Martin Ewing, AA6E

28 Wood Rd, Branford, CT 06520; **aa6e@arrl.net**

A Software-Based Remote Receiver Solution

Need to get your receiver off site to avoid interference? Here is how one amateur connected a remote radio to a club station using a mixture of Linux, Windows, *and* Python.

The local interference environment is an increasing problem for ham radio operation, making weak signal operation impossible at some times and frequencies. The ARRL Staff Club Station, W1HQ, has this problem in spades when W1AW, just across the parking lot, is transmitting bulletins and code practice on seven bands with high power. Operating W1HQ on HF in the prime evening hours is not possible. There is no problem transmitting in this environment — it's a receiver problem. We could solve the problem by setting up a full remote base station, but for W1HQ, this is not necessary. A remote receiver is all that we need. That is the origin of this project.

There was only a small budget for a remote operation, but we did have a number of usable receivers, and we had an offer of software support from an eager volunteer. (That would be me.) So we were off and running to develop a low-cost remote receiving capability.

The club wanted a receiver capable of operating at least CW and SSB on all HF bands, controllable from the W1HQ operating desk. It needed to be located far enough from W1AW so that interference was negligible, but close enough so that propagation would be nearly the same at both locations. The system that evolved was a small remote computer that could be deployed at a ham's home (the "host QTH") and that would require little if any local support. The remote would require an all-band antenna of some kind, while it would attach to the host's home Internet connection. At the base station (W1HQ), the station computer would run control software to manage the remote operation.

This article focuses mostly on the software — how we developed a mixed *C* and *Python* (and mixed *Linux* and *Windows*) project with audio and Internet aspects. Interested readers will probably want to consult the code listings. These are provided at **www.aa6e.net/wiki/rrx_code** and are also available for download from the ARRL *QEX* files website.[1] Additional project details are available at **www.aa6e.net/wiki/W1HQ_remote**. Hardware construction is straightforward for anyone with moderate experience.

[1]Notes appear on page 6.

System Overview

Figure 1 shows the overall system design. The BeagleBoard XM or "BBXM" is an inexpensive (~$150) single board computer with a 1 GHz ARM processor, Ethernet and USB connections, and on-board audio capability.[2] It has 512 MB of RAM and

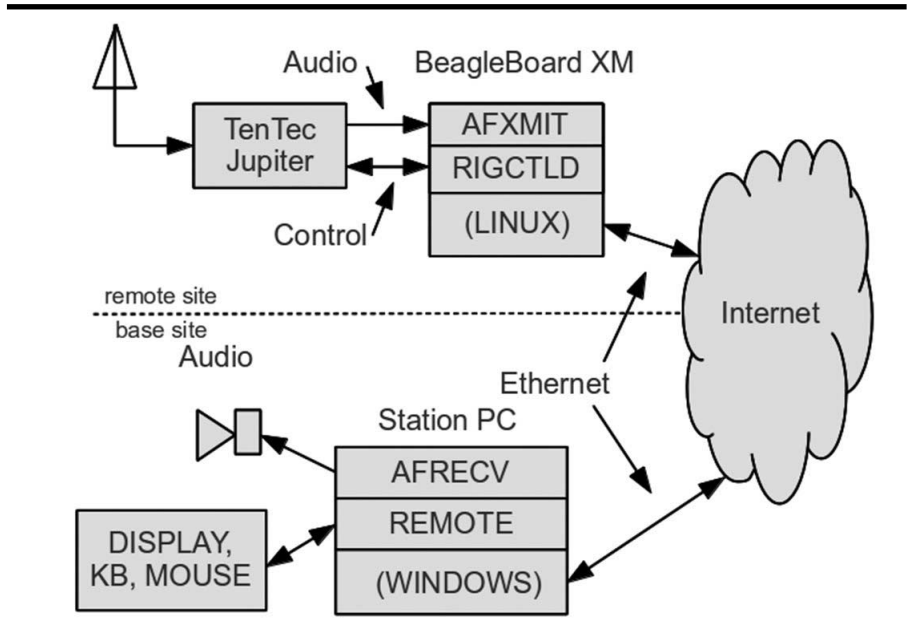

Figure 1 — This block diagram shows an overview of the remote receiver system.

a microSD flash memory module serving as a "hard drive" (typically 8 GB). The BBXM will support a variety of operating systems.[3] We installed *Ubuntu Linux*, choosing to operate in a "headless" mode — without video graphics, mouse or keyboard. In normal operation, the BBXM only communicates via Ethernet to the base station. (A video graphics port and serial I/O port are available if needed for development or testing.)

One required feature for the receiver is full computer aided transceiver (CAT) control.[4] That eliminated some fine older radios that might have been used. In the end, we selected a Ten-Tec Jupiter transceiver. Its serial I/O port can be connected to the BeagleBoard through a USB-to-Serial converter. (The BBXM's on-board serial port was reserved for maintenance use.)

The BBXM and a power supply were packaged in a small case. Figure 2 shows the remote system and the Jupiter transceiver. Figure 3 shows the internal layout of the BBXM, power supply and audio isolation board. The power supply offers 12 V dc at 2 A to run the transceiver in receive mode.

At the base station location, a 3 GHz Pentium IV PC running *Windows XP* is used as the main station computer, supporting the remote receive operation as well as other station functions.

Software Design for the BeagleBoard XM

We have worked out some software approaches that may be useful to W1HQ and the amateur community. We are providing the software under the GNU General Public License, so you are free to adapt it as you like for your own use and education.[5]

Control

Remote control of the Jupiter transceiver was easily implemented with the Hamlib system.[6] Hamlib provides an interface library coded in *C* that defines an application programming interface (API), allowing an application program to control any of a large number of different Amateur Radio transceivers, such as the Jupiter. Hamlib also provides an Internet server that accepts rig commands via Internet TCP/IP packets in a simple text format. This server (**rigctld**) runs on the BeagleBoard and accepts connections from the base computer over a specified TCP/IP port. Hamlib is available for download from most *Linux* repositories. (Source code and *Windows* versions are also available, but they were not needed for our project.) A Google search for hamlib will return numerous sites for downloads and documentation.

Audio

Creating an efficient Internet audio channel was the largest task in this project. There are numerous streaming audio systems available, but these can have long latencies that don't allow interactive Amateur Radio communications. We might have chosen to work with an open source VOIP scheme or a service like *Skype*, but in the end we chose to develop a simple transmission scheme that works well in this application.

Figure 2 — Here is the remote control box containing the BeagleBoard XM computer and the Ten Tec Jupiter transceiver.

Our voice-to-Internet application is called **afxmit**. It is written in *C* using the PortAudio framework.[7] **Afxmit** samples the receiver audio at 8 kHz (the lowest standard rate) with 16 bit resolution. (This is overkill for normal Amateur Radio operations, giving frequency response to 4 kHz and much more dynamic range than amateur channels normally require.) The program uses the Speex codec to compress the audio before sending it to the Internet.[8] With normal settings, we require only about 3 kB/s of Internet bandwidth, including Hamlib communications.

Note that the Speex codec in this application solves a very different problem than codecs used for digital voice over RF channels, such as Codec2.[9] For RF communications, a modem sends data *on the air*, using the lowest possible bit rate while being resistant to propagation changes and interference. In our case, we are sending data *on the Internet*, seeking a low bit rate while preserving good communications fidelity.

Given a compressed bit stream from Speex, how should it be transmitted? We first looked at the Transmission Control Protocol often called TCP/IP, which would guarantee error-free end-to-end transfer.[10] The features of TCP, however, actually make things harder for our application, because if there are transmission errors, data will be retransmitted, and the audio stream will be delayed. For real-time streaming data like ours, it's usually better to forget about missing packets or other errors and just move on with the good data.

In the end, we adopted the User Datagram Protocol (UDP) for audio transmission. Audio data is taken by the BBXM audio system, transmitted by UDP, and finally played in real time by the base station computer sound card. The two audio clocks (input sound card and output sound card) will always have slightly different frequencies, meaning that the playback buffer will eventually overflow or starve. With appropriate framing, the receiver can skip data or insert silent data as needed.

Speex provides a packet every 20 ms, and the BeagleBoard buffers four of these for a UDP packet every 80 ms. There is no guarantee of delivery, and no acknowledgement comes back from the receiving computer. This is a low-overhead method that works well with our real-time audio transfer scheme. An occasional missing packet or an over- or under-flow is no problem. We can drop an occasional excess packet or insert silence as needed on a typical HF channel, and the operator usually does not notice.

Afxmit listens on its UDP port for commands from the base station computer to set the codec parameters and to start or stop sending audio.

Addressability and Security

The remote BBXM has to be addressable from the Internet. Since we are relying on a host Internet account, that means that the host Amateur Radio operator's Internet router needs to be addressable. We use the dynamic DNS addressing provided by Dyn to provide data for an address like *my-address.dyndns.org*.[11] A small program on the BeagleBoard runs occasionally to ensure that the Dyn DNS database has the actual IP of the host's service.[12] Using the host's current numerical address might work for a time without DNS, but ISPs sometimes change your IP address without notice.

To make the BeagleBoard available to the outside network, we also need to insert some "pinholes" in the host router's firewall configuration. Incoming traffic for ports assigned to Secure Shell (SSH) terminal traffic, **rigctld**, and **afxmit** is directed to corresponding ports of the BBXM. With this firewall setup, remote receiver traffic goes straight from the host operator's router to the BeagleBoard and does not depend on the host's own computer in any way.

Login to the BeagleBoard is supported using secure shell (SSH), which provides a cryptographically secure link. Because the base station software initiates its sessions automatically, SSH logons are mainly useful for system maintenance.

A similar level of security can be provided for **rigctld** if needed, by implementing an "SSH tunnel" for traffic for port 4532. Unfortunately, UDP traffic cannot be tunneled this way, so our audio link will always be somewhat insecure without some further work. (Software could restrict UDP responses to certain IP addresses, for example.)

Base Station Computer Software

The main remaining component of our system is the software application **remote** that runs in the base station computer. It is written in *Python,* a powerful language that runs well on both *Linux* and *Windows* computers.[13] With the help of the WxWidgets framework for graphical user interface, **remote** provides a virtual control panel for the remote receiver, shown in Figure 4.[14] While this control panel is tailored for the Ten Tec Jupiter transceiver, it could be used with most other CAT controllable rigs with minor changes.

We might highlight the FSpin class in our *Python* code. This defines the control widget that manages the setting of the VFO frequency. It offers "up" and "down" buttons for each frequency digit that allow rapid tuning to a desired frequency. When the FSpin control has keyboard focus, it will also respond to the mouse wheel, which will tune a particular digit up or down. Direct frequency entry from the keyboard is also supported.

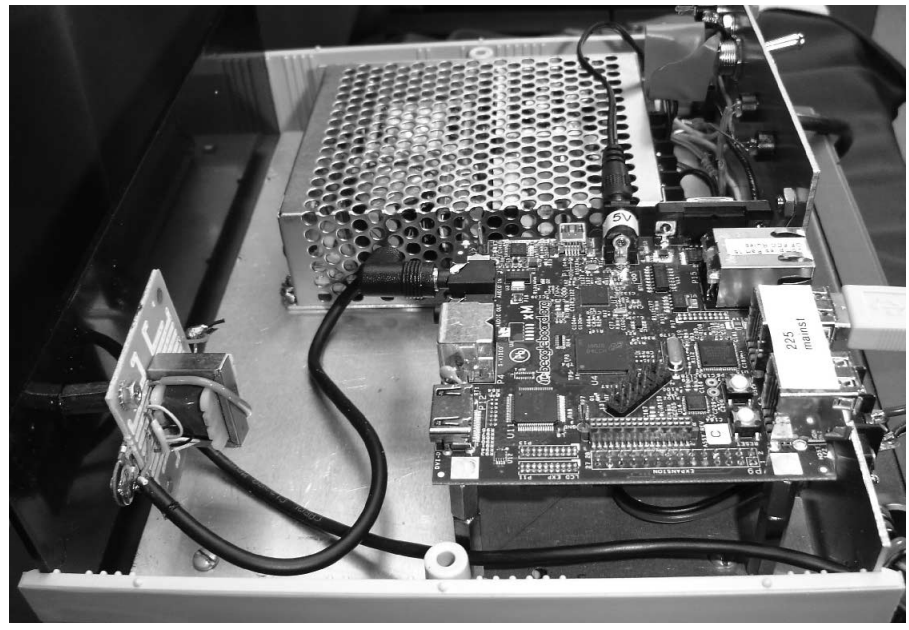

Figure 3 — This is an internal view of the remote control box. The BeagleBoard XM is in the lower right corner of the photo.

Another feature is the SMeter class. It is adapted from the wxWidgets SpeedMeter. Because our Hamlib connection will only offer a new S-Meter reading every second or so, we cannot provide the elegance of some recent transceivers with LCD meter emulations. (A convincing meter emulation requires bandwidth!) Still, it is a useful display, and it is calibrated.

Remote Communication

The user can run **remote** in either a *Windows* or *Linux* based computer environment. A number of things need to happen that are outside the *Python* code. First, **remote** launches a *Windows* batch file (or *Linux* shell script) that starts an SSH session with the remote computer. In a *Windows* machine, we use **PuTTY** and **plink** for this purpose.[15] The SSH command runs a shell script on the BeagleBoard remote that starts the two server processes, **rigctld** and **afxmit**, setting up the remote to listen for commands from the base computer. **Remote** has a module that manages Hamlib commands for **rigctld** via a TCP/IP connection.

A separate *C++* program, **afrecv**, receives the UDP audio data sent from **afxmit**. For a *Linux* base station, **afrecv** was straightforward to code using the PortAudio library. Alas, we needed a version of **afrecv** to run under *Windows*. Could we port PortAudio and Speex to *Windows*? It turned out that porting Speex was straightforward, but we had problems with PortAudio. We finally selected the *Windows* XAudio2 API instead.[16] XAudio2 is part of the Microsoft *DirectX* Software Development Kit. It is a gaming-oriented audio system that is functionally close enough to PortAudio that **afrecv** can invoke either library by conditional compilation. **Afrecv** and Speex project files for Microsoft *Visual C++ 2010 Express* are provided with our other code at the website mentioned earlier.

Remote directs **afrecv** to tell **afxmit** to start and stop audio transmission on behalf of the user. This involves running a batch file (shell script), which can take noticeable time to complete, especially under *Windows*. For faster transmit/receive switching, we provide a command that simply mutes the PC audio, leaving the UDP stream running.

Interface Complication

We originally connected the Ten-Tec Jupiter headphone output straight to the BeagleBoard audio input, but there was some weird behavior. A loud static crash in the receiver would cause the computer to reboot. After some head-scratching and consulting of schematics, we saw that the Jupiter's speaker output and headphone jack are driven by a bridge circuit, with both terminals floating. Our static crash was injecting unfriendly current surges into the computer's ground plane. The solution was an audio isolation transformer, which you can see in the left foreground of Figure 3.

Operating Results

The remote system is currently installed at my home, about 30 miles away from

ARRL Headquarters, where it uses a small fraction of a cable TV Internet connection. The Jupiter is normally connected to an 80 m dipole, which is usable for reception across the HF bands. Differential propagation (favoring the transmitting site over the receiving site, or vice versa) has not been a problem so far, but we hope to eventually move the remote site closer to W1HQ.

The remote capability has given W1HQ the option to be active in the face of severe local interference from nearby W1AW, at a modest cost. What is it like to use the system, and what would we do differently if we were doing it again?

The audio quality is good — good enough to support PSK31, we believe, though this has not been fully tested. One issue is the delay time (latency) when tuning the receiver. Tuning up and down the band is slower than it would be on a local rig. Some of this delay is caused by the Jupiter CAT interface, which (like many radios) uses relatively slow serial 57.6 kb/s communication. There is also a delay in audio transmission caused by audio buffering, the Internet, and the Speex compression we use. We see about ½ second total delay. That's enough to confuse an operator monitoring transmission in real time or scanning the band. It's not a problem for typical transmit/receive switching, but it could be an issue in rapid-fire DX or contest operation. There are software parameters, particularly buffering ratios, that can be adjusted if lower delays are required. The minimum packet length with Speex compression is 20 ms, but compression can be turned off if needed. In general, increased responsiveness requires less buffering, more packets, and more Internet bandwidth.

If we were doing the project over, we would naturally look at newer computer options, if not newer radios. The BBXM is still viable and attractive because of its on-board audio system and its multiple I/O ports. There are less expensive, smaller boards now available such as the BeagleBone and the Raspberry Pi, among others.[17] These both require off-board audio adapters, but a simple USB audio "dongle" will probably work for our 8 kHz monaural requirement. You should be able to reduce the hardware cost and reduce system size with one of these choices.

Python and *wxWidgets* work well for the base station control program (**remote**). With more recent versions of *PyAudio* (a *Python* wrapping of PortAudio), it should be possible to integrate the **afrecv** function into the *Python* main program. This would be especially attractive to a *Linux* programmer who (like your author) needs to port the application to *Windows*. *Python* and *wxWidgets* work well on both platforms.

Building our own remote system has been an interesting and educational challenge, combining a number of technologies, and not least, introducing a new capability to our club station. Adding a transmit capability for two-way remote operation should not be too difficult, using some of the components presented here. The base station code can be readily adapted to other radios, or made to run in another tiny computer like the BeagleBone or Raspberry Pi.

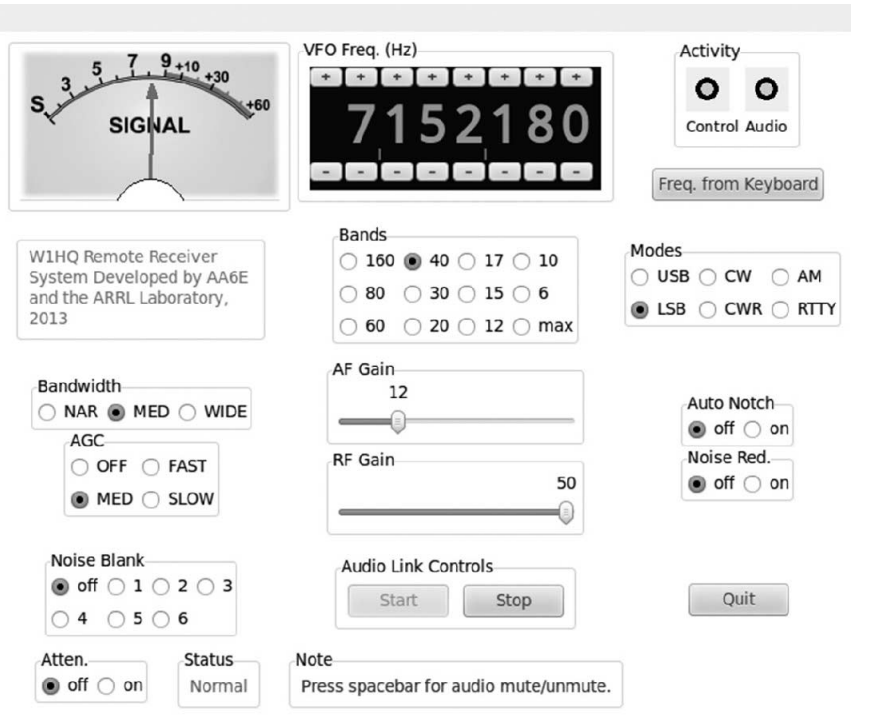

Figure 4 — This screenshot shows the virtual control panel, as displayed on the W1HQ computer.

Acknowledgements

I am very grateful for the support and encouragement of Ed Hare, W1RFI, and the entire ARRL Laboratory staff. Bob Allison,, WB1GCM, kindly provided the photographs.

Martin Ewing was first licensed in 1957 as K5MXF in New Mexico. He studied at Swarthmore College and received a PhD in Physics at the Massachusetts Institute of Technology, specializing in Radio Astronomy, the cosmic microwave background, and pulsars. He joined Caltech Radio Astronomy working on Very Long Baseline Interferometry, designing digital correlators and developing a version of the Forth *computer language for real-time control. At Yale University, he served as Director of Information Technology in the Faculty of Engineering. An ARRL Member, Martin became an ARRL volunteer and Technical Advisor after retiring. At the ARRL Laboratory, he works with Software Defined Radio and applications of small* Linux *computers.*

Notes

[1]The software files associated with this article are available for download from the ARRL QEX files website. Go to **www.arrl.org/qexfiles** and look for the file **1x14_Ewing.zip**.

[2]See **http://beagleboard.org**. More recently, the BeagleBone series has been introduced. See the website for additional information.

[3]Go to: **http://elinux.org/BeagleBoard**.

[4]Computer Aided Transceiver (CAT) is Yaesu's term for a radio's computer control capability. It is now used as a generic term.

[5]For complete details of the GNU software license, go to: **www.gnu.org/licenses/gpl.htm**.

[6]See **http://hamlib.org**.

[7]See **http://portaudio.org**.

[8]For more information, see **http://speex.org**. Since this project began, Speex has been supplanted by Opus, **http://opus-codec.org**, which is recommended for new applications.

[9]See **www.codec2.org** and **www.rowetel.com/blog/?page_id=2458**.

[10]This, and many other computer terms, can be found by searching **http://en.wikipedia.org**.

[11]See **http://dyn.com**.

[12]Martin Ewing, AA6E, "DNS Choices for Your Ham Server," *QST*, Nov 2006, pp 77-78.

[13]See **www.python.org**.

[14]See **www.wxwidgets.org**.

[15]For more information about PuTTY and to download the files, go to: **www.chiark.greenend.org.uk/~sgtatham/putty/download.html**.

[16]You can learn more about Microsoft XAudio2 and download the files at: **msdn.microsoft.com/en-us/library/windows/desktop/hh405049(v=vs.85).aspx**.

[17]For more information about the Raspberry Pi computer, go to: **www.raspberrypi.org**.