4GHz router. To be even more sure, I usually disconnect the WiFi connection to our household router, and uncheck “connect automatically”.

If using the laptop beyond convenient range of an ethernet cable, I enable WiFi, and connect to the household 2.4-GHz router, as it normally is for regular internet usage. I also use “WLAN Optimizer” to attempt to disable WiFi scanning. As with the desktop, the extra level of indirection seems not to be a problem, and I discovered that the outdoor range of the 2.4-GHz router is somewhat greater than the 5-GHz AC2YD-QTH router, better for working from out on a hammock in the woods (haven’t done that yet)!

Particulars for setting up the dedicated AC2YD-QTH router (actually, switch). Your particulars may need to be different; this is just an example:

- Internet Setup:

    - Automatic Configuration – DHCP (but this really doesn't matter … this is the default setting for use as a router, connecting to outside world internet service provider, using the WAN port, which I do not use)
- Network Setup:
    - Router Address: 192.168.1.40 (my random choice, so I can directly access AC2YD-QTH router configuration menus … if left at default 192.168.1.1, it would conflict with the household router)
    - Subnet mask: 255.255.255.0 (default)
    - DHCP Server: Disabled (Would conflict with household router's DHCP. Raspberry Pi does not need DHCP because it has static IP address, and having DHCP enabled messed with my desktop computer's address when I had occasion to plug it, temporarily, directly into the AC2YD-QTH router)
- 5GHz Wireless Settings:
    - Network Mode: Wireless A Only (i.e., standard channel width of 20 MHz; no need for higher bandwidth, but higher bandwidth N mode or mixed probably would be fine, as long as you set things up properly)
    - Network Name (SSID): AC2YD-QTH
    - Standard Channel: 149 – 5.745 GHz (I'm using only Wireless A, standard channel width of 20 MHz)
    - SSID Broadcast: Enabled (this periodically sends "AC2YD-QTH" over the air!)
- 2.4 GHz Wireless Settings:
    - Network Mode: Disabled (using 5 GHz band only)
- 5 GHz Wireless Security (IIUC, encryption is illegal if I boost EIRP and operate under ham rules):
    - WPA-Personal (but you can set this to your preference)
    - Encryption: TKIP (again, set to your preference)
    - Passphrase

Signal Strength:

My experience with Raspberry Pi 4 WiFi suggests that -65 dBm (measured by running iwconfig on the Pi; see Tools) is a good minimum level to strive for. Once the signal gets to -70 dBm, it can still work well, but may have occasional radio sound crackles from audio dropouts. At -73 dBm, the connection starts to get a bit "iffy", and may even (so it seems) trigger a scan if/when the Pi thinks it might be disconnected. Ping (see Tools) is your friend for detecting bad behavior.

Some learning/experimentation history: In my initial trials, my Raspberry Pi 3B+ in the shed seemed to "barely make it" to the Linksys WRT600N 5GHz Access Point in my office, a path of about 90 feet, passing through 2 or 3 walls and 1 or 2 cathedral ceiling roof surfaces, depending on AP position (height) within my office.

This connection path required careful positioning of AP and Pi, and even some makeshift reflectors placed behind the Pi as well as the AP. If everything was not right, it failed to connect. However, once aligned, and once connected, it worked well, and seemed to hold onto the connection (although I noticed rain may make a difference). Typical signal strength for this connection path was about -80dBm, measured at the Raspberry Pi using iwconfig (see Tools). It seemed to work at that level, but reconnection could be "iffy", and rain seemed to make a difference in reliability.

I re-positioned the 5 GHz AP to our attic, which is above the cathedral ceiling path. I ran an ethernet cable through a wall to my office, to connect the 5 GHz AP to our household AP. I placed the 5 GHz

AP just inside the unfinished attic wall closest to the shed and maple tree. IIRC, comparing AP placement in an open window vs. the wall position, the wall attenuated the 5 GHz signal on the order of 3-6 dB. IIRC, attenuation through the closed window was about the same as the wall. I believe that the 5 GHz signal was probably attenuated about twice as much as a 2.4 GHz signal would have been, although I don't recall experimenting with this aspect.

With this AP position, there was a noticeable improvement in signal strength. I also changed the position of the Raspberry Pi from a table top in the shed, to placement just inside a closed window, turned sideways so the top of the Raspberry Pi case faced toward (but not directly toward) the AP in the attic (see below). With this configuration, -65 to -70 dBm is typical signal strength, again measured by the Raspberry Pi running iwconfig. This is at least 10 dB better than the "iffy" setup, and works more reliably, even for reconnection, and even in the rain. I used this setup successfully with the borrowed Hermes Lite 2, and it has behaved well in multiple contests, using CW and SSB.

Small changes in Raspberry Pi orientation can make a significant difference in signal strength. For positioning (pointing) the Raspberry Pi, you can find directional patterns on-line, e.g.:

https://antennatestlab.com/antenna-examples/raspberry-pi-model-3b-antenna-evaluation-gain-pattern

… and I think I saw an even better one out there, somewhere. I've been placing the Pi sideways, so the top surface of the circuit board points toward the AP, but not directly towards it. The highest gain lobe for 5.7 GHz seems to be off to the side a bit … hard to describe in words … see the animated directional gain video on the site above.

I also have a successful setup with my rig in a weatherproof box at the base of my "Maple Tree Vertical". For the Raspberry Pi 3B+ in that box, I attached a homebrew horn antenna for 5.76 GHz made from a template in the ARRL Antenna Handbook created by Paul Wade K1GHZ (N1BWT when the template was made). This worked amazingly well, in spite of some sloppiness in my construction of the horn. As with the Raspberry Pi built-in antenna, the best gain was a bit off-axis. The horn sits nicely on the floor of the box, with a little upward tilt (provided by the horn itself) to point to the AP in the attic, about 90 feet away. Today, the weather is beautiful (no rain), and I'm seeing -58dBm signal strength showing on iwconfig. With over 100 times (over 20 dB) the signal power of my initial setup, this rig-in-a-box setup is rock solid, and has served me well.

In looking into the horn antennas, I also ran across slotted waveguide antennas, which provide omnidirectional (azimuth) gain by focusing the signal vertically into a "pancake" pattern. Such gain might be good for the AP; ultimately, I'd like to support remote radios at antennas scattered around our property.

Paul Wade, W1GHZ, has some really good microwave antenna information at:

http://www.w1ghz.org/antbook/contents.htm

Also see the ARRL books "High Speed Multimedia for Amateur Radio" and "Antenna Handbook".

Platform Considerations:

Raspberry Pi Considerations:

See sections “WiFi Considerations”, and “Tools”.

I have used Raspberry Pi 3B+ or 4 exclusively at the Remote Radio end since beginning this project. I never tried a Pi at the Control Head end. They run Quisk well, and are happy with monitor, keyboard, and mouse, or headless … no need to change any configuration settings, IIRC, to change from one to the other, just plug in or unplug the peripherals. The Pi4 in the shed has keyboard/mouse/monitor. The Pi3B+ in the box is headless. VNC works great either way as a remote control interface for the Raspberry Pi desktop (although it is just a bit sluggish). See VNC description in Tools.

Update: My Pi4 always had a monitor connected. However, when recently trying to run the Pi4 headless via VNC, I discovered that VNC on my Laptop said “Cannot currently display the desktop” when trying to connect to the Pi4. Internet research showed that, before going headless, I needed to set a particular display resolution on the Pi4 via sudo raspi-config. However, even after setting it to 1920x1080 (the monitor resolution), VNC on the Windows Laptop is displaying a desktop (good), but at a lower resolution (bad). TBD!

In version 0.1 of this document, I reported that trying to use the Raspberry Pi 3B+’s built-in sound output jack on the Remote Radio computer slowed down Quisk operation considerably, to the point of not keeping up with the number of sound samples per second needed for a 48 KHz sample rate! Since then, however, I have successfully used the Raspberry Pi 4’s sound output jack, and I will guess that my problems with the 3 B+ were probably due to instability from my bad practice of calling functions in the “app” thread directly. This threading bug has since been fixed by using wx.CallAfter() wrapper for such calls. I have not tried the sound jack on the 3B+ since fixing the threading bug, because the 3B+ is now in the outdoor box, and I have no need for the jack.

Windows Considerations:

See sections “WiFi Considerations”, and “Tools”.

I have not tried using Windows at the Remote Radio end. Doing so may require a tweak or two to the network programming for Remote Quisk; I needed to tweak the Control Head side slightly to adapt to Windows from Linux.

At the Control Head end, however, I use a Windows laptop almost exclusively (except for software development, which I do on Linux desktop), running Windows-based logging software (N1MM+, ACLog, SKCC Logger, LOTW) conveniently on the Windows laptop, along with Quisk. I use a USB serial port adapter for key input from straight key and bug (my thanks to Jim Ahlstrom for wiring diagrams!). I do not (yet) use electronic keying; I use the loggers only as loggers, but they are able to nicely track the Quisk tuning frequency via serial port connection through the com0com software serial port emulator for Windows.

Normally, I have the Windows laptop connected directly to my household AP via ethernet cable. I haven’t spent much time trying to use the laptop via WiFi connection … it works pretty well, but I haven’t done extensive testing as to how well I can get Windows to avoid scans. There is a small software app named “WLAN Optimizer” that claims to turn off scans; it seems to help, at least. It may also be possible to add code to Quisk to turn off scans, using the same call used by WLAN Optimizer.

Just recently, to use for the Field Day contest (away from my home AP), I connected my Windows laptop directly to the 5 GHz AP via ethernet. This required setting up the Windows ethernet connection for:

- static IP address (instead of DHCP automatic address assignment)
- disabled firewall

I discovered that I could also simultaneously connect to my home AP via WiFi, using the current unchanged Windows setup for that connection, i.e. DHCP and the normal Windows firewall setup for public internet access.

In an effort to reduce network traffic (especially for WiFi), I disabled the following from being started at boot time:

- Cortana
- HP Message Service
- Microsoft OneDrive
- Skype
- Spotify
- Webex

This is accessed via Ctl-Alt-Delete -> Task Manager -> Startup

Tools:

ping:

Command line tool for checking round-trip time of network packets. Consistently low round-trip time, e.g. 2-5 msec, is vital for good CW performance. Watch for occasional bumps in round trip time; this may indicate WiFi scanning (bad!) by one end or the other, especially if the bumps occur truly periodically (e.g., once per minute). Linux ping defaults to continuous operation, making round-trip requests once per second, e.g.:

```
ping 192.168.1.60  (assuming Remote Radio is at that address)
```

For Windows, you can run ping from within a Command Prompt window (cmd.exe). By default, Windows ping sends out only 4 requests, so you may want to use the -t option for continuous requests, e.g.:

```
ping -t 192.168.1.60
```

To narrow down the source of scans or other troubles, try running ping between a given device and the AP to which it is connected, rather than over the entire Control Head to Remote Radio link.

iwconfig:

Command line tool for checking WiFi network connection quality on Linux systems. Shows signal strength (in dBm!) and current transfer rate. I like to run it on the Remote Radio (Raspberry Pi) from my Control Head computer via VNC (see below). Remote Quisk may be able to "work" even with signal strength of -80 dBm, but I recommend -65 dBm as a good level for solid response.

For Windows, a near equivalent is the following command. Unfortunately, it shows signal strength as a percentage, rather than in dBm. Run this from within a Command Prompt window:

```
netsh wlan show interfaces
```

WLAN Optimizer:
Windows tool that claims to turn off WiFi scans. I'm not sure that it eliminates scans entirely, but it seems to help, at least! I discovered that it's probably best to use this app on an "as needed" basis, i.e. only when you are using Quisk, so it does not interfere with initial connection or roaming (don't try to roam if you are using Remote Quisk via WiFi!). You can show/hide this app via a little icon in the Windows desktop tray. Freeware available from online download.

WiFi Analyzer:
Windows tool for assessing your neighborhood's "constellation" (my term) of WiFi Access Points (APs), including signal strength. I used it on my laptop, walking around our property to see how my APs would behave, and whether I had any neighbors using my Remote Radio WiFi channel. One interesting result was that my 5GHz AP seems to have only 1/2 - 2/3 the range of my 2.4GHz AP. This tool is available free from the Microsoft Store.

Wireshark:
Sophisticated development tool for analyzing network packets sent and received by your computer. For development, I run this typically on Linux desktop Control Head end, with capture filter "host 192.168.1.60" to capture all traffic, in both directions, between desktop Control Head (DHCP) and the Remote Radio (static IP 192.168.1.60). This is for development only, not needed for normal operation of Remote Quisk.

VNC:
Remote access software for controlling and viewing one computer from another. VNC Server comes installed by default in Raspberry Pi operating system. I use VNC on my Linux desktop or Windows laptop Control Head computer to see and interact with the desktop screen of the Raspberry Pi Remote Radio computer. This view is very useful in order to launch and configure the Remote Radio Quisk, verify that Quisk is running properly and is tracking the Control Head, run iwconfig, edit Raspberry Pi configuration files, etc. If your WiFi connection is "iffy", or your Raspberry Pi is sluggish, you can close the VNC connection to save compute cycles and WiFi bandwidth, and you can easily restart the connection if something goes wrong and you need to re-start Quisk on the Remote Radio end.

For Windows, I set VNC's File->Preferences->Expert->SendSpecialKeys to FALSE, to allow me to switch desktop views within Windows using Ctl-Windows-Left/Right key combinations.

Multiple Desktop Views:
I have 4 desktop views set up on my Linux desktop computer, and 5 on my Windows laptop. So, I can run Quisk in one, run VNC in another, do development in another, etc. I have only one monitor.