Ray Mack, W5IFS

17060 Conway Springs, Austin, TX 78717: **w5ifs@arrl.net**

SDR Simplified

Step One towards a working SDR.

Beginning A Real Radio

We have looked at a lot of topics related to SDR and DSP over the past years. It is time to actually build a radio. The goal is to design a VLF/LF/MF/HF radio that puts the transmitter and receiver as close as possible to the antenna. The first step is to create a desktop radio that handles SSB voice, CW, and AM, and covers the frequency range from 10 kHz to 30 MHz. We will take the system step by step and assemble a complete working transceiver.

Human Factors in Device Design

Perhaps I am old school, but I believe that radios should be radios and computers should be computers. From a human factors perspective, controlling a radio with a computer means you have multiple interconnect and control interfaces to manage, so the system does neither very well, especially when you want to use the computer as a computer! A single device that encapsulates the entire user interface is much simpler to manage as a user. The worst interface I have seen in a tool was an early computer controlled Hewlett-Packard oscilloscope. There was one adjust knob and one select button. Everything else was done with menus on a screen. It was impossible to make a set of measurements without constantly taking your eyes off your work. I was lucky I didn't fry more circuits from having the probe slip. I quickly went back to a much older analog scope because it was easier to use by feel alone.

There are a number of radios currently on the market that I believe share a similar failure of human factors. One hand-held radio has three levels of menus with each of the 16 buttons on the panel having at least three different mode-specific functions. You have to look directly at the radio in order to control just about every function, and you will also need the manual at hand in order to navigate the menu structure for even simple top-level functions. For me, if I cannot do the basic control functions within two minutes of walking up to a radio for the first time, the interface has failed.

Especially in a mobile radio, it is important that we be able to control the common functions with a small number of dedicated knobs and buttons. At the top level of control, we need an on/off button, volume control, a frequency control, and probably band select. These functions need to be discernable without looking at the radio! The next level, which also needs dedicated buttons, would include things such as noise blanking, selectivity, A/B band selection, and transmission mode. These might need a quick glance at the screen for confirmation, but should never require visual engagement to perform the function. Each time one changes frequency (either band change or tuning within a band) the radio should give a clear indication on the screen of both the frequency and mode with nothing more than a quick glance. A desktop radio can be slightly more complicated because we can devote our entire attention to the user interface. The user interface should not be so complicated, however, that it gets in the way of the primary function of the radio, which is to communicate!

Figure 1 shows the front panel design of my proposed desktop radio. The central feature is the 7 inch TFT display, which provides 800 × 480 pixels.[1] The industry calls this WVGA resolution (Wide VGA). My first design for a radio used a 4.3 inch WQVGA resolution screen (Wide-Quarter VGA) which is 480 × 272 pixels because the displays were inexpensive and a large number of manufacturers make equivalent devices. The size and resolution became popular because it was used in the Playstation Portable game system. Now, smart phones are our friend. The proliferation of smart phones has pushed resolution to WVGA in both 5 inch and 7 inch sizes from the same manufacturers and at reasonable prices.

The button and knob usage and layout is inspired by any number of analog radios I have used over the years. The main difference is that the tuning knob is moved

[1]Notes appear on page 44.

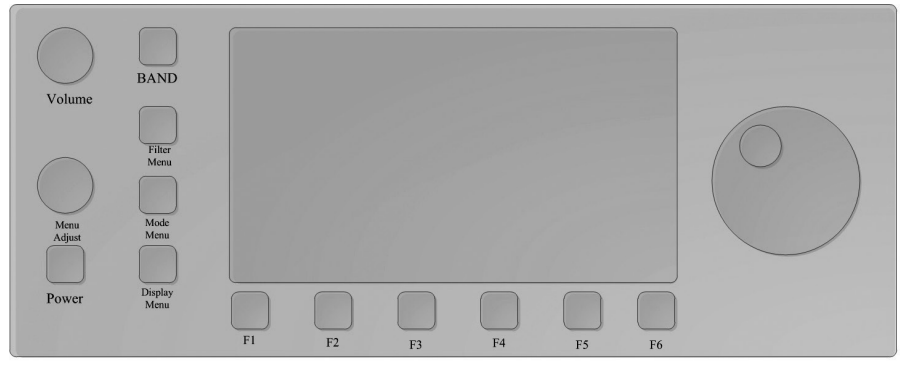

Figure 1 — Here is my proposed front panel layout for a self contained SDR. The display panel has 800 × 480 pixels, to give a full color internal computer display of control functions and data. The menu keys along the left have dedicated functions and the keys along the bottom are soft function keys.

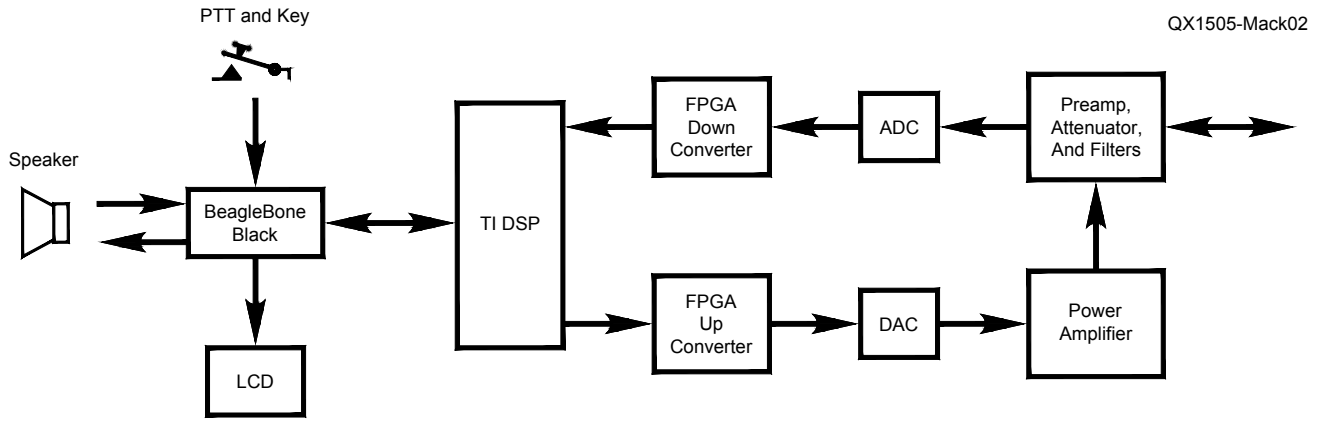

Figure 2 — A block diagram of the proposed transceiver. There are three main processing elements: the BeagleBone Black *Linux* computer, the Analog Devices DSP, and the FPGA hardware accelerator.

to the right side to accommodate the full size screen. Almost all analog desktop radios place the tuning knob in the center, with small readouts along the top of the panel. Feel free to comment on any aspects of the layout. The tuning knob is the focus of our laboratory exercise this time.

Top Level System Design

Figure 2 is the block diagram of the transceiver system. We are still going to want all of the graphics-intensive functions that we see on a computer controlled SDR, such as ones using *PowerSDR*, so we will need a computer that can run a modified version of one of those programs. The BeagleBone Black (just BeagleBone from here) is an inexpensive prototype computer that fits the bill for our needs. In fact, it is a complete *Linux* computer on a 2 × 3.5 inch circuit board. It is based on a 1 GHz Texas Instruments Sitarra AD3359 processor.

The AD3359 has internal support for TFT LCDs for the display, quadrature encoder support for our frequency control knob, and ADC support for our analog controls like volume. All of the normal peripheral interface buses such as USB, I2S, SPI, and I2C are available for the other devices we will need to control. Ethernet is available for debugging and other network uses. The McASP interface ports are actually enhanced I2S ports similar to those used for audio CODECs on sound cards. We will use at least one of those ports for receiving data for waterfall and spectrum displays from the main DSP portion of the radio.

Any computer that does more than the very basic bare metal control is going to need an operating system to help us manage the various software tasks in the system. Some systems such as our DSP portion need exact or very fast response to incoming data. These are called hard real time systems. Even a slip of a few milliseconds can cause problems for the system. Our user interface is less demanding. Our eyes and hands need feedback about 10 to 20 times per second in order for us to perceive that tasks are happening continuously. A difference of 10 to 50 ms will not change our perception of continuous control. These systems are called soft real time.

Windows is far from a real time operating system. It is not uncommon for the mouse pointer to pause noticeably as we move it across the screen. Modern *Linux* is a better system. It includes real time extensions that allow for implementing soft real time systems. Fortunately, the BeagleBone comes with *Linux* installed. *Linux* also provides us with the advantage that many of the SDR functions we will implement for the display are supported in the *Linux* versions of software like *PowerSDR*.

Using the Quadrature Encoder

We will use the quadrature encoder in the same way we use the wheel on the mouse for a PC. Turning one direction will increase the frequency and turning it the other way will decrease frequency. Figure 3 shows several quadrature optical encoders I have collected over the years. An optical encoder is made of an opaque disc with equally spaced slots that let light transmit from one or more LEDs on one side of the disk to two phototransistors on the other side of the disk. The phototransistors are arranged so that there is overlap of both the light phase and the dark phase. The transitions are arranged so that the square wave signals from the two phototransistors are 90° out of phase (hence quadrature encoding). The 90° offset allows the electronic hardware to determine the direction that the encoder is turning. This operation is shown in Figure 4. In our example, when the A signal goes low to high before the B signal goes low to high (and the A signal goes high to low before the B signal goes high to low), the shaft is rotating clockwise. The converse is true; when the B signal goes from low to high before the A signal, the shaft is rotating counterclockwise.

Of course, we can attach the A and B signals to GPIO pins (general purpose I/O) and watch their state every few milliseconds in software, but the load on a CPU to perform this relatively mundane operation is huge. Instead, the AM3359 provides up to three Enhanced Quadrature Encoder Pulse (eQEP) modules that can be used to gather motion data. The eQEP module is useful for both human interface encoders and electric motor feedback. For our purposes, we only need the A and B input pins to allow the module to count pulses up or down. The eQEP module contains the QPOSCNT register, which keeps a running tally of both clockwise and counterclockwise (the British refer to this as *anti-clockwise* in some documents).

The finest resolution optical encoder has 64 pulses per revolution, but those are likely too expensive for use in our radio. A cost effective one (CUI C14D32P-A3, $17.22 at Digi-Key) has 32 pulses per revolution. We can start the position counter register at half scale so that we almost never have to do the logic required to handle overflow or underflow of the register. Each time we read the register, we simply subtract the previous count from the present count to determine how many changes and the direction of the changes since we last read the register.

There is a timer that is part of the eQEP hardware. It is possible to set up the timer to capture the position counter every time the counter matches the compare value. The time for the capture and the position counter can be used to determine the rotation velocity

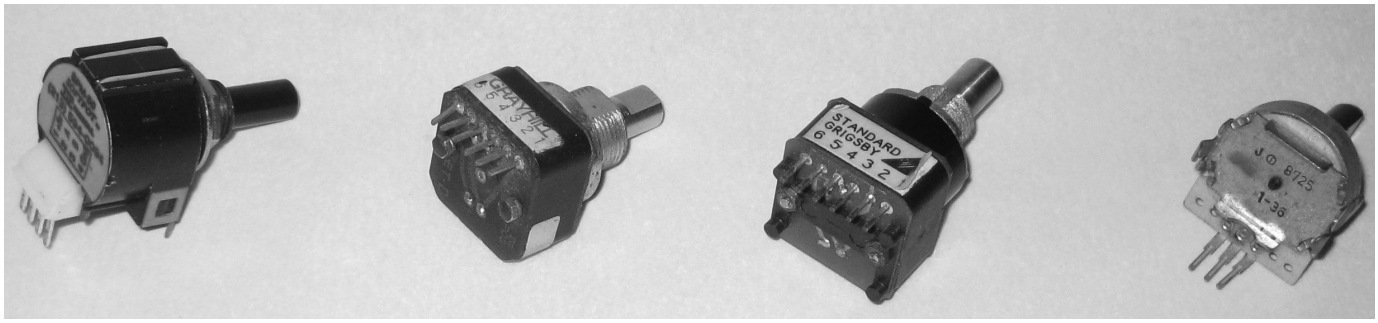

Figure 3 — This photo shows three optical encoders and a mechanical encoder. The two in the middle also contain a push button switch. One is Standard Grigsby and the other is Grayhill, but the traces on the circuit board appear identical.

of the encoder. We can use the velocity to modify our frequency change logic, to change the frequency faster if the velocity is higher. This could be a feature in a future version of the software. Many systems use this velocity mechanism to adapt the speed of change of a variable to the speed of the control knob.

Connecting the Encoder to Hardware

There are three eQEP blocks that show up on various pins depending on how the hardware is configured in software. We need to look at the pins that are available on P8 and P9 of the BeagleBone. We look at the tables on pages 70 and 72 of the BeagleBone user manual to see which pins are used, and their functions. The eQEP2 pins conflict with the eMMC1 pins, which are used for the SD card, and the eQEP1 pins conflict with the LCD. This leaves us with pins 27 and 42 on header P9 as available for eQEP0 operation. Page 71 of the manual indicates that pin 42 is connected to processor pin B12 *and* C18. In order to use this header pin for eQEP0, we need to set B12 for use as an eQEP input and C18 as a GPIO input. We could also make sure the pin is safe by removing the extra resistor as described in the user manual.

The BeagleBone has 3.3 V circuits, so my older 5 V encoders need some level adjustment. The circuit in Figure 5 shows the circuit that lowers the signal to 3.3 V. The user manual warns that using 5 V signals will damage the circuits. If you buy a brand new encoder, you will want to select a version that works at 3.3 V.

Linux Introduction

Linux is a computer operating system that has its roots in the *Unix* operating system from AT&T. If you have ever written a truly useful program, especially in *C*, you have run into bugs that have caused your program to "crash and burn." When you do that with an Arduino project or an SDR running on a dedicated DSP, there are no real consequences. You just debug to find your problem, edit, compile, and restart the debug process. On a computer such as a PC running *Windows* or *Linux*, however, your program could do real damage to things like the hard disk or burn up the CPU if your program crashes in the absence of some sort of protection. *Windows* 3.1 ran on top of *MS-DOS,* which did not have any protection mechanism, and crashing the system was a normal occurrence. The solution is a hardware/software mechanism called protected mode. *Linux* has always run in protected mode and *Windows*, starting with NT 3.1, now runs in protected mode.

The protection comes from special hardware inside the CPU, called the memory management unit and a bit in the status register called the supervisor bit. When the supervisor bit is turned "on," the system is in Supervisor Mode and all software has complete access to everything on the computer: all I/O ports and every memory location. This is what allows rogue software to do damage. When the supervisor bit is "off," the software is in User Mode and the software is very limited in what it can access. This restriction is forced by the memory management unit hardware. In general, a User Mode program cannot touch any I/O ports since those are considered essential to proper operation of the computer. The only resource a User Mode program can touch is a small amount of memory, which is just large enough to run the program. So, for example, you cannot directly toggle a bit on the parallel port or the modem control lines on a serial port from a program that you run from the command line.

So, how can we connect a new device like the quadrature optical encoder to the computer and use it for input in our SDR program? A protected mode operating system implements functions that are part of a device driver, to give us protected access to hardware. A device driver is responsible for collecting information from your User Mode program and then passing that information to the operating system. These are usually a call to read() or write(). Once the read or write call verifies that your information is not going to crash the system, it uses an operating system mechanism to shift the computer from User Mode to Supervisor Mode. The code inside the operating system, running in Supervisor Mode, now has permission to talk directly to the I/O ports of the hardware of the computer. It does the read or write from the hardware and then sends the requested information back up through the protection system and then changes the CPU back into User Mode. Now, you can access the data provided by the device driver. We will use a device driver to connect to the quadrature encoder.

The operating system has basically two functions: protect the computer hardware from defective programs, and allow multiple independent programs to run "simultaneously" on one CPU. On the BeagleBone, *Linux* creates the illusion of

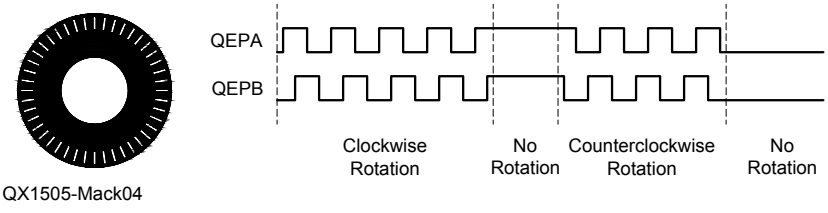

Figure 4 — An illustration of the waveforms for clockwise rotation. The circle shows what the optical disk looks like. Notice that the sensors need to be positioned near the inner part of the clear slot so that the light and dark areas are the same size.

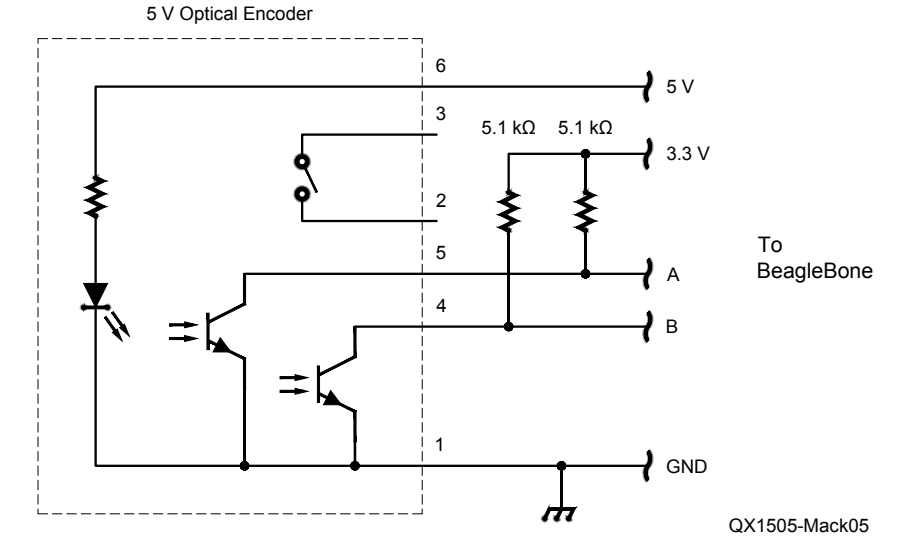

Figure 5 — The schematic of my level shifter to allow use of a 5 V optical encoder in the 3.3 V BeagleBone system.

simultaneous operation of multiple programs by time division multiplexing the operation of multiple user programs. This is easy because programs do not need the CPU while they are waiting on some piece of hardware. During the time one program is waiting, *Linux* runs another program that has work to do.

Linux comes in many different flavors including Debian, Gentoo, and Fedora. These are called distributions or "distros." If you have used desktop *Linux*, it is likely you have used Ubuntu, which is a version of Debian optimized for desktop use. The distribution shipped on the BeagleBone is Debian.

Connecting the Encoder to *Linux*

One of the nice features of *Linux* is that it has a mechanism to install a device driver into the kernel from the User Mode command line. Device drivers that are a part of a distribution are simply compiled into the kernel and are always present. Since a driver for the TI eQEP is not part of the distribution, we must design our own driver and install it from the command line.

A device driver must implement the init() and exit() functions as a minimal set. Of course, such a driver would not be especially useful. The read() and write() functions are required as the next level of operation to be useful. The write() function will simply return without any operation since there is no actual write that can occur. It is included so a call to the generic write() function from User Mode will not cause an error. There are two final functions needed to provide the mechanism to glue a User Mode program to our driver: open() and close().

Listing 1 shows the init() and exit() functions. The program listings and various other files associated with this article are available for download from the ARRL *QEX* files website.[2] The init() function must set up the hardware and do the other tasks to connect the hardware to the kernel. The exit() function can be trivial because we will have minimal resources to release or other cleanup to do. In fact, normal operation will never exit from the driver. So exit() only needs to disconnect the /proc/eqep file from the kernel and then exit. We put kprintf() calls in these two functions so we can see the results of installing or removing the driver on the console as a sanity check.

The init() function must connect our driver to a device file so we can open the file as a character device in a User Mode program. The first operation is to tell the kernel where to put the interface for the device file. It is convention that installable drivers are placed in the /proc directory. This is the location of the driver for User Mode program use; it is not where the actual binary file (eqep.o) is located. The kernel also needs us to connect our driver data and functions and tell it who owns the file. Next, the init function must set up the eQED hardware for use by the driver. We set the mode for the eQEP0 A and B input pins to MODE1 and then turn on the eQEP hardware.

We need to decide how to encode the data from the encoder into data that is read from the file. The easiest way is to simply encode the data as a signed 8 bit value. It is very unlikely that we can spin the knob fast enough to generate more than 127 pulses between reads of the device file. To that end, the read() function will read the position register and subtract the half value from the register. Then the value will be truncated to an 8 bit value. This makes the logic simple, as shown in Listing 2. For now, the write() function does nothing. If we decide later that it would be nice to set parameters of operation, we can extend the driver so writes to the device file affect its operation.

We compile and link the program and create a file called eqep.o. You can place it in any convenient directory. I prefer /root just because that is where I put things when using the QNX operating system. Change to /root and type the command:

insmod eqep.o

You should see the message from the init() function show up on the console. We will now be able to interact with the eQEP driver by opening and reading the file /proc/eqep.

An Example User Mode Program

Listing 3 is an example User Mode program that looks at the eQEP driver every 100ms and displays an absolute frequency that is adjusted by turning the encoder. We initialize the program to display 7100.00 kHz. Each position change of our encoder will adjust the frequency up or down by 0.01 kHz.

Making a Build Computer

It is possible to set up the BeagleBone to build the software we need, but it would require a USB hard disk and other accessories to make it a real computer. It is much easier to build using an existing computer (workstation) and transfer the drivers and programs to the BeagleBone (target). My suggestion is to set up a computer with a dual boot of *Windows* and Ubuntu *Linux*. The current version of "long term support" *Linux* is 14.04. Ubuntu is supported by Canonical which is a company in the United Kingdom. A lot of what makes Ubuntu a truly usable desktop solution is because of their efforts and support. You can find much information on the Internet for support of BeagleBone and Ubuntu.

I have both *Windows 7* and *Windows 8* systems that I have set up as dual boot. I recommend avoiding a *Windows 8* machine if you can. There are issues with getting the dual boot menu to populate correctly. My *Windows 8* machine requires that I power cycle and hit <esc> to force it into the BIOS mode, where I select <F9> to go to the boot menu. I can probably fix it at some point, but I prefer working on SDR projects.

The following is an abbreviated set of instructions for those who feel comfortable doing normal computer building operations. There are very detailed instructions, notes,

and pointers to on-line resources in the zip file on the ARRL *QEX* files website site for this issue. See Note 2.

The first step is to make sure you are connected to the Internet for the whole process. There are numerous updates that must come from the Internet. You download the ISO image from the Ubuntu site and burn a DVD (since it is 900 MB of data, it won't fit on a CD). Ubuntu is free, but the $16 suggested donation is a small amount to help them maintain stable, working software. You will need to use the *Windows Disk Manager* utility to reduce the size of your hard disk for *Windows*, to make an unused partition. I set mine to 250 GB out of 750 GB, but about 100 MB should be enough. My system is almost fully set up, and uses 6 MB of space.

Now that you have an unused partition, you need to power cycle and boot from your DVD. If your system does not boot the DVD, start up in the BIOS configuration mode. This varies from computer to computer. It is always <esc>, <del>, or a function key. You need to make sure that your BIOS is set up to boot from the DVD as the first option. Your system will boot into a small program called GRUB that lets you select the operation to perform. You want to "install *Linux*" rather than "try *Linux*." The system will start up and prompt you for various actions. You want to install *Linux* beside *Windows*, so you can choose which way to boot each time. The system will prompt you to install in the newly created empty partition. Let the system complete the task and you will have a working *Linux* computer. Be sure to write down the password that you select! This is the administrator (root) password, which you will use very frequently. I also recommend that you select the "automatically log me in" option at that point. When the system re-boots, select *Linux* from the boot menu. When you are prompted to update Ubuntu, select yes, because we need the extra packages that get installed at this point.

The Ubuntu desktop looks similar to a *Windows* desktop, but has some unfamiliar characteristics. The "X" to kill a program is on the left rather than the right of each program window. There is a task bar, but it operates differently from *Windows*. Basically it is where programs are "pinned." The top left icon is the Ubuntu "search your computer" program. Click on that icon and type in "ClassicMenu." Double click the program icon that is displayed. That will place a small "Ubuntu icon" on the top right of the menu bar. Selecting that icon now gives you something much more like the *Windows* "Start Menu."

The next step is to click on the settings icon (the adjustable wrench on top of a gear) on the left bar. Double click on Appearance and change "Show the menus for a window" from "in the menu bar" to "in the window's title bar." This makes the menus look a lot more like *Windows*. The system still automatically hides the menu, but it stays on the window instead of the top of the display. Programs in Ubuntu do not typically have "Close" or "Exit" buttons. Changes take effect immediately in many cases. The other thing I found different is the lack of a scroll bar on windows. The scroll tool automatically hides. It is at the right side of each window and pops up if you hover the cursor above it. Once it displays, it operates similarly to a *Windows* scroll bar.

We use the command console for most software development tasks in *Linux*. This is a holdover from old *Unix* days. Select the ClassicMenu icon/Utilities/Terminal to bring up a command window. One really nice feature of the terminal is that it remembers your command history even across power cycles.

We need to install a lot of software developer programs at this point. First, we need the cross compiler, arm-linux-gnueabi-gcc, and associated tools. Ubuntu was installed with the native x86 version of the compiler, but we need the ARM version for the BeagleBone. The command to do this is: sudo apt-get install gcc-arm-linux-gnueabi You can think of "sudo" as meaning "do this command as super user." It changes your security permission from normal user to super user for the next 15 minutes. Only super users are allowed to do certain operations, such as install new software. This is another mechanism to keep you from accidentally killing your computer. After you type the command, the computer will ask you for your password.

Next, you need to get Eclipse, which is an IDE (integrated development environment) for building programs. This is the same IDE that we saw when we used Code Composer for the Texas Instruments DSP board. The command here is: sudo apt-get install eclipse eclipse-cdt g++ gcc This gets and installs the Eclipse IDE and the plugin for *C* development as well as updating the g++ and gcc compilers in case they are out of date. We need a few more utilities: sudo apt-get install git build-essential lzop u-boot-tools

Git is a configuration management tool that we need to use to retrieve files for the kernel from the public repository for the BeagleBone *Linux* distribution. The files are compressed so we need the lzop tool to decompress those files, and we might need the u-boot files for later operations. The next step is to create a build directory for the *Linux* kernel and other necessary software. I choose to call it bbb_linux, but you can pick any name you like. Do the following commands:

```
sudo mkdir /bbb_linux
sudo chown ray /bbb_linux
cd /bbb_linux
```

Of course, substitute your login name for "ray."

The last operation before we can actually do some work is to retrieve the kernel directory tree: git clone git://github.com/beaglebone/linux

This git command will create a new directory /bbb_linux/linux and store the *Linux* tree there. The next step is to build the kernel. We need to build the kernel and collect the necessary header files. The instructions for doing that are more complicated and not amenable to including here. Please download the files from the ARRL *QEX* files website for a truly detailed sequence. (See Note 2.) A big problem is that much of what you will find on the Internet is out of date or incomplete. My instructions capture the errors you can uncover and how to avoid them, at least at the time of this writing!

Quick Start for the BeagleBone

You will want to read the quick start instructions on the BeagleBone website.[3] The most important admonition at this point is to *always* use the power switch next to the Ethernet connector to shut down the BeagleBone. Failure to do so can leave the power controller IC in an indeterminate state. The first step is to plug your BeagleBone into the PC using just the USB cable. The BeagleBone will show up as a removable drive with various files for *Windows*. You will not be able to see the *Linux* file system, though. The next step is to install the USB driver that lets you connect to the BeagleBone with an IP address through USB. The IP address is always 192.168.7.2. You can connect to the BeagleBone through Chrome or Firefox (not Internet Explorer) by typing in the IP address.

The next steps install a new kernel image from the repository. Step one is to download the latest image from the beagleboard.org website. Store it someplace convenient like the desktop on the PC. Next, you need to get *7-zip* from its website and install it.[4] Use *7-zip* to convert the compressed .img.xz file on your desktop to a 4 GB uncompressed .img file. Get the program *Win32imager* from Ubuntu. Be very careful when you run *Win32imager*. It only lets you write removable media, but if you have multiple devices plugged in, you could accidentally kill the wrong device. Plug the micro-SD card into the adapter and plug that into your PC. Use *Win32imager* to write the image to the SD card.

Now is the time to connect the infrastructure for your BeagleBone. Turn off the BeagleBone. Disconnect the USB cable from the PC. Attach a micro-HDMI cable from the board to your monitor (you might need a HDMI to DVI adapter), a USB hub, a keyboard, a mouse, and a network cable. Insert the micro-SD card with the new image and a 5 V power supply. This is a place where the current instructions on the Internet are wrong. The image is not set up to automatically copy the new image from the SD card into the internal eMMC flash drive. You need to navigate to Applications/Accessories and double click on LXTerminal. You need to change to the /media/rootfs directory. Use the "sudo chown debian boot" command so you can save files in the directory. Now change to the boot directory and "sudo chown debian uEnv.txt" so you can edit the file. The easiest way is to navigate to the file using the file manager, right click, and select edit. Find the line that begins "#cmdline=init=/opt" and remove the "#". Save the file. Shutdown the BeagleBone and restart it. The new image should take hold of the boot process and you will see the various operations on the monitor. Once it finishes, the system will power off, so you know it is finished. The uEnv.txt file will be removed from the rootfs/boot directory so you do not have to worry about the process happening twice. You can see which install is active on your BeagleBone by the command "cat /opt/dogtag".

Plans for Next Time

My inspiration for the next step is the yearly scheduled transmissions from the Alexanderson alternator in Sweden, which transmits every year on the last Sunday in June or first Sunday in July (Alexanderson Day) on 17.2 kHz. I tried unsuccessfully to listen for it during a special transmission event on February 13, 2015. I am hoping to have a minimal SDR up and running to listen for it this summer. The system will use the BeagleBone, the LCD, and an audio codec. It should also be useful for listening to WWVB on 60 kHz. I was able to copy WWVB quite easily from the parking lot at work in Cedar Park, TX (an Austin suburb) using an old tube based test receiver.

Notes

[1]Thin film transistor (TFT) is a special type of LCD in which each pixel of the display is controlled by an individual transistor deposited on the glass surface. All LCD TV screens use TFT glass. A consequence of the way TFT works is that it requires a constant refresh of the data to be displayed, a horizontal clock, a vertical clock, and a pixel clock. These clocks create a signal that is very similar to a standard analog TV signal. The Sitarra processors include internal hardware that produces these signals with no software involvement.

[2]The program listings and other various 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 **5x15_Mack_SDR.zip**.

[3]The top level of support for the BeagleBone Black is found at **www.beagleboard.org**. The newest image for the BeagleBone is found at **www.beagleboard.org/latest-images**.

[4]You can find more information about the 7-Zip program, and download the program file at **www.7-zip.org**.